HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Web-Component-basierte Entwicklung mit Polymer

Sandro Tonon

Konstanz, 15.02.2016

BACHELORARBEIT

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

an der

Hochschule Konstanz

Technik, Wirtschaft und Gestaltung

Fakultät Informatik

Studiengang Angewandte Informatik

Thema: **Web-Component-basierte Entwicklung mit Polymer**

Bachelorkandidat: Sandro Tonon, Allemannenstraße 10, 78467 Konstanz

1. Prüfer: Prof. Dr. Marko Boger
2. Prüfer: Dipl. Inf. (FH) Andreas Maurer

Ausgabedatum: 15.10.2015

Abgabedatum: 15.02.2016

Zusammenfassung (Abstract)

Thema:	Web-Component-basierte Entwicklung mit Polymer
Bachelorkandidat:	Sandro Tonon
Firma:	Seitenbau GmbH
Betreuer:	Prof. Dr. Marko Boger Dipl. Inf. (FH) Andreas Maurer
Abgabedatum:	15.02.2016
Schlagworte:	Web Components, Polymer, AngularJS, JavaScript, HTML, Custom Elements, HTML Templates, Shadow DOM, HTML Imports

In der heutigen Webentwicklung kommt es häufig vor, dass für diverse Probleme oftmals die gleiche, oder eine ähnliche Lösung entwickelt werden muss, ohne dass es hierfür ein Plugin-System gibt. Diesem Problem widmen sich die Web Components, welche dieses bereitstellen. Der Begriff Web Components ist dabei ein Dachbegriff für mehrere entstehende Standards, welche jedoch noch nicht von allen Browsern unterstützt werden. Sie sollen es ermöglichen, wartbare, interoperable und gekapselte Komponenten zu entwickeln.

Die von Google entwickelte Library "Polymer" setzt diese Technologien um und soll den Umgang mit ihnen vereinfachen sowie sie auf ältere Browser portieren. Sie ermöglicht es, gekapselte Komponenten zu entwickeln, welche wiederum von Komponenten verwendet oder mit anderen Komponenten verbunden werden können. Dies ermöglicht die Realisierung komplexer Applikationen.

Mittels Beispielimplementierungen von Komponenten sowohl ohne, als auch mit Polymer sowie einer ähnlichen Implementierung mit dem Framework "AngularJS", werden die Unterschiede der jeweiligen Ansätze dargestellt.

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Sandro Tonon, geboren am 02.07.1990 in Waldshut-Tiengen*, dass ich

(1) meine Bachelorarbeit mit dem Titel

Web-Component-basierte Entwicklung mit Polymer

selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 15.02.2016

(Unterschrift)

Inhaltsverzeichnis

1 Einleitung

Der Begriff “Web Components” ist ein Dachbegriff für mehrere entstehende Standards [Schaffranek 2014], welche es für Webentwickler ermöglichen sollen, komplexe Anwendungsentwicklungen mit einer neuen Sammlung an Werkzeugen zu vereinfachen. Diese sollen die Wartbarkeit, Interoperabilität und Kapselung verbessern und somit ein Plugin-System für das Web schaffen. Durch die neuen Standards soll das Web zu einer Plattform werden, die es ermöglicht, die Web-Sprache Hypertext Markup Language (HTML) zu erweitern. Dies ist bisher nicht möglich, da die HTML-Technologie - und somit die Möglichkeiten, HTML-Tags zu benutzen - vom World Wide Web Consortium (W3C) definiert und standardisiert wird. Unter den wichtigsten der neuen Standards sind die folgenden 4 Technologien aufzuführen: Custom Elements, Shadow Document Object Model (DOM), HTML Templates und HTML Imports. Custom Elements ermöglichen es einem Webentwickler, eigene HTML-Tags und deren Verhalten zu definieren, oder bereits vorhandene oder native HTML-Tags zu erweitern. Das Shadow DOM stellt ein Sub-DOM in einem HTML-Element bereit, welches dem Element zugehöriges Markup, Cascading Style Sheets (CSS) und JavaScript kapselt. HTML Templates stellen, wie der Name impliziert, einen Template-Mechanismus für HTML bereit und HTML Imports erlauben das Laden von HTML-Dokumenten in andere HTML-Dokumente. [Kröner 2014b], [Kröner 2014a]

Diese neuen Technologien werden allerdings noch nicht vollständig von allen populären Browsern, zu welchen Google Chrome, Mozilla Firefox, Opera und der Internet Explorer bzw. Edge, gehören, unterstützt. Des Weiteren ist das Implementieren einer Applikation, welche diese Technologien nativ benutzt, bisher sehr komplex und schwierig zu organisieren. Im Zuge dessen, entwickelt Google aktiv an einer Library namens “Polymer”, welche sich diesen Problemen annimmt. Polymer stellt dabei eine Reihe an unterschiedlichen Schichten dar, welche den Umgang mit Web Components vereinfachen sollen. So stellt Polymer eine Sammlung an Mechanismen bereit, welche älteren Browsern die nötigen Features für den Einsatz von Web Components beibringen. Ebenso soll das Erstellen von eigenen HTML-Elementen mit der Polymer-Library und der damit bereitgestellten Application Programming Interface (API) für Entwickler komfortabler gemacht werden. Um bereits entwickelte Web Components einfach wiederverwenden zu können, bietet Polymer eine Sammlung von vorgefertigten Elementen an.

Web Components und die Polymer-Library greifen stark in den Entwicklungsprozess von Webseiten ein und sollen diesen verbessern und vereinfachen. Die Seitenbau GmbH interessiert sich stark für diese neue Technologie, da Wiederverwendbarkeit, Wartbarkeit und neue Technologien im Fokus des Frontend-Engineerings des Unternehmens stehen. Die Seitenbau GmbH ist ein mittelständischer IT-Dienstleister und unterstützt seit 1996 Organisationen aus Privatwirtschaft und öffentlicher Verwaltung bei der Planung, Konzeption und Umsetzung hochwertiger Softwarelösungen für E-Business und

E-Government. Zu den Kernkompetenzen der Seitenbau GmbH zählen dabei vor allem das Frontend Engineering und Content Management, die Konzeption und Entwicklung von Individualsoftware sowie der Aufbau von personalisierten Intranet- und Portallösungen.

Im Rahmen dieser Bachelorarbeit sollen die verschiedenen Technologien unter dem Dachbegriff Web Components sowie deren Funktionsweise sowohl ohne, als auch mit der Polymer-Library, untersucht werden. Zur Veranschaulichung soll eine Web Component mit Hilfe von Polymer implementiert und mit einer ähnlichen Implementierung mit AngularJS verglichen werden. Am Beispiel einer Web Component in Form einer Multi-Navigations-Applikation sollen die Vor- und Nachteile des Einsatzes von Polymer in Hinblick auf Implementierung und Performance dargestellt werden.

In Kapitel 2 werden die Standards der Web Components beschrieben, auf welche die in Kapitel 3 beschriebene Library Polymer aufsetzt. Wie sie dies im Detail umsetzt wird in Kapitel 4 dargestellt. In Kapitel 5 werden zusätzliche Funktionalitäten dieser Library aufgezeigt und in Kapitel 6 werden einige Best Practices im Umgang mit ihr erklärt. Die in den Kapiteln 3 bis 6 gewonnenen Erkenntnisse werden in Kapitel 7 in einer Beispielimplementierung umgesetzt und mit einer ähnlichen Implementierung mit AngularJS verglichen. In Kapitel 8 wird abschließend eine Zukunftsprognose aufgestellt.

2 Web Components nach dem vorläufigen W3C-Standard

In diesem Kapitel wird in Abschnitt 2.1 auf die Problemlösung der Web Components nach den Vorstellungen des W3C eingegangen. In Abschnitt 2.2 wird die erste Technologie vorgestellt, die Custom Elements, Abschnitt 2.3 wird sich den HTML Templates widmen, in Abschnitt 2.4 wird auf den Shadow DOM eingegangen und in Abschnitt 2.5 die letzte Technologie, die HTML Imports, gezeigt. In Abschnitt 2.6 werden die für diese Technologien entwickelten Polyfills erklärt. Abschließend wird in Kapitel 2.7 eine exemplarische Komponente implementiert.

2.1 Problemlösung

In der heutigen Webentwicklung kommt es häufig vor, dass für diverse Probleme oftmals die gleiche, oder eine ähnliche Lösung entwickelt werden muss. Diese unterscheiden sich stets leicht, bringen im Kern aber dennoch meist die selben Features mit sich. Um diese Features auf der Webseite verfügbar zu machen, sind eine Reihe an verschiedenen Technologien notwendig. Zum einen muss das von ihr benötigte HTML-Markup geschrieben werden, zum anderen muss ein JavaScript eingebunden und über eine vorgegebene API konfiguriert werden. Damit die Komponente auch optisch funktioniert, muss ein entsprechendes Stylesheet mit den Style-Definitionen eingebunden werden. Da CSS-Regeln immer global auf das gesamte Dokument angewendet werden, kann es dabei zu ungewollten Auswirkungen auf andere Bestandteile der Webseite kommen. Fasst man diese Punkte zusammen, so wird deutlich, dass es in der Webentwicklung kein Plugin-System gibt um Webseiten schnell und einfach zu erweitern.

Diesem Problem widmen sich die Web Components. Sie sollen der Frontend-Entwicklung ein Plugin-System bereitstellen, welches diese Probleme löst. Eine Komponente steht dabei als eigenes HTML-Element, welches ihre gesamte Funktionalität kapselt und nach außen verbirgt. Konflikte mit anderen Komponenten oder der einbindenden Webseite selbst werden somit vermieden. Dabei ist das Verhalten nach außen für jede Komponente dasselbe, es gibt also für jede Komponente die gleiche Schnittstelle, um sie zu konfigurieren und einzubinden. Dies erleichtert deren Umgang deutlich, da die einzige benötigte Technologie HTML ist. Dadurch können Komponenten verwendet werden wie jedes native HTML-Element. Sie sind verschachtelbar und haben Attribute, über welche sie konfiguriert werden können. Web Components bilden dabei eine Sammlung an Technologien, um jene Eigenschaften zu gewährleisten. In den folgenden Abschnitten werden diese Technologien erklärt und auf ihre Anwendung eingegangen.

2.2 Custom Elements

Webseiten werden mit sogenannten Elementen, oder auch Tags, aufgebaut. Das Set an verfügbaren Elementen wird vom W3C definiert und standardisiert. Somit ist die Auswahl an den verfügbaren Elementen stark begrenzt und nicht von Entwicklern erweiterbar, sodass diese nicht ihre eigenen, von ihrer Applikation benötigten Elemente, definieren können. Betrachtet man in Abbildung 2.1 den Quelltext einer populären Web-Applikation Google Mail, wird schnell deutlich, worin das Problem liegt.

Abbildung 2.1: Screenshot des DOMs der Applikation Google Mail

Sie besteht aus vielen geschachtelten -Elementen und macht folgende strukturelle Probleme deutlich: Die Semantik der Elemente fehlt vollständig und es ist nicht ersichtlich was es darstellt und welche Funkionen es hat, wodurch die Applikation nur sehr schwer wartbar ist.

Dieser Problematik widmen sich die Custom Elements. Sie bieten eine neue API, welche es ermöglicht, eigene, semantisch aussagekräftige HTML-Elemente mit Eigenschaften und Funktionen zu definieren. Wird das obige Beispiel nun also mit Hilfe von Custom Elements umgesetzt, so könnte das zugehörige DOM wie in Listing 2.1 dargestellt aussehen [Bidelman 2013a].

```
1  <hangout-module>
2    <hangout-chat from="Paul, Addy">
3      <hangout-discussion>
4        <hangout-message from="Paul" profile="profile.png"
5            profile="11807591949662637591" datetime="2013-07-17T12:02">
6          <p>Hier werden Web Components eingesetzt.</p>
7        </hangout-message>
8      </hangout-discussion>
9    </hangout-chat>
10   <hangout-chat>...</hangout-chat>
11 </hangout-module>
```

Listing 2.1: Beispiel einer Applikation mit Web Components

Die Spezifikation des W3C ermöglicht nicht nur das Erstellen eigenständiger Elemente, sondern auch das Erstellen von Elementen, welche native Elemente erweitern. Somit können die APIs von nativen HTML-Elementen um eigene Eigenschaften und Funktionen erweitert werden.

2.2.1 Neue Elemente registrieren

Um nun ein eigenes Custom Element zu definieren, muss der Name des Custom Elements laut der W3C-Spezifikation zwingend einen Bindestrich enthalten, wie beispielsweise in `my-element`. Somit ist gewährleistet, dass der Parser des Browsers die Custom Elements von den nativen Elementen unterscheiden kann [Glazkov 2015]. Ein neues Element wird mittels JavaScript mit der Funktion `document.registerElement('my-element');` registriert. Zusätzlich zum Namen des Elements kann optional der Prototyp des Elements angegeben werden. Dieser ist jedoch standardmäßig ein `HTMLElement`, somit also erst wichtig, wenn es darum geht, vorhandene Elemente zu erweitern (siehe Abschnitt 2.2.2). Durch das Registrieren des Elements wird es in die Registry des Browsers geschrieben, welche dazu verwendet wird, die Definitionen der HTML-Elemente aufzulösen [Bidelman 2013a]. Ist ein Element noch nicht definiert und nicht beim Browser registriert, steht aber im Markup der Webseite, wird dies keinen Fehler verursachen, da dieses Element das Interface von `HTMLUnkownElement` benutzen muss [WHATWG web].

Nachdem das Element beim Browser registriert wurde, muss es zunächst mittels der Anweisung `document.createElement(tagName);` erzeugt werden, der `tagName` ist hierbei der Name des zuvor registrierten Elements. Danach kann es imperativ per JavaScript mittels `document.body.appendChild(myelement);` oder deklarativ direkt im HTML-Dokument mittels `<my-element><my-element>` verwendet werden.
[Overson und Strimpel 2015, S. 127-138]

2.2.2 Vorhandene Elemente erweitern (Type Extensions)

Statt neue Elemente zu erzeugen, können sowohl native HTML-Elemente als auch bereits erstellte Custom Elements durch prototypische Vererbung um Funktionen und Eigenschaften erweitert werden, was auch als "Type Extension" bezeichnet wird. Zusätzlich zum Namen des erweiterten Elements wird nun der Prototyp sowie der Name des zu erweiternden Elements der `registerElement`-Funktion als Parameter übergeben. Soll also ein erweitertes `button`-Element registriert werden, muss dies wie in Listing 2.2 gemacht werden.

```
1 var ButtonExtendedProto = document.registerElement('button-extended', {
2   prototype: Object.create(HTMLButtonElement.prototype),
3   extends: 'button'
4 });
```

Listing 2.2: Registrieren eines erweiterten Button-Elements

Anschließend kann es mit dem Namen des zu erweiternden Elements als erstem Parameter und dem Namen des erweiterten Elements als zweitem Parameter erzeugt werden. Alternativ kann es auch mit Hilfe des Konstruktors erzeugt werden (siehe Listing 2.3) [Kitamura 2014a].

```
1  var buttonExtended  = document.createElement('button', 'button-extended
       ');
2
3  // Alternativ
4  var buttonExtended = new ButtonExtendedProto();
```

Listing 2.3: Erzeugen eines erweiterten Button-Elements

Um es nun im DOM zu benutzen, muss der Name des erweiterten Elements via dem Attribut `is="elementName"` des erweiternden Elements angegeben werden. So wird der erweiterte Button deklarativ mittels `<button is="button-extended"></button>` in das Dokument eingebunden.

2.2.3 Eigenschaften und Methoden definieren

Anhand des obigen Beispiels wird deutlich, wie ein Custom Element eingesetzt werden kann, jedoch sind die internen JavaScript-Mechanismen nicht ersichtlich. Custom Elements entfalten ihr vollständiges Potential jedoch erst, wenn man für diese auch eigene Eigenschaften und Methoden definiert. Wie bei nativen HTML-Elementen ist das auch bei Custom Elements auf analoge Weise möglich [Overson und Strimpel 2015, S. 127-138]. So kann einem Element eine Funktion zugewiesen werden, in dem diese dessen Prototyp mittels einem nicht reservierten Namen angegeben wird. Selbiges gilt für eine neue Eigenschaft. Die Eigenschaften können, nachdem sie im Prototyp definiert wurden, im HTML-Markup deklarativ konfiguriert werden (siehe Listing 2.4). Eigene Elemente mit einem spezifischen Eigenverhalten und Aussehen, wie beispielsweise ein neuer Video-Player, sind dadurch mit einem Tag statt mit einem Gerüst aus `<div>`-Tags oder Ähnlichem umsetzbar.

```
1  <script>
2  // Methode definieren
3  ButtonExtendedProto.alert = function () {
4    alert('foo');
5  };
6
7  // Eigenschaft definieren
8  ButtonExtendedProto.answer = 42;
9  </script>
10
11 <!-- Beispiel einer deklarativen Konfiguration -->
12 <button-extended answer="41">Ich bin ein Button</button-extended>
```

Listing 2.4: Eigenschaften und Methoden definieren und konfigurieren

2.2.4 Custom Element Lifecycle-Callbacks

Custom Elements bieten eine standardisierte API an speziellen Methoden, den "Custom Element Lifecycle-Callbacks", welche es ermöglichen Funktionen zu unterschiedlichen Zeitpunkten - vom Registrieren bis zum Löschen eines Custom Elements - auszuführen. Diese ermöglichen es, zu bestimmen, wann und wie ein bestimmter Code des Custom Elements ausgeführt werden soll.

createdCallback Wird ausgeführt, wenn eine Instanz des Custom Elements mittels `var mybutton = document.createElement('custom-element')` erzeugt wurde.

attachedCallback Wird ausgeführt, wenn ein Custom Element dem DOM mittels der Funktion `document.body.appendChild(mybutton)` angehängt wurde.

detachedCallback Wird ausgeführt, wenn ein Custom Element aus dem DOM mittels der Funktion `document.body.removeChild(mybutton)` entfernt wurde.

attributeChangedCallback Wird ausgeführt, wenn ein Attribut eines Custom Elements mittels `MyElement.setAttribute()` geändert wurde.

So können die Lifecycle-Callbacks für ein neues erweitertes Button-Element, wie in Listing 2.5 dargestellt, definiert werden [Schaffranek 2014].

```
1 var ButtonExtendedProto = Object.create(HTMLElement.prototype);
2
3 ButtonExtendedProto.createdCallback = function() {...};
4 ButtonExtendedProto.attachedCallback = function() {...};
5
6 var ButtonExtended = document.registerElement('button-extended', {
7   prototype: ButtonExtendedProto
8 });
```

Listing 2.5: Lifecycle-Callbacks des erweiterten Buttons definieren

2.2.5 Styling von Custom Elements

Das Styling von eigenen Custom Elements funktioniert analog dem Styling von nativen HTML-Elementen indem der Name des Elements als CSS-Selektor angegeben wird (siehe Listing 2.6). Erweiterte Elemente können mittels dem Attribut-Selektor in CSS angesprochen werden [Overson und Strimpel 2015, S. 127-138].

Ein Custom Element, welches zwar standardkonform deklariert oder erstellt, aber noch nicht beim Browser registriert wurde, ist ein "Unresolved Element". Steht dieses Element am Anfang des DOM, wird jedoch erst später registriert, kann es nicht von CSS angesprochen werden. Dadurch kann ein Flash Of Unstyled Content (FOUC) entstehen, was bedeutet, dass das Element beim Laden der Seite nicht gestylt dargestellt wird, sondern das definierte Aussehen erst übernimmt, nachdem es registriert wurde. Um dies zu verhindern, sieht die HTML-Spezifikation eine neue CSS-Pseudoklasse `:unresolved`

(siehe Listing 2.6) vor, welche deklarierte, aber nicht registrierte Elemente anspricht. Somit können diese Elemente initial beim Laden der Seite ausgeblendet und nach dem Registrieren wieder eingeblendet werden [Gasston 2014].

```
1  /* Eigenes Custom Element */
2  my-element {
3    color: black;
4  }
5
6  /* Erweitertes natives HTML-Element*/
7  [is="button-extended"] {
8    color: black;
9  }
10
11 /* Unregistrierte Custom Elements */
12 :unresolved {
13   display: none;
14 }
```

Listing 2.6: Styling eines Custom Element und des erweiterten Button

2.2.6 Browserunterstützung

Custom Elements sind noch nicht vom W3C standardisiert, sondern befinden sich noch im Status eines "Working Draft" [Glazkov 2015]. Sie werden deshalb bisher nur von Google Chrome ab Version 43 und Opera ab Version 33 nativ unterstützt (siehe Abbildung 2.2) [Can I Use - Custom Elements web].

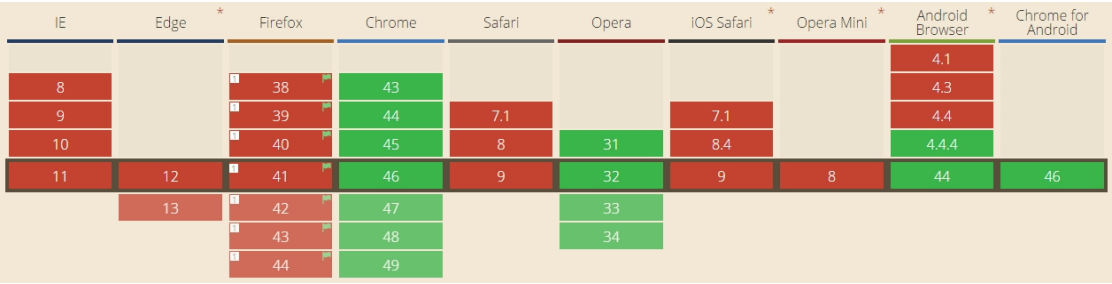

Abbildung 2.2: Browserunterstützung von Custom Elements

2.3 HTML Templates

Bisher gibt es ohne eine Library oder Framework keine Möglichkeit, im Browser Templates zu rendern, um bestimmte Inhalte der Seite zur Laufzeit auszuwechseln. Die Technologie "HTML Templates" ist eine neue Technologie im Rahmen der Web Components und versucht eben dieses Problem mit einer nativen API zu lösen.

Im Kontext der Entwicklung einer Model View Controller (MVC)-Applikation ist der Mechanismus der Darstellung des Views besonders wichtig. Bisher ist dies ohne weiteres problemlos serverseitig in PHP, Ruby oder ähnlichem möglich, da diese Sprachen für die

Webentwicklung eine Syntax für die Einbettung dynamischer Inhalte in HTML bieten, die sogenannten Templates. Im Gegensatz zu den serverseitigen Technologien existieren Client-seitige Lösungen bisher nur als Library oder Framework, wie beispielsweise Mustache.js oder Handlebars.js [Namics 2014]. Eine native Möglichkeit, Templates auf der Client-Seite zu benutzen, fehlt bisher hingegen. An diese Problematik setzen die HTML Templates an, durch welche diese Technik auch Einzug in den Browser erhält. [Overson und Strimpel 2015, S. 101-107]

> Das HTML template-Element <template> dient dazu, Client-seitige Inhalte zu gruppieren, die nicht gerendert werden, wenn die Seite geladen wird, sondern anschließend zur Laufzeit mittels JavaScript gerendert werden können. Template kann als Inhaltsfragment aufgefasst werden, das für eine spätere Verwendung im Dokument gespeichert wird. [MDN 2015b]

2.3.1 Bisherige Umsetzung von Templates im Browser

Dennoch gibt es diverse Methoden, diese Technologie im Browser zu simulieren. Diese sind jedoch eher als Hacks zu betrachten, da ihre eingesetzten Mittel nicht für dieses Problem gedacht sind. Sie bringen also einige Nachteile mit sich. Einige dieser Methoden werden nachfolgend aufgezeigt [Kitamura 2014b].

Via versteckten `<div>`-Element

Das Listing 2.7 zeigt die Umsetzung eines Templates mit Hilfe eines `<div>`-Blocks, der via CSS versteckt wird.

```
1 <div id="mydivtemplate" style="display: none;">
2   <div>
3     <img src="myimage.jpg">
4   </div>
5 </div>
```

Listing 2.7: Umsetzung eines Templates mit einem `<div>`-Block

Der Nachteil dieser Methode ist, dass alle enthaltenen Ressourcen, beim Laden der Webseite heruntergeladen werden. Zwar werden sie nicht angezeigt, dennoch verursachen sie eine große Datenmenge, welche initial übertragen werden muss. Dies geschieht selbst wenn die Ressourcen eventuell erst später oder gar nicht benötigt werden. Des Weiteren kann es sich als schwierig erweisen, ein solches Code-Fragment zu stylen oder gar Themes auf mehrere solcher Fragmente anzuwenden. Eine Webseite, die das Template verwendet, muss alle CSS-Regeln für das Template mit `#mydivtemplate` erstellen, welche sich unter Umständen auf andere Teile der Webseite auswirken können. Eine Kapselung des Inhalts leistet diese Methode nicht.

Via <script>-Element

Eine weitere Möglichkeit ein Template umzusetzen besteht darin, den Inhalt eines Templates in ein <script>-Tag zu schreiben, wie in Listing 2.8 gezeigt.

```
1 <script type="text/template">
2   <div>
3     <img src="myimage.jpg">
4   </div>
5 </script>
```

Listing 2.8: Umsetzung eines Templates mit einem <script>-Element

Wie bei dem Beispiel mit einem -Element wird auch bei dieser Methode der Inhalt nicht gerendert, da für ein <script>-Tag standardmäßig die CSS-Eigenschaft `display: none` gesetzt ist. In diesem Fall werden jedoch die benötigten Ressourcen nicht geladen, somit gibt es keine zusätzlichen Performance-Einbrüche. Es besteht dennoch der Nachteil, dass der Inhalt des <script>-Tags via `innerHTML` in den DOM geklont werden muss, was eine mögliche Cross Site Scripting (XSS) Sicherheitslücke darstellt. Es muss also abgewägt werden, welche der Nachteile für den Entwickler am ehesten hinnehmbar sind und welche Methode verwendet werden soll.

2.3.2 Das <template>-Tag

Den Problemen der oben genannten Methoden widmet sich das <template>-Tag, welches eine native und sichere Methode für das Einbinden von dynamischen Inhalten etabliert. Das Template und die darin enthaltenen Inhalte bilden ein HTML-Gerüst, welches beim Rendern der Webseite vollständig ignoriert wird. Es wird weder angezeigt, noch werden ihre benötigten Inhalte beim Laden der Webseite übertragen. Dadurch können beliebig viele <template>-Tags ohne signifikanten Performance-Einbruch im Quelltext stehen, da nur ihr Markup übertragen wird, es jedoch nicht vom Browser geparst werden muss, was erst beim Einfügen des Templates in das DOM mittels JavaScript geschieht. Die Templates können dabei beliebig oft und an beliebiger Stelle in den DOM der Webseite eingefügt werden. Des weiteren werden in Templates enthaltene JavaScripts nicht ausgeführt, auch kann JavaScript von außen nicht in das Template hinein traversieren. In Listing 2.9 wird die grobe Struktur eines einfachen Templates dargestellt.

```
1 <template id="mytemplate">
2   <style>/* Styles */</style>
3   <script>// JavaScript</script>
4   <img src="bild.jpg"> <!-- Kann dynamisch gesetzt werden -->
5 </template>
```

Listing 2.9: Grobe Struktur eines Templates mit CSS und JavaScript

2.3.3 Benutzung

Natürlich soll ein Template nicht nur im Quelltext stehen, damit es existiert, sondern es soll dynamisch zur Laufzeit geladen und gerendert werden. Dabei kann es an einer beliebigen Stelle im Quelltext stehen. Um es aus dem Quelltext in den DOM zu importieren und zu rendern, muss es zunächst via JavaScript selektiert werden, was mit der Funktion `var template = document.querySelector('#mytemplate');` möglich ist. Mit der Funktion `var templateClone = document.importNode(template.content, true);` wird eine Kopie als DOM-Knoten des Templates erstellt. Als erster Parameter wird dabei der Inhalt des Templates (`template.content`), als zweiter Parameter ein Boolean für `deep`, welcher bestimmt ob auch Kinderknoten geklont werden sollen, angegeben. Nun kann der Inhalt des Templates mittels `document.body.appendChild(templateClone);` an einer beliebigen Stelle des DOM eingefügt werden.

2.3.4 Browserunterstützung

HTML Templates sind zum Stand dieser Arbeit als einzige Technologie des Web Components Technology Stacks vom W3C als Standard erklärt worden [W3C 2014b]. Somit ist auch die Browserunterstützung in den aktuellen Browsern, bis auf den Internet Explorer, sehr gut (siehe Abbildung 2.3). Sie sind des Weiteren die einzige Technologie der Web Components, die bisher von Microsofts Edge ab Version 13 unterstützt werden.

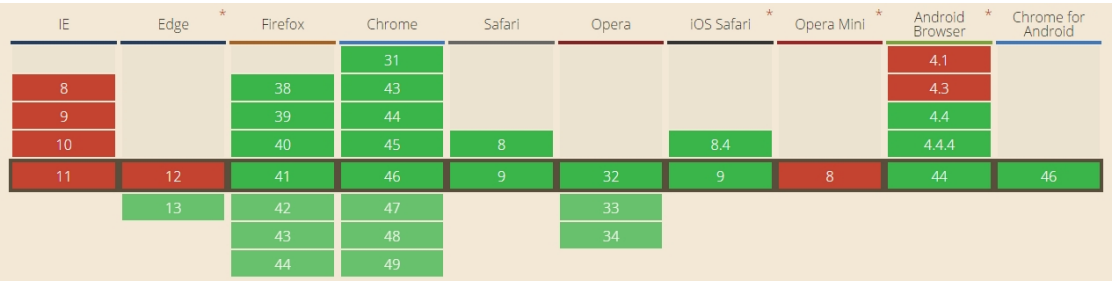

Abbildung 2.3: Browserunterstützung des HTML Template Tags

2.4 Shadow DOM

Durch Kapselung ist es möglich, Details eines Objektes von anderen Teilen des Programms zu verstecken. Das Programm muss nur wissen, wie es auf die benötigten Funktionen zugreift, jedoch nicht, wie das Objekt die Funktionen intern umsetzt. Dieses Konzept ist in allen objektorientierten Programmiersprachen umgesetzt, jedoch nicht in der Webentwicklung. Beispielsweise kann das CSS oder JavaScript, das für ein Element geschrieben ist, auch das CSS oder JavaScript anderer Elemente beeinflussen. Je größer das Projekt wird, desto unübersichtlicher und komplexer wird es, zu gewährleisten, dass CSS oder JavaScript sich nicht ungewollt auf andere Teile der Webseite auswirkt.

Diesem Problem widmet sich das sogenannte Shadow DOM, welches ein Sub-DOM unterhalb eines Elements darstellt und es ermöglicht, HTML und CSS in sich zu kapseln und zu verstecken. Als Kontrast zu Bezeichnung "Shadow DOM" wird das reguläre DOM des Hauptdokuments auch oft als "Light DOM" bezeichnet. Das Shadow DOM wird bereits in HTML5 standardmäßig eingesetzt, wie beispielsweise im <input>-Tag. Beim Inspizieren des Elements mit Hilfe der Chrome Developer Tools (siehe Abbildung 2.4) wird deutlich, dass das <input>-Tag ein Shadow DOM beinhaltet, welches das eingegebene Passwort kapselt. [Overson und Strimpel 2015, S. 109-126]

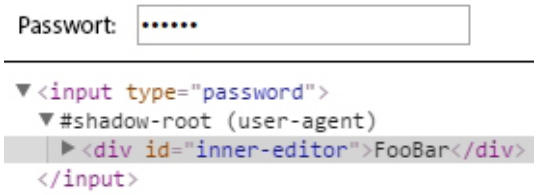

Abbildung 2.4: Passwort-Input-Element

2.4.1 Shadow DOM nach W3C

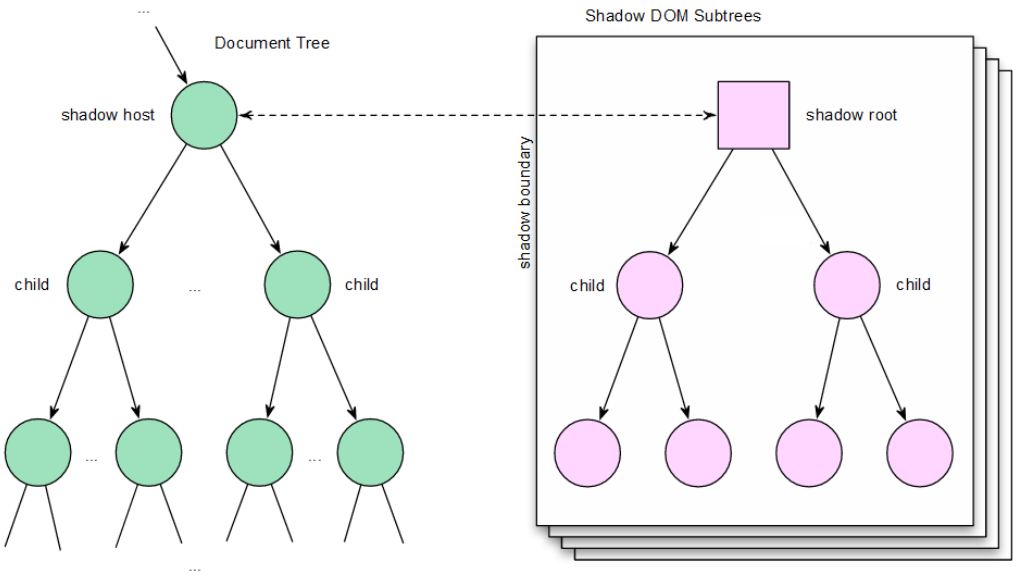

Abbildung 2.5: Shadow DOM und Shadow Boundary nach W3C

Wie auf Abbildung 2.5 zu sehen, liegt das Shadow DOM dabei parallel zu dem DOM-Knoten des beinhaltenden Elements. Ein Knoten im Document Tree (links abgebildet) wird als "Shadow Host" - ein Element, welches ein Shadow DOM beinhaltet - markiert. Die gestrichelte Linie zeigt die Referenz zu der entsprechenden Shadow DOM Wurzel, dem "Shadow Root". Die Referenz geht dabei durch die sogenannte "Shadow Boundary", welche es ermöglicht, das Shadow DOM, und alles was dieses beinhaltet,

zu kapseln [Ihrig 2012]. Dies verhindert, dass externes CSS oder JavaScript das interne Markup oder umgekehrt internes CSS oder JavaScript das Light DOM oder andere Shadow DOMs ungewollt beeinflussen können. Ein Element kann auch mehrere Shadow DOM Wurzeln referenzieren, allerdings wird nur die zuletzt hinzugefügte vom Browser gerendert, da dieser zum Rendern einen Last In First Out (LIFO) Stack benutzt. Dabei wird der zuletzt hinzugefügte Shadow Tree "Youngest Tree" genannt, der jeweils zuvor hinzugefügte Shadow Tree wird "Older Tree" genannt. Das dynamische Hinzufügen von Shadow DOMs ermöglicht es, die Inhalte der Webseite nach dem Rendern zu ändern.

2.4.2 Content Projection

Neben dem vom Shadow DOM vorgegebenen HTML, können auch Inhalte aus dem Light DOM in den Shadow DOM projiziert werden. Der Shadow DOM nimmt dabei die zu projizierenden Inhalte und projiziert sie an der vorgegebenen Stelle im Shadow DOM. Die Inhalte bleiben dabei an der ursprünglichen Stelle im DOM stehen und werden nicht verschoben, gelöscht oder geändert. Der Shadow DOM ermöglicht es somit eigenes, gekapseltes HTML sowie dynamische Inhalte des Light DOM anzuzeigen. Diese Projektion der Inhalte aus dem Light DOM in den Shadow DOM erfolgt mittels sogenannten "Insertion Points". Diese sind vom Entwickler definierte Stellen im Shadow DOM, in welche der Inhalt projiziert wird. Es kann hierbei zwischen 2 Arten von Insertion Points unterschieden werden.

2.4.3 Insertion Points

Um das zu präsentierende HTML und den Inhalt zu trennen, wird ein <template>-Tag benutzt. Dieses beinhaltet das komplette Markup, das im Shadow DOM stehen und nicht nach außen sichtbar sein oder von CSS oder JavaScript von außen manipuliert werden soll. Um nun Inhalte aus dem Light DOM in das DOM des <template>-Tags zu projizieren, muss das <template>-Tag einen <content>-Tag beinhalten, in welchem die Inhalte von außen dargestellt werden sollen. Mittels `createShadowRoot()` wird das ausgewählte Element zu einem Shadow Host, also dem Shadow DOM beinhaltendem Element gemacht. Der Inhalt des Templates wird geklont und dem Shadow Host angehängt. Das Shadow DOM projiziert nun alle Inhalte des Shadow Roots in den <content>-Tag (siehe Listing 2.10) [Cooney und Bidelman 2013].

Im Light DOM gerendert wird dabei nur der Text "Inhalt" des -Tags mit der ID `shadow`, der Wrapper um das <content>-Tag wird nicht gerendert, da dieser im Shadow DOM steht. Somit wurde eine Trennung des präsentierenden HTML und dem Inhalt erreicht, die Präsentation erfolgt im Shadow DOM, der Inhalt steht im Light DOM. Werden nun mehrere HTML-Elemente oder Knoten in den Shadow DOM projiziert, werden diese "Distributed Nodes" genannt. Diese Distributed Nodes sind nicht wirklich im Shadow DOM, sondern werden nur in diesem gerendert, was bedeutet, dass sie auch von

```
1  <div id="shadow">Content</div>
2  <template id="myTemplate">
3    <div class"hiddenWrapper">
4      <content></content>
5    </div>
6  </template>
7
8  <script type="text/javascript">
9    var shadow = document.querySelector('#shadow').createShadowRoot();
10   var template = document.querySelector('#myTemplate');
11   var clone = document.importNode(template.content, true);
12   shadow.appendChild(clone);
13 </script>
```

Listing 2.10: Template-Definition und Erstellen eines Shadow DOMs

außen gestylt werden können, mehr dazu im Abschnitt 2.4.5. Des Weiteren können auch nur bestimmte Elemente in das Shadow DOM projiziert werden, ermöglicht wird dies mit dem Attribut `select="selector"` des `<content>`-Tags. Dabei können sowohl Namen von Elementen, als auch CSS-Selektoren verwendet werden [Bidelman 2013b]. Der Inhalt des `<content>`-Tags kann mit JavaScript nicht traversiert werden, beispielsweise gibt `console.log(shadow.querySelector('content'));` den Wert `null` aus. Allerdings ist es erlaubt, die Distributed Nodes mittels `.getDistributedNodes()` auszugeben. Dies lässt darauf schließen, dass der Shadow DOM nicht als Sicherheits-Feature angedacht ist, da die Inhalte nicht komplett isoliert sind.

2.4.4 Shadow Insertion Points

Neben den `<content>`-Tags gibt es auch die `<shadow>`-Tags, welche Shadow Insertion Points genannt werden. Shadow Insertion Points sind ebenso wie Insertion Points Platzhalter, doch statt einem Platzhalter für den Inhalt eines Hosts, sind sie Platzhalter für Shadow DOMs. Falls jedoch mehrere Shadow Insertion Points in einem Shadow DOM sind, wird nur der Erste berücksichtigt, die Restlichen werden ignoriert. Wenn nun mehrere Shadow DOMs projiziert werden sollen, muss im zuletzt hinzugefügten Shadow DOM - dem “Younger Tree” - ein `<shadow>`-Tag stehen, dieser rendert den zuvor hinzugefügten Shadow DOM - den “Older Tree”. Somit wird eine Schachtelung mehrerer Shadow DOMs ermöglicht [Bidelman 2014].

2.4.5 Styling mit CSS

Eines der Hauptfeatures des Shadow DOMs ist die Shadow Boundary, welche Kapselung von Stylesheets standardmäßig mit sich bringt. Sie gewährleistet, dass Style-Regeln des Light DOM nicht den Shadow DOM beeinflussen und umgekehrt [Dodson 2014]. Dies gilt jedoch nur für die Präsentation des Inhalts, nicht für den Inhalt selbst. Nachfolgend wird auf die wichtigsten Selektoren für das Styling eingegangen.

:host Das Host-Element des Shadow DOMs kann mittels dem Pseudoselektor `:host` angesprochen werden. Dabei kann dem Selektor optional auch ein Selektor mit übergeben werden wie beispielsweise mit `:host(.myHostElement)`. Mit diesem Selektor ist es möglich, nur Hosts, welche diese Klasse haben, anzusprechen. Zu beachten ist, dass das Host-Element von außen gestylt werden kann, also die Regeln des `:host`-Selektors überschreiben kann. Des Weiteren funktioniert der `:host`-Selektor nur im Kontext eines Shadow DOMs. Man kann ihn also nicht außerhalb benutzen. Besonders wichtig ist dieser Selektor, wenn auf die Aktivität der Benutzer reagiert werden muss. So kann innerhalb des Shadow DOMs angegeben werden, wie das Host Element beispielsweise beim Hover mit der Maus auszusehen hat.

:host-context() Je nach Kontext eines Elements kann es vorkommen, dass Elemente unterschiedlich dargestellt werden müssen. Das wohl am häufigsten auftretende Beispiel hierfür ist das Theming. Themes ermöglichen es, Webseiteninhalte auf unterschiedliche Arten darzustellen. Oftmals bietet eine Webseite oder eine Applikation mehrere Themes für die Benutzer an, zwischen welchen sie wechseln können. Mit dem `:host-context()`-Selektor wird es möglich, ein Host-Element je nach Klasse des übergeordneten Elements - dem Parent-Element - zu definieren. Hat ein Host-Element mehrere Style-Definitionen, so werden diese nach der Klasse des Parent-Elements getriggert. Soll eine Style-Definition beispielsweise nur angewendet werden, wenn das umschließende Element die Klasse `theme-1` hat, so kann das mit `:host-context(.theme-1)` erreicht werden.

2.4.6 Styling des Shadow DOM von außerhalb

Trotz der Kapselung von Shadow DOMs ist es mit speziellen Selektoren möglich, die Shadow Boundary zu durchbrechen. So können Style-Regeln für Shadow Hosts oder in ihm enthaltene Elemente vom Elterndokument definiert werden [W3C 2015b].

::shadow Falls ein Element nun einen Shadow Tree beinhalten sollte, so kann dieses mit der entsprechenden Klasse oder ID sowie dem `::shadow`-Selektor angesprochen werden. Jedoch können mit diesem Selektor keine allgemeingültigen Regeln erstellt werden. Stattdessen muss stets ein in dem Shadow Tree enthaltener Elementname angesprochen werden. Nimmt man sich nun die Content Insertion Points zur Hilfe, so ist es dennoch möglich, Regeln für mehrere unterschiedliche Elemente zu definieren. Die Selektor-Kombination `#my-element::shadow content` Spricht nun alle <`content`>-Tags an, welche im Shadow Tree des Elements `#my-element` vorhanden sind. Da in diesen <`content`>-Tag wiederum die Inhalte hineinprojiziert werden, werden die Regeln für die projizierten Elemente angewendet. Sollten nun noch weitere Shadow DOMs in diesem Element geschachtelt sein, so werden die Regeln jedoch nicht für diese angewandt, sondern nur für das direkt folgende Kind des Elements `#my-element`.

`>>>` Dieser Kombinator ist ähnlich dem `::shadow`-Selektor. Er bricht jedoch durch sämtliche Shadow Boundaries und wendet die Regeln auf alle gefundenen Elemente an. Dies gilt - im Gegensatz zum `::shadow`-Selektor - auch für geschachtelte Shadow Trees. Mit dem Selektor `my-element >>> span` werden also alle im Element <`my-element`> sowie in dessen geschachtelten Shadow Trees enthaltenen <`span`>-Elemente angesprochen.

::slotted Die Kombination `#my-element::shadow content` zeigt die Möglichkeit, wie die Inhalte eine Shadow DOMs von außen gestylt werden können. Sollen jene in den Content Insertion Points enthaltenen Elemente jedoch von innerhalb gestylt werden, so ist dies mit dem `::slotted`-Selektor möglich. Ist innerhalb eines Elements die Regel `::slotted p` definiert, so werden alle in den Shadow DOM projizierten <`p`>-Tags angesprochen.

2.4.7 CSS-Variablen

Die oben gezeigten Selektoren und Kombinatoren eignen sich hervorragend um die Shadow Boundary zu durchdringen und eigene Styles den Elementen aufzuzwingen. Jedoch sprengen sie das Prinzip der Kapselung, das man mit Web Components zu gewinnen versucht. Dennoch haben sie ihre Existenzberechtigung. Sie ermöglichen es den Entwicklern, fremde Components sowie native HTML-Elemente, die einen Shadow DOM benutzen, wie z.B. <`video`> oder <`input`>-Elemente, zu stylen. Allerdings sollte bei ihrer Anwendung äußerst vorsichtig gearbeitet werden, insbesondere da sie schnell missbraucht werden können.

Ein Ansatz zur Lösung dieser Problematik sind CSS-Variablen. Mit ihnen können Elemente in ihrem Shadow DOM Variablen für die inneren Styles bereit halten, welche von außerhalb des Shadow DOMs instanziiert werden können. Dadurch ist es möglich, das innere Styling eines Elements nach außen zu reichen. Statt die Barriere nun zu durchbrechen, können Elemente miteinander kommunizieren. Es wird somit eine Style-Schnittstelle geschaffen [W3C 2015a]. Ein Element `my-element` kann nun in dessen Shadow DOM beispielsweise die Schriftfarbe mit `color` definieren. Doch statt einem Wert wird nun der Name der nach außen sichtbaren Variablen angegeben. Die Syntax hierfür lautet `var(--varable-name, red)`, wobei `red` hier der Standardwert ist, falls die Variable `--variable-name` von außen nicht gesetzt wird. Nachdem die Variable definiert wurde, kann sie von außerhalb des Shadow DOM mittels der Regel `#my-shadow-host { --varable-name: blue; }` überschrieben werden.

2.4.8 Beispiel eines Shadow DOMs mit Template und CSS

Listing 2.11 zeigt eine exemplarische Implementierung eines Shadow DOMs, welcher gekapseltes CSS und JavaScript beinhaltet.

```
1  <style>
2    .styled { color: green; }
3    .content { background-color: khaki; }
4  </style>
5
6  <div id="hello">
7    <div>Hallo</div>
8    <div class="styled">Gestyled</div>
9    <p class="hidden">Inhalt, der vom Shadow DOM ueberschrieben wird.</p>
10 </div>
11
12 <template id="myTemplate">
13   <style>
14     .wrapper {
15       border: 2px solid black;
16       width: 120px;
17     }
18     .content { color: red; }
19   </style>
20
21   <div class="wrapper">
22     Shadow DOM
23     <div class="content">
24       <content select="div"></content>
25     </div>
26   </div>
27 </template>
28
29 <script>
30   var root = document.querySelector('#hello').createShadowRoot();
31   var template = document.querySelector('#myTemplate');
32   var clone = document.importNode(template.content, true);
33   root.appendChild(clone);
34 </script>
```

Listing 2.11: Beispielimplementierung eines Shadow DOMs mit Template und CSS

Das obige Beispiel wird vom Browser, wie in Abbildung 2.6 dargestellt, gerendert.

Abbildung 2.6: Shadow DOM Beispiel

Anhand des gerenderten Outputs werden einige Dinge deutlich. Mit der Angabe des `select`-Attributs, werden im `<content>`-Tag nur die `<div>`-Tags mit der ID `hello` aus dem Shadow Root gerendert. Der Shadow Root wird dabei mittels der Anweisung `var root = document.querySelector('#hello').createShadowRoot()` erzeugt. Der Paragraph mit der ID `hidden` wird hingegen nicht gerendert, da er nicht im `select` mit

inbegriffen ist. Die CSS-Regel `.content { background-color: khaki; }` des Eltern-HTML-Dokuments greift nicht, da die Styles des Shadow Roots durch die Shadow Boundary gekapselt werden. Die CSS Regel `.styled { color: green; }` greift allerdings, da das -Element mit der Klasse `styled` aus dem Light DOM in den Shadow DOM projiziert wird. Außerdem können innerhalb des Templates CSS-Regeln für die beinhaltenden Elemente definiert werden, somit wird das -Element ohne eine zugehörige Klasse mit der Regel `.content { color: red; }` auch dementsprechend in Rot gerendert.

2.4.9 Browserunterstützung

Der Shadow DOM ist noch nicht vom W3C standardisiert, sondern befindet sich noch im Status eines "Working Draft" [W3C 2015c]. Er wird deshalb bisher nur von Google Chrome ab Version 43 und Opera ab Version 33 nativ unterstützt (siehe Abbildung 2.7) [Can I Use - Shadow DOM web].

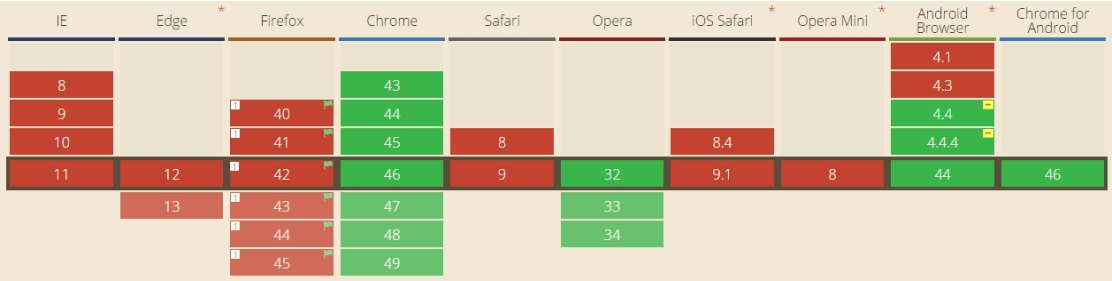

Abbildung 2.7: Browserunterstützung des Shadow DOMs

2.5 HTML Imports

Bisher erlauben es praktisch alle Plattformen, Codeteile zu importieren und zu verwenden, nur nicht das Web bzw. HTML. Das heutige HTML ermöglicht es externe Stylesheets, JavaScript Dateien, Bilder etc. in ein HTML Dokument zu importieren, HTML-Dateien selbst können jedoch nicht importiert werden. Auch ist es nicht möglich, alle benötigten Dateien in einer Ressource zu bündeln und als einzige Abhängigkeit zu importieren. HTML Imports versuchen eben dieses Problem zu lösen. So soll es möglich sein, HTML-Dateien und wiederum HTML-Dateien in HTML-Dateien zu importieren. So können auch verschiedene benötigte Dateien in einer HTML-Datei gesammelt und mit nur einem Import in die Seite eingebunden werden. Doppelte Abhängigkeiten sollen dadurch automatisch aufgelöst werden, sodass Dateien, die mehrmals eingebunden werden sollten, automatisch effektiv nur einmal heruntergeladen werden.

2.5.1 HTML-Dateien importieren

Imports von HTML-Dateien werden, wie andere Imports auch, per <link>-Tag deklariert. Neu ist jedoch der Wert des `rel`-Attributes, welches auf `import` gesetzt wird [Overson und Strimpel 2015, S. 139-147]. So kann beispielsweise das Bootstrap-Framework statt wie bisher mit mehreren Imports mit nur einem Import eingebunden werden. Bisher könnte die Einbindung unter Berücksichtigung der Abhängigkeiten wie in Listing 2.12 aussehen.

```
1 <link rel="stylesheet" href="bootstrap.css">
2 <link rel="stylesheet" href="fonts.css">
3 <script src="jquery.js"></script>
4 <script src="bootstrap.js"></script>
5 <script src="bootstrap-tooltip.js"></script>
6 <script src="bootstrap-dropdown.js"></script>
```

Listing 2.12: Einbindung von Bootstrap ohne HTML Imports

Stattdessen kann dieses Markup nun in ein einziges HTML Dokument, welches alle Abhängigkeiten verwaltet, geschrieben werden. Dieses wird dann mit einem einzigen Import `<link rel="import" href="bootstrap.html">` in das eigene HTML Dokument importiert.

Es ist jedoch zu beachten, dass HTML Imports nur auf Ressourcen der gleichen Quelle, also dem gleichen Host, respektive der gleichen Domain zugreifen können. Imports von HTML-Dateien von verschiedenen Quellen stellen eine Sicherheitslücke dar, da Webbrowser die Same Origin Policy (SOP) verfolgen.

> The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin. It is a critical security mechanism for isolating potentially malicious documents. [MDN 2015a]

Sollte das jedoch dennoch erlaubt werden, so muss das Cross Origin Resource Sharing (CORS) für die entsprechende Domain auf dem Server aktiviert werden.

> These restrictions prevent a client-side Web application running from one origin from obtaining data retrieved from another origin, and also limit unsafe Hypertext Transfer Protocol (HTTP) requests that can be automatically launched toward destinations that differ from the running application's origin. [W3C 2014a]

2.5.2 Auf importierte Inhalte zugreifen

Importierte HTML-Dateien werden nicht nur in das Dokument eingefügt, sondern vom Parser verarbeitet, das bedeutet, dass mit JavaScript auf das DOM des Imports zugegriffen werden kann. Wenn sie vom Parser verarbeitet worden sind, sind sie zwar verfügbar, allerdings werden die Inhalte nicht angezeigt bis sie mittels JavaScript in das DOM

eingefügt werden. Ihre enthaltenen Scripte werden also ausgeführt, Styles und HTML-Knoten werden jedoch ignoriert. Um nun die Inhalte eines Imports auf der Seite einzubinden, muss auf die `.import` Eigenschaft des <link>-Tags mit dem Import zugegriffen werden `var content = document.querySelector('link[rel="import"]').import;`. Nun kann auf das DOM des Imports zugegriffen werden. So kann ein enthaltenes Element mit der Klasse `element` mit `var el = content.querySelector('.element');` geklont und anschließend durch `document.body.appendChild(el.cloneNode(true));` in das eigene DOM eingefügt werden. Falls es jedoch mehrere Imports geben sollte, können den Imports IDs zugewiesen werden, anhand derer die Imports voneinander unterschieden werden können (siehe Listing 2.13) [Hongkiat 2014].

```
1  <head>
2    <link rel="import" href="my-import.html" id="import1">
3    <link rel="import" href="another-import.html" id="import2">
4  </head>
5  <body>
6    <script>
7      var content = document.querySelector('#import1').import;
8      var el = content.querySelector('.element');
9      document.body.appendChild(el.cloneNode(true));
10   </script>
11 </body>
```

Listing 2.13: Einbinden und Verwenden von HTML Imports

2.5.3 Abhängigkeiten verwalten

Jedoch kann es vorkommen, dass mehrere eingebundenen Bibliotheken die gleichen Abhängigkeiten haben und diese verwaltet werden müssen. Sollte dies nicht sorgfältig gemacht werden, können unerwartete Fehler auftreten, die mitunter nur schwer zu lokalisieren sind. HTML Imports verwalten Abhängigkeiten automatisch. Um das zu erreichen, werden die Abhängigkeiten gebündelt und in eine zu importierende HTML-Datei geschrieben. Haben mehrere Imports die gleichen Abhängigkeiten (siehe Listing 2.14), werden diese vom Browser erkannt und nur einmal heruntergeladen. Mehrfachdownloads und Konflikte der Abhängigkeiten werden so verhindert [Kitamura 2015].

```
1  <!-- index.html -->
2  <link rel="import" href="component1.html">
3  <link rel="import" href="component2.html">
4
5  <!-- component1.html -->
6  <script src="jQuery.html"></script>
7
8  <!-- component2.html -->
9  <script src="jQuery.html"></script>
10
11 <!-- jQuery.html -->
12 <script src="js/jquery.js"></script>
```

Listing 2.14: Abhängigkeitsverwaltung mit HTML Imports

Selbst wenn die jQuery-Library in mehreren Dateien eingebunden wird, wird hier sie dennoch nur einmal übertragen. Wenn nun fremde HTML-Dateien eingebunden werden, muss auch nicht auf die Reihenfolge der Imports geachtet werden, da diese selbst ihre Abhängigkeiten beinhalten.

2.5.4 Sub-Imports

Das obige Beispiel zeigt, dass HTML Imports selbst wiederum auch HTML Imports beinhalten können. Diese Imports werden Sub-Imports genannt und ermöglichen einen einfachen Austausch oder eine Erweiterungen von Abhängigkeiten innerhalb einer Komponente. Wenn eine Komponente A eine Abhängigkeit von einer Komponenten B hat und eine neue Version von Komponente B verfügbar ist, kann dies einfach in dem Import des Sub-Imports angepasst werden ohne den Import in der Eltern-HTML-Datei anpassen zu müssen [HTML5Rocks 2013].

2.5.5 Performance

Ein signifikantes Problem der HTML Imports ist die Performance, welche bei komplexen Anwendungen schnell abfallen kann. Werden auf einer Seite mehrere Imports eingebunden, die selbst wiederum Imports beinhalten können, so steigt die Anzahl an Requests an den Server schnell exponentiell an, was die Webseite stark verlangsamen kann. Auch das Vorkommen von doppelten Abhängigkeiten kann die Anzahl an Requests in die Höhe treiben, doch da Browser die Abhängigkeiten nativ schon selbst de-duplizieren, kann dieses Problem in diesem Zusammenhang ignoriert werden [Perterkroener 2014].

Dennoch sind viele Einzel-Requests ein gewünschtes Verhalten von Web Components, da sie bereits mit Blick auf Version 2 des HTTP-Protokolls entwickelt wurden. In diesem können mehrere Requests in einer Transmission Control Protocol (TCP)-Verbindung übertragen werden. Des Weiteren kann der Client die Priorität der angeforderten Dateien bestimmen, um Dateien mit einer hohen Priorität vor Dateien mit niedrigerer Priorität zu erhalten. Ebenso wird mit einer neuen Header-Kompression die zu übertragende Datenmenge verkleinert. Neu sind außerdem die sogenannten “Server Pushes”, welche es dem Server ermöglichen, Nachrichten an den Client zu schicken, ohne dass dieser sie anfordern muss.

Voraussetzung hierfür ist jedoch die Unterstützung des neuen Protokolls seitens des Browsers. Bis auf den Internet Explorer und Safari unterstützen jedoch alle Browser HTTP/2 schon ab frühen Versionen, allerdings nur bei mittels Transport Layer Security (TLS) verschlüsselten Webseiten [Can I Use - HTTP/2 web]. Der Anteil der über HTTP/2 ausgelieferten Webseiten liegt momentan bei circa 3% [W3Techs 2015].

2.5.6 Asynchrones Laden von Imports

Das Rendern der Seite kann unter Umständen sehr lange dauern, da das in den Imports enthaltene JavaScript das Rendern blockiert. Um das zu verhindern und alle Dateien möglichst schnell laden zu können, ist ein `async`-Attribut vorgesehen. Dieses funktioniert jedoch nur, wenn als Protokoll HTTP/2 gewählt wird. Das Attribut ermöglicht es, dass mehrere Dateien asynchron geladen werden können. In den Imports enthaltenes JavaScript blockiert somit nicht das Rendern von bereits geladenem HTML Code. Falls HTTP/2 nicht vorhanden sein sollte, kann alternativ das `defer`-Attribut gewählt werden, wodurch das JavaScipt erst nach vollständigem Parsen des HTML ausgeführt wird. Eine weitere Methode ist das Laden der Imports am Ende der Seite. Somit wird sichergestellt, dass die Scripte erst ausgeführt werden, wenn die Seite geladen und gerendert wurde.

2.5.7 Anwendungen

Besonders einfach machen HTML Imports das Einbinden ganzer Web Applikationen mit HTML/JavaScript/CSS. Diese können in eine Datei geschrieben, ihre Abhängigkeiten definieren und von anderen importiert werden. Dies macht es sehr einfach, den Code zu organisieren. So können etwa einzelne Abschnitte von Anwendungen oder Code in einzelne Dateien ausgelagert werden, was Web Applikationen modular, austauschbar und wiederverwendbar macht. Falls nun ein oder mehrere Custom Elements in einem HTML Import enthalten sind, so werden dessen Interface und Definitionen automatisch gekapselt. Auch wird die Abhängigkeitsverwaltung in Betracht auf die Performance stark verbessert, da der Browser nicht eine große JavaScript-Library, sondern einzelne kleinere JavaScript-Abschnitte parsen und ausführen muss. Diese werden durch die HTML Imports parallel geparst, was mit einem enormen Performance-Schub einher geht [HTML5Rocks 2013].

2.5.8 Browserunterstützung

HTML Imports sind noch nicht vom W3C standardisiert, sondern befinden sich noch im Status eines “Working Draft” [Can I Use - HTML Imports web]. Sie werden deshalb bisher nur von Google Chrome ab Version 43 und Opera ab Version 33 nativ unterstützt (siehe Abbildung 2.8). Seitens Mozilla und dessen Browser Firefox wird es für HTML Imports auch keine Unterstützung geben, da ihrer Meinung nach der Bedarf an HTML Imports nach der Einführung von ECMAScript 6 Modules nicht mehr existiert und da Abhängigkeiten schon mit Tools wie Vulcanize aufgelöst werden können [Mozilla 2015]. Auf Details zu ECMAScript 6 Modules wird an dieser Stelle nicht eingegangen, da diese den Umfang dieser Arbeit überschreiten.

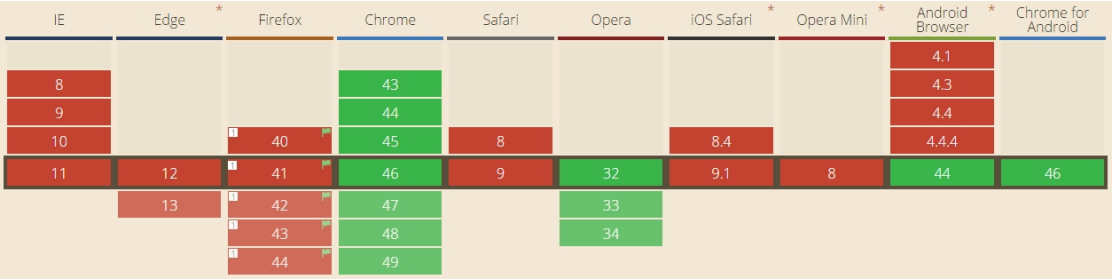

Abbildung 2.8: Browserunterstützung der HTML Imports

2.6 Polyfills

In den Abschnitten 2.2 bis 2.5 wurde gezeigt, wie die Web-Components-Technologien funktionieren und ob diese bereits in allen Browsern unterstützt werden. In diesem Abschnitt wird genauer darauf eingegangen was Polyfills sind. Ebenso wird auf deren Performance eingegangen und wie sie die Browserunterstützung der Web-Components-Technologien verbessern.

2.6.1 Native Browserunterstützung von Web Components

In den einzelnen Unterkapiteln zu den Technologien wurde jeweils kurz gezeigt, ob diese von den Browsern unterstützt wird oder nicht. Es wurde deutlich, dass Chrome und Opera bisher die einzigen Vorreiter sind. Bis auf HTML Templates, welche von allen modernen Browsern unterstützt werden, unterstützen sie als einzige alle Technologien. [Bateman 2014]

Chrome Hat alle Spezifikationen der Web-Component-Standards ab Version 43 komplett implementiert.

Firefox Unterstützt nativ HTML Templates. Custom Elements und Shadow DOM sind zwar implementiert, müssen aber über das Flag `dom.webcomponents.enabled` manuell in den Entwicklereinstellungen aktiviert werden. HTML Imports werden, wie in Kapitel 2.5 erwähnt, bis auf Weiteres nicht unterstützt.

Safari HTML Templates werden ab Version 8 unterstützt, Custom Elements und Shadow DOM befinden sich in der Entwicklung (Stand Januar 2016), HTML Imports werden jedoch nicht unterstützt.

Internet Explorer Als einziger Browser unterstützt der Internet Explorer keine der Web-Components-Technologien. Die Unterstützung wird - auf Grund der Einstellung der Entwicklung und des Wechsels zu Microsoft Edge - auch nicht nachträglich implementiert werden.

Microsoft Edge Templates werden ab Version 13 unterstützt, über die Entwicklung der restlichen Technologien kann allerdings abgestimmt werden [Microsoft web].

Mobile Browser Alle Technologien werden bisher nur auf Android in den Browsern Chrome für Android, Opera und Android Browser unterstützt.

Die Browserunterstützung der Technologien der Web Components ist momentan also noch verhalten. Das bedeutet jedoch nicht, dass sie noch nicht verwendet werden können. Mittels JavaScript besteht die Möglichkeit, die Technologien den Browsern beizubringen, welche sie nicht unterstützen. Ein solches JavaScript wird "Polyfill" genannt.

2.6.2 Polyfill webcomponents.js

> A polyfill, or polyfiller, is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide natively. [Sharp 2010]

Mit Hilfe von Polyfills können die Technologie-Lücken der Browser also auf mehrere, unterschiedliche Arten ("Poly") gefüllt ("fill") werden [Satrom 2014]. Eine Sammlung an Polyfills Technologien der Web Components bildet das JavaScript webcomponents.js. Es wurde von Google im Rahmen von Polymer entwickelt und hat eine dermaßen weite Verbreitung erfahren, dass es auszugliedern wurde. Somit kann es auch unabhängig von der Benutzung von Polymer eingesetzt werden [WebComponents - Polyfills web].

2.6.3 Browserunterstützung

Mit dem Einsatz der webcomponents.js Polyfills werden die Web Components auch auf den Internet Explorer, Firefox sowie Safari portiert. Eine detaillierte Matrix der Browserunterstützung der Web Components mit Einsatz der Polyfills ist in Abbildung 2.9 [WebComponents - webcomponents.js web] dargestellt.

Polyfill	IE10	IE11+	Chrome*	Firefox*	Safari 7+*	Chrome Android*	Mobile Safari*
Custom Elements	~	✓	✓	✓	✓	✓	✓
HTML Imports	~	✓	✓	✓	✓	✓	✓
Shadow DOM	✓	✓	✓	✓	✓	✓	✓
Templates	✓	✓	✓	✓	✓	✓	✓

Abbildung 2.9: Browserunterstützung der Web Components Technologien mit webcomponents.js

Jedoch werden die Technologien der Web Components auch mit Einsatzes der Polyfills nur von den aktuelleren Versionen des jeweiligen Browsers unterstützt. Darunter fallen jedoch weiterhin nicht ältere Browser, wie beispielsweise der Internet Explorer in Version 8 und 9. Des Weiteren werden einige Technologien auf Grund der Komplexität nicht komplett simuliert. Hier muss bei einigen Technologien auf folgende Punkte geachtet werden.

Custom Elements Die CSS-Pseudoklasse `:unresolved` wird nicht unterstützt.

Shadow DOM Das Shadow DOM kann auf Grund der Kapselung nicht komplett künstlich simuliert werden, dennoch versucht das webcomponents.js Polyfill einige der Features zu simulieren. So sprechen definierte CSS-Regeln alle Elemente in einem künstlichen Shadow Root an - Als würde man den >>> Selektor benutzen - auch die `::shadow` und `::content` Pseudoelemente verhalten sich so.

HTML Templates Templates, welche mit einem Polyfill erzeugt werden, sind nicht unsichtbar für den Browser, ihre enthaltenen Ressourcen werden also schon beim initialen Laden der Seite heruntergeladen.

HTML Imports Die zu importierenden HTML-Dateien werden mit einem XMLHttp-Request (XHR), und somit asynchron heruntergeladen, selbst wenn das `async`-Attribut (siehe Abschnitt 2.5.6) nicht gesetzt ist.

2.6.4 Performance

Das webcomponents.js-JavaScript [WebComponents - webcomponents.js web] bringt mit seiner Größe von 116 KB einen großen Umfang mit, was sich negativ auf die Ladezeiten der Webseite auswirkt. Des Weiteren müssen die von den Browsern nicht unterstützten und ignorierten CSS-Regeln - wie `::shadow` oder `::slotted` - mit Regular Expressions nachgebaut werden, was momentan 40 Stück sind. Das macht die Polyfills extrem komplex und träge. Die Funktionen zum Traversieren des DOMs müssen angepasst werden, damit nur die richtigen Elemente angezeigt werden und eine Shadow Boundary simuliert wird. Diese werden mit 42 Wrappern umgesetzt, was wie die Regular Expressions zur Simulation der CSS-Regeln sehr aufwändig ist. Allerdings können einige Funktionen wie `window.document` schlichtweg nicht überschrieben werden. Im Allgemeinen wird die DOM-API stark verlangsamt, wodurch die Performance - speziell auf mobilen Geräten - drastisch sinkt und mitunter nicht tolerierbar ist [Polymer - Shady DOM web].

2.6.5 Request-Minimierung mit "Vulcanize"

Webseiten können viele verschiedene, modular aufgebaute JavaScript-Dateien, Stylesheets, etc. beinhalten, welche die Anzahl an Requests erhöhen. Um die Anzahl an Requests zu verringern gibt es in der Webentwicklung bereits mehrere verschiedene Hilfsmittel. So werden die einzelnen Stylesheets oder auch JavaScript Dateien zu einer einzigen Datei konkateniert, sodass für das komplette Styling und JavaScript jeweils eine große Datei entsteht, welche nur einen Request an den Server benötigen. Zusätzlich kann sie anschließend noch minifiziert werden, um ihre Größe zu verringern und die Ladezeiten zu verkürzen. Selbiges Prinzip kann auch auf die HTML Imports angewendet werden. Google stellt hierfür das Tool Vulcanize [Polymer - Vulcanize web] bereit, welches serverseitig ermöglicht, einzelne kleine Web Components in einer einzigen, großen Web

Component zusammenzufassen. Benannt nach der Vulkanisation, werden metaphorisch die einzelnen Elemente in ein beständigere Materialien umgewandelt. Vulcanize reduziert dabei eine HTML-Datei und ihre zu importierenden HTML-Dateien in eine einzige Datei. Somit werden die unterschiedlichen Requests in nur einem einzigen gebündelt und die Ladezeiten sowie die benötigte Bandbreite minimiert.

2.7 Implementierung einer Komponente mit den nativen Web-Component-APIs

Anhand der vorhergehenden Abschnitte wird in diesem Abschnitt die Implementierung der Komponente <`custom-element`> mit den nativen HTML APIs erläutert. Diese soll dabei das Markup in einem Shadow DOM kapseln und den übergebenen Inhalt darstellen. Des Weiteren soll dessen Farbe über das Attribut `theme` optional konfiguriert werden können.

2.7.1 Custom Element mit Eigenschaften und Funktionen definieren

Um ein neues Custom Element zu registrieren, wird zunächst ein `HTMLElement` Prototyp `CustomElementProto` mittels `Object.create(HTMLElement.prototype)` erstellt. Dieser wird anschließend um die Eigenschaft `theme` und dessen Standardwert `style1` erweitert, welches das deklarativ konfigurierbare Attribut `theme` abbildet. Nun können die Lifecycle-Callback-Funktionen `createdCallback` und `attributeChangedCallback` der Komponente definiert werden (siehe Listing 2.15).

```
1  CustomElementProto.createdCallback = function() {
2    if (this.hasAttribute('theme')) {
3      var theme = this.getAttribute('theme');
4      this.setTheme(theme);
5    } else {
6      this.setTheme(this.theme);
7    }
8  };
9
10 CustomElementProto.attributeChangedCallback = function(attr, oldVal,
       newVal) {
11   if (attr === 'theme') {
12     this.setTheme(newVal);
13   }
14 };
```

Listing 2.15: Custom Element mit Eigenschaften und Funktionen definieren

Die `createdCallback`-Funktion soll zunächst prüfen ob das Attribut `theme` des <`custom-element`>-Tags verwendet und ein entsprechender Wert gesetzt wurde und übergibt dieses der Hilfsfunktion `setTheme`. Wird das Attribut nicht gesetzt, wird der Standardwert `style1` übergeben. Falls das `style`-Attribut von Außen geändert wird, soll die `attributeChangedCallback` Funktion gewährleisten, dass die Änderung auch von der Kompo-

nente übernommen wird, indem sie das Attribut der Hilfsfunktion `setTheme` übergibt. Um das Setzen und Ändern des `theme`-Attributs zu implementieren wird zuletzt die Hilfsfunktion `setTheme` für den Prototyp definiert (siehe Listing 2.16).

```
1 CustomElementProto.setTheme = function(val) {
2   this.theme = val;
3   this.outer.className = "outer " + this.theme;
4 };
```

Listing 2.16: HilfsFunktion `setTheme` definieren

Diese setzt den übergebenen Parameter, das `theme`-Attribut, als Klasse auf den umschließenden Wrapper `.outer`, welche dabei den zu verwendenden Style der Komponente bestimmt. Da der Prototyp nun alle erforderlichen Eigenschaften besitzt, kann er mit `document.registerElement("custom-element", { prototype: CustomElementProto });` als HTML-Tag `custom-element` in dem importierenden Dokument verfügbar gemacht werden.

2.7.2 Template erstellen und Styles definieren

Bisher ist das Custom Element zwar funktional, bietet aber noch kein gekapseltes Markup. Hierfür wird ein Template mit der ID `myElementTemplate` angelegt (siehe Listing 2.17), welches die für die Komponente notwendige HTML Struktur beinhaltet.

```
1 <template id="myElementTemplate">
2   <div class="outer">
3     Welcome in the Web Component
4     <div class="name">
5       <content></content>
6     </div>
7   </div>
8 </template>
```

Listing 2.17: Template erstellen und Styles definieren

Dieses enthält dabei einen Insertion Point `<content>`, in welchem die Kind-Elemente der Komponente in das interne Markup projiziert werden. Zusätzlich werden 2 Hilfs-Wrapper und Text definiert, damit die Elemente schneller mittels JavaScript selektierbar sind und das gewünschte Aussehen erreicht wird. Um nun die verschiedenen, mittels dem `theme`-Attribut konfigurierbaren, Styles zur Verfügung zu stellen, werden diese, neben generellen Styles, in einem `<style>`-Tag innerhalb des Templates definiert (siehe Listing 2.18). In diesem Beispiel werden 2 Optionen, `style1` und `style2` zur Verfügung gestellt.

```
1 <style>
2   .outer { /* Generelle Styles */ }
3   .style1 { color: green; }
4   .style2 { color: blue; }
5   .name { font-size: 35pt; padding-top: 0.5em; }
6 </style>
```

Listing 2.18: Styles definieren

Das Template wird nun zwar schon heruntergeladen, jedoch noch nicht in den DOM eingefügt. Hierzu muss es dem Shadow Root hinzugefügt werden, was in dem nachfolgenden Abschnitt 2.7.3 dargestellt wird.

2.7.3 Template bereitstellen und Shadow DOM zur Kapselung benutzen

Bevor das erstellte Template eingebunden werden kann, muss zunächst ein Shadow Root mittels `var shadow = this.createShadowRoot();` erzeugt werden. Hierfür wird die bereits definierte Lifecycle-Callback-Funktion `createdCallback` erweitert. Somit kann das Shadow DOM sofort initialisiert werden, sobald das Element erzeugt wurde. Nun kann der Inhalt des Templates mit der ID `myElementTemplate` mittels `var template = importDoc.querySelector('#myElementTemplate').content;` importiert und mit der Anweisung `shadow.appendChild(template.cloneNode(true));` dem Shadow Root hinzugefügt werden. Die Variable `importDoc` stellt dabei die Referenz auf die importierte Komponente, also das <`custom-element`>-Element, dar und kann mittels der Funktion `var importDoc = document.currentScript.ownerDocument;` ermittelt werden. Wird dies nicht getan, so würde der `querySelector` auf das Eltern-Dokument der eingebetteten Komponente zugreifen und das Template nicht finden. Nun ist der Inhalt des Templates als Shadow DOM innerhalb des Elements gekapselt und nach Außen nicht sichtbar. Die gerenderte Komponente wird in Abbildung 2.10 dargestellt.

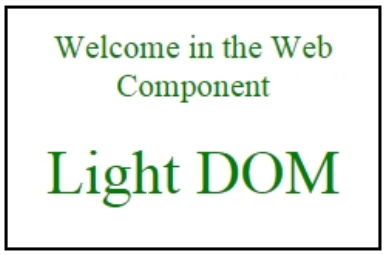

Abbildung 2.10: Gerenderte Komponente mit nativen APIs

2.7.4 Element importieren und verwenden

Das Element ist somit vollständig und kann in einer beliebigen Webseite oder Applikation eingesetzt werden. Hierzu muss das Element mittels <`link rel="import" href="elements/custom-element.html"`> zunächst importiert werden. Es kann anschließend mit entsprechenden Attributen und Inhalt in die Webseite eingebunden werden, beispielsweise mit der Konfiguration <`custom-element theme="style1"`>`Light DOM`</`custom-element`>. Das vollständige Beispiel der Komponente, sowie dessen Einbindung in ein HTML-Dokument sind im Anhang A zu finden.

3 Einführung in Polymer

Die Library Polymer setzt auf die in Kapitel 2 gezeigten Web-Components-Standards auf und soll den Umgang mit ihnen vereinfachen sowie deren Funktionalitäten erweitern. Dadurch will es Polymer ermöglichen, gekapselte Komponenten zu entwickeln, welche wiederum von Komponenten verwendet oder mit anderen Komponenten verbunden werden können. Diese Komponenten können dann zu einer komplexen Applikationen zusammengefügt werden. Der Name Polymer ("Poly" - mehrere, "mer" - Teile) ist dabei eine Metapher für die Polymerisation von einzelnen Monomeren (den nativen HTML-Elementen) zu einem großen Molekül (einer Web-Komponente). In diesem Kapitel wird in Abschnitt 3.1 die Architektur der Library gezeigt, in Abschnitt 3.2 wird der darauf aufsetzende Elemente-Katalog dargestellt.

3.1 Architektur

Eine mit Hilfe von Polymer implementierte Komponente lässt sich in mehrere Schichten, wie in Abbildung 3.1 dargestellt, unterteilen. Die Browser-Schicht stellt die nativen APIs der Web-Technologien dar, welche von der Polyfill-Schicht, den webcomponents.js-Polyfills (siehe Abschnitt 2.6), ersetzt oder erweitert werden können, falls der Browser die notwendige Technologie nicht unterstützt. Polymer kann dabei als Konformitäts-Schicht aufgefasst werden, welche auf die nativen Technologien bzw. den Polyfills aufsetzt und sich aus folgenden 3 Schichten zusammensetzt [Polymer - Documentation web]: Der `polymer-micro`-Schicht, welche die grundlegenden Funktionalitäten für das Erzeugen von Custom Elements bietet, der `polymer-mini`-Schicht, welche den Umgang mit einem lokalen DOM in einer Polymer-Komponente erweitert und erleichtert, und zuletzt der `polymer-standard`-Schicht mit allgemeinen und zusätzlichen Funktionalitäten für den Umgang mit Web Components. Auf die Polymer-Schicht können die Polymer-Elemente aufgesetzt werden. Diese bilden eine weitere Schicht, welche durch den Elemente-Katalog [Polymer - Element Catalog web] repräsentiert wird. In dem Elemente-Katalog sind diverse mit Polymer umgesetzte Komponenten - sowohl für das User Interface (UI) als auch für Kernfunktionen zum Entwickeln von Applikationen - vorhanden.

3.2 Elemente-Katalog

Der von Google verwaltete Elemente-Katalog verkörpert die Polymer-Philosophie "There is an element for that" [Savage 2015]. Sie bilden eine Sammlung an Komponenten, welche die von Google vorgeschlagenen Implementierungen als Lösungen von einfachen bis komplexen wiederkehrenden Problemen sind. Entwickler können diese in ihrer eigenen Applikation optional einsetzen, sie sind beim Einsatz von Polymer aber nicht zwingend notwen-

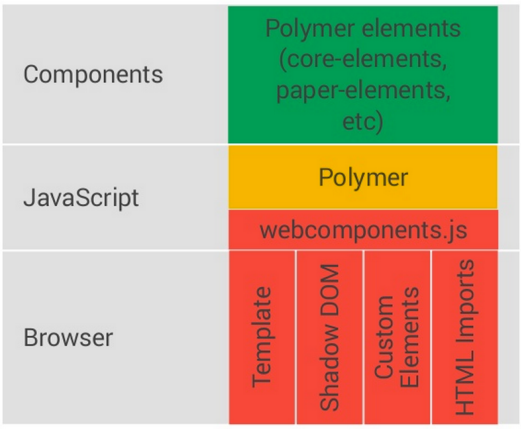

Abbildung 3.1: Schichtenmodell von Polymer

dig. Es werden dabei allerlei Anwendungsmöglichkeiten behandelt, von DOM-Rendering in Form von Animationen, über Browser-API-Interaktionen durch Push-Nachrichten, bis hin zu Remote-API-Interaktionen mittels einem XHR. Der Katalog besteht aus 7 Kategorien (Stand Januar 2016), welche die Komponenten nach Anwendungsfällen sortieren. Nachfolgend werden die Kategorien aufgelistet und erläutert.

Iron Elements - Fe Eisen ist der Kern der Erde. Daran orientiert sich die Metapher der Iron Elements, welche den Kern von Polymer und das Zentrum der Polymer Elemente bilden. Sie sind die wichtigsten Elemente, welche in vielen Projekten benötigt werden.

Paper Elements - Md Die Paper Elements sind Googles-Design-Philosophie "Material Design" gehorchende Elemente wie Listen, Menüs, Tabs. Sie ermöglichen das Erstellen einer UI. Paper ist dabei eine Metapher für ein erweitertes Papier, es kann zusammengesteckt werden, sich transformieren oder Schatten werfen.

Google Web Components - Go Um die eigenen Services leichter verwendbar zu machen, stellt Google die Google Web Components bereit. Sie kapseln diese Services und APIs in Komponenten, wodurch Google Maps oder auch Google Drive usw. in der eigenen Applikation eingebunden und verwendet werden können.

Gold Elements - Au Die Gold Elements sind eine Sammlung an buchstäblich wertvollen, goldenen Elementen, welche im Bereich E-Commerce eingesetzt werden können. Sie sind Komponenten für das Arbeiten mit Zahlungsmethoden oder Kreditkarteninformationen und können dabei Helfen, die Conversion-Rate der Applikation zu erhöhen.

Neon Elements - Ne Die Neon Elements sind neben den Paper Elements auf UI-Ebene einsetzbar. Mit ihnen können beispielsweise Animationen erstellt werden.

Platinum Elements - Pt Wie die Gold Elements sind auch die Platinum Elements sehr wertvoll, jedoch in einer anderen Hinsicht. Sie sind nicht kommerziell orientiert, sondern für im Hintergrund laufende Services wie Push- oder Offline-Funktionalitäten etc. gedacht. Sie sind Lösungen für sehr schwer lösbare, komplexe Probleme.

Molecules - Mo Die Molecules sind weitere Elemente in Form von Wrappern für third-party-Libraries.

Carbon Elements - C (in Entwicklung) Die Carbon Elements befinden sich zum Stand dieser Arbeit noch in Entwicklung und sind noch nicht fertiggestellt. Sie sind metaphorisch sehr gewichtige Elemente und besitzen die Fähigkeit zur Bildung komplexer Moleküle, die essenziell für lebende Strukturen sind. Im übertragenen Sinn sind sie framework-orientierte Elemente und werden sich um strukturelle Probleme auf Applikations-Ebene kümmern.

3.3 Alternative Sammlung von Komponenten

Statt den Polymer-Komponenten können auch selbst entwickelte oder aus anderen Quellen stammende Komponenten in eigenen Projekten verwendet werden. Der Elemente-Katalog wird dabei nur von Google vertrieben und erlaubt keine nicht von Google entwickelten Komponenten. Als eine Alternative für den Polymer Elemente-Katalog kann der customelements.io-Katalog [customelements.io web] herangezogen werden. In diesem sind bereits mehrere tausend Komponenten gesammelt, welche von Google-unabhängigen Entwicklern für die unterschiedlichsten Anwendungsfälle entwickelt wurden.

4 Analogie zu nativen Web Components

In erster Linie vereinfacht Polymer das Erstellen und den Umgang mit Custom Elements, wobei versucht wird die von den Web-Components-Standards definierten Prämissen einzuhalten und zu erweitern. Ebenso wie bei bei dem nativen Erzeugen einer eigenen Komponente, werden bei Polymer für jede Komponente einzelne HTML-Dateien erstellt, welche die Elemente repräsentieren. Darin werden alle JavaScript-, CSS- und HTML-Strukturen gesammelt und gekapselt. Man distanziert sich somit von der Code-Trennung, stattdessen werden alle Element-spezifischen Inhalte in einer Datei gebündelt. Wie das funktioniert und wie die Technologien mit Polymer umgesetzt werden, wird in diesem Kapitel dargestellt. In Abschnitt 4.1 werden dabei die Custom Elements erläutert, in Abschnitt 4.2 der Shadow DOM zusammen mit den HTML Templates und in Abschnitt 4.3 schließlich die HTML Imports.

4.1 Custom Elements

Die Intention der Polymer-Library ist das Erstellen eigener Komponenten oder Elemente, den Custom Elements. Das Erstellen, Erweitern und Verwalten von Custom Elements kann mit den nativen Mitteln bei steigender Komplexität mitunter schwierig werden. Polymer stellt hierfür eine Reihe an hilfreicher Funktionen bereit, welche das Arbeiten mit den Web-Components-Standards erleichtern und erweitern sollen [Polymer - Documentation web].

4.1.1 Neues Element registrieren

Mit der Funktion `Polymer(prototype);` können mit Polymer neue Elemente registriert werden, in dieser werden die generellen Polymer-Einstellungen in Form eines Prototyp-Objektes vorgenommen. Um ein Element zu definieren, muss der zu übergebende Prototyp die Eigenschaft `is: 'element-name'` haben, welche den HTML-Tag-Name des zu erstellenden Custom Elements angibt (in diesem Fall `element-name`). Der Name muss dabei als String übergeben werden und ebenso wie beim nativen Erzeugen eines Custom Elements einen Bindestrich enthalten. Die Polymer-Funktion registriert beim Aufruf automatisch das neue Element und gibt einen Konstruktor zurück, mit dem das Element instanziiert werden kann. Dies geschieht imperativ und deklarativ analog zu dem Erstellen mit der nativen Anweisung `var element = document.createElement('element-name');` bzw. mit dem zurückgegebenen Konstruktor oder im HTML-Markup mit dem erstellten HTML-Tag `<element-name></element-name>`. Soll statt dem Standard-Konstruktor ein Konstruktor erstellt werden, dem Argumente übergeben werden können, so muss in dem Prototyp die Methode `factoryImpl` mit den entsprechenden Argumenten definiert werden. Diese löst nach dem Ausführen einen `factoryImpl`-Callback (siehe

Abschnitt 4.1.1) aus. Die `factoryImpl`-Methode wird allerdings nur aufgerufen, wenn ein Element mit dem zurückgegebenen Konstruktor, nicht jedoch beim Verwenden der `document.createElement`-Methode oder durch HTML-Markup erzeugt wird.

4.1.2 Elemente erweitern

Ebenso wie nativ erzeugt Custom Elements können auch mit Polymer erzeugte Elemente die nativen Elemente erweitern, was unter Polymer ebenso als "Type Extension Custom Element" bezeichnet wird. Jedoch kann ein Polymer-Element nur native Elemente erweitern, andere Polymer-Elemente hingegen noch nicht. Dies soll allerdings in einer zukünftigen Version möglich sein. Das Erweitern und Erzeugen eines "Type Extension Custom Elements" funktioniert mit Polymer analog zu der nativen Methode. Hierzu muss im Prototyp die Eigenschaft `extends: 'HTMLElement'` gesetzt werden, wobei das `'HTMLElement'` ein natives HTML-Element wie z.B. `input` oder `button` sein muss. Zum Erzeugen des Elements kann nun wieder entweder die imperative Methode (siehe Abschnitt 4.1.1) oder die deklarative Methode analog dem nativen Erstellen mit `<HTMLElement is="my-HTMLElement">` mittels dem `is` Attribut, gewählt werden.

4.1.3 Declared Properties - Eigenschaften und Methoden definieren

Custom Elements können auch mit Hilfe von Polymer um Eigenschaften und Methoden erweitert werden. Hierzu bietet Polymer das `properties`-Objekt an, mit welchem die Eigenschaften der Komponente im JSON-Format definiert werden können. Im Gegensatz zu den nativen Methoden muss somit nicht jede Eigenschaft bzw. Property einzeln hinzugefügt werden. Die in diesem Objekt definierten Properties können beim Verwenden der Komponente im HTML-Markup konfiguriert werden. Dadurch ist es möglich, eine API für das Element zu erstellen, da die von außen konfigurierbaren Attribute ein Interface der Komponente bilden. Eine Property kann dabei mit folgenden Parametern spezifiziert werden.

type Konstruktor-Typ der Property, er kann ein Boolean, Date, Number, String, Array oder Object sein.

value Standardwert der Property, der Wert muss boolean, number, string oder eine Funktion sein.

reflectToAttribute Boolean-Wert, gibt an ob die Property mit dem HTML-Attribut synchronisiert werden soll. Das bedeutet, dass falls die Property geändert wird, das zugehörige HTML-Attribut des Elements geändert wird, was äquivalent zu `this.setAttribute(property, value);` ist. Dabei muss der Name des HTML-Attributs in dem Properties-Objekt kleingeschrieben werden. Enthält der Name zusätzlich Bindestriche, so muss die Property kleingeschrieben und die Bindestriche

entfernt werden, wobei jedes Wort nach einem Bindestrich groß geschrieben werden muss.

readOnly Boolean-Wert, gibt an ob die Property nur gelesen oder auch geschrieben werden soll (siehe Abschnitt 5.1).

notify Boolean-Wert, die Property ist für Two-Way-Data-Binding (siehe Abschnitt 5.1) verfügbar, falls true gesetzt ist. Zusätzlich wird ein `property-name-changed`-Event ausgelöst, wenn sich die Property ändert.

computed Name der Funktion als String, welche den Wert der Property berechnen soll (siehe Abschnitt 4.1.4).

observer Name der Funktion als String, welche aufgerufen wird, wenn der Wert der Property geändert wird (siehe Abschnitt 4.1.5).

Die genannten Parameter sind alle optional anzugeben. Wird keine der Parameter definiert, so kann die Property direkt mit dem `type` definiert werden. Soll also beispielsweise eine Property `propertyName` als String und ohne Parameter angegeben werden, so würde dies mit `properties: { propertyName: String }` erreicht werden.

4.1.4 Computed Properties

Polymer unterstützt des Weiteren zusammengesetzte, virtuelle Properties, welche aus anderen Properties berechnet werden. Um dies zu erreichen, muss die dafür verwendete Funktion im `properties`-Objekt mit entsprechenden Parametern angegeben werden. Soll beispielsweise die virtuelle Property `result` die Zusammensetzung der Properties a und b darstellen, so wird sie als `result: { type: String, computed: computeResult(a, b) }` im `properties`-Objekt angegeben. Die entsprechende Funktion muss dann als `computeResult: function(a, b) { ... }` im Polymer-Prototyp definiert werden. Diese wird einmalig aufgerufen, wenn sich eine der Eigenschaften a oder b ändert und wenn keine von beiden undefiniert ist. Der von ihr zurückgegebene Wert wird anschließend in der Variable `result` gespeichert.

4.1.5 Property Oberserver

Wird für eine Property der Parameter `observer` angegeben, so wird sie auf Änderungen überwacht. Die angegebene Funktion, welcher als optionale Argumente der neue und alte Wert übergeben werden können, wird somit aufgerufen, falls der Wert der Property geändert wird. Allerdings kann in dem `properties`-Objekt jeweils nur eine Property von einem Observer überwacht werden. Sollen mehrere Properties von demselben Observer überwacht werden, kann das `observers`-Array des Polymer-Prototyps verwendet werden. Soll beispielsweise die Funktion `computeResult(a, b)` ausgeführt werden, sobald

sich eine der Properties `a` oder `b` ändert, so kann diese Funktion in das Array übernommen werden. Jedoch wird die Funktion auch hier nur dann ausgeführt, wenn keiner der Werte undefiniert ist. Des Weiteren wird bei angegebenen Funktionen - im Gegensatz zu den im `properties`-Objekt angegebenen Observern - nur der neue Wert statt des neuen und des alten Werts übergeben. Mit dem `observers`-Array ist es auch möglich, Sub-Properties oder Arrays zu überwachen.

4.1.6 Das hostAttributes-Objekt

Zusätzlich zu den Declared Properties können auch HTML-Attribute im Polymer-Prototyp definiert werden. Hierzu bietet Polymer das `hostAttributes`-Objekt an. Die darin angegebenen Schlüssel-Wert-Paare werden beim initialen Erstellen des Elements auf dessen Attribute abgebildet. Das `hostAttributes`-Objekt kann dabei alle HTML-Attribute bis auf das `class`-Attribut definieren, darunter fallen beispielsweise die Attribute `data-*`, `aria-*` oder `href`. Wird im `hostAttributes`-Objekt beispielsweise das Attribut `selected` mit `true` definiert, wird es bei einem `my-element`-Element in Form von `<my-element selected>Item</my-element>` ausgegeben. Wichtig ist hier die Serialisierung der Schlüssel-Wert-Paare. Wird ein String, Date oder Number als Wert übergeben, so werden diese als String serialisiert, werden jedoch Arrays oder Objekte übergeben, so werden diese mittels `JSON.stringify` serialisiert. Boolean-Werte werden bei `false` entfernt und bei `true` angezeigt. Um Daten in der anderen Richtung von einem HTML-Element an das `hostAttributes`-Objekt zu propagieren, muss auf eine alternative Syntax zugegriffen werden (siehe Abschnitt 5.1.3).

4.1.7 Lifecycle-Callback-Funktionen

Die nativen Lifecycle-Callback-Funktionen (siehe Abschnitt 2.2.4) werden ebenso von Polymer unterstützt. Diese können in dem Prototyp als Attribut bei ihrem normalen Namen oder in verkürzter Form angegeben werden, so heißt die `createdCallback`-Methode `created`, die `attachedCallback`-Methode heißt `attached` etc.. Soll beispielsweise die `created`-Methode definiert werden, so kann diese mit `created: function { ... }` in dem Prototyp angegeben werden. Zusätzlich bietet Polymer einen `readyCallback`, welcher aufgerufen wird, nachdem Polymer das Element erstellt und den lokalen DOM initialisiert hat, also nachdem alle im lokalen DOM befindlichen Elemente konfiguriert wurden und jeweils ihre `ready`-Methode aufgerufen haben. Sie ist besonders hilfreich, wenn nach dem Laden der Komponente dessen DOM nachträglich manipuliert werden soll. Falls mit den Lifecycle-Callbacks gearbeitet wird, muss auf die richtige Anwendung der Reihenfolge geachtet werden. So werden die Callbacks eines Elements in der Reihenfolge `created`, `ready`, `factoryImpl` und `attached` ausgeführt.

4.2 Shadow DOM und HTML Templates

Die bisher gezeigten Methoden ermöglichen das Erstellen einer Polymer-Komponente, die jedoch noch kein internes Markup beinhaltet. Wie bei den nativen Technologien können auch mit Polymer erzeugte Custom Elements um HTML-Markup, das lokale DOM, erweitert werden [Polymer - Documentation web]. Hierzu dient das Polymer <dom-module>-Element, welches als ID den Wert der is-Property des Polymer-Prototyps haben muss. Das zu verwendende HTML-Markup muss dann in einem <template>-Tag dem <dom-module> hinzugefügt werden. Auf das Klonen des Inhalts des Templates mittels der importNode-Funktion (siehe Abschnitt 2.3.3) kann hierbei verzichtet werden, da Polymer diesen automatisch in den lokalen DOM des Elements klont. Soll also das lokale DOM des <element-name>-Tags deklariert werden, so wird dies wie in Listing 4.1 dargestellt erreicht:

```
1 <dom-module id="element-name">
2   <template>Local DOM / Inneres HTML Markup</template>
3
4   <script>
5     Polymer({
6       is: 'element-name'
7     });
8   </script>
9 </dom-module>
```

Listing 4.1: Template einer Polymer-Komponente

Der deklarative Teil des Elements, das <dom-module> und dessen Inhalte, sowie der Imperative Teil mit dem Polymer({ ... })-Aufruf können entweder in derselben oder in getrennten HTML-Dateien stehen. Hierbei spielt es jedoch keine Rolle, ob das <script>-Tag innerhalb oder außerhalb des <dom-module>-Tags steht, solange das Template vor dem Polymer-Funktionsaufruf geparst wird.

4.2.1 Shady DOM

Wie in Abschnitt 2.4.9 gezeigt, wird das Shadow DOM nicht von allen Browsern unterstützt, ebenso ist der Polyfill für dieses, aufgrund dessen schlechter Performance (siehe Abschnitt 2.6.4), nur als allerletzte Instanz zu sehen. Aus diesen Gründen ist in Polymer das sogenannte “Shady DOM” implementiert [Polymer - Shady DOM web]. Dieses bietet einen dem Shadow DOM ähnlichen Scope für den DOM-Tree. dabei rendert er das DOM, als wenn kein Shadow DOM in dem Element vorhanden wäre. Dies bringt wiederum auch die dadurch entstehenden Nachteile mit sich, wie dass internes Markup nach außen sichtbar ist oder keine Shadow Boundary verfügbar ist. Der Vorteil ist jedoch, dass das Shady DOM genug Methoden für seinen Scope bereitstellt, um sich wie ein Shadow DOM verhalten zu können, ohne die Performance zu mindern. Hierzu ist es jedoch zwingend notwendig, die eigens entwickelte Shady-DOM-API im Umgang mit dem DOM

zu benutzen, was mit der `Polymer.dom(node)`-Funktion erreicht wird. Will man beispielsweise alle Kinder mit der `children`-Eigenschaft des Shadow-Host-Elements <`my-element`> selektieren, so erfolgt dies mit das Shady-DOM-API mittels `var children = Polymer.dom(my-element).children;` statt mit der normalen DOM-API mittels `var children = document.getElementsByTagName("my-element")[0].children;`. Diese gibt dann nur die von außen übergebenen (Light-DOM-)Elemente zurück, ohne die Elemente des internen Markups des Templates zu berücksichtigen. Die Shady-DOM-API bildet dabei alle Funktionen der nativen DOM-API ab und ist performanter als der Shadow-DOM-Polyfill, da nicht dessen Verhalten, sondern nur ein eigener DOM-Scope implementiert ist. Jedoch beschränkt sich Polymer nicht nur auf den eigenen Shady DOM, vielmehr ist es mit dem nativen Shadow DOM kompatibel, sodass die Shady-DOM-API auf das native Shadow DOM zugreifen kann, falls dieses von dem Browser unterstützt wird. Dadurch kann eine Applikation implementiert werden, die auf allen Plattformen mit einer verbesserten Performance ausgeführt wird, wovon besonders die mobilen Plattformen profitieren. Standardmäßig benutzt Polymer jedoch immer die eigene Shady-DOM-API, wie das verhindert werden kann, ist in Abschnitt 6.1.2 dargestellt.

4.2.2 DOM-Knoten automatisch finden

Um das Traversieren in dem lokalen DOM zu beschleunigen, bietet Polymer eine Hilfsfunktion für das automatische Finden eines Elements, auch “Automatic Node Finding” genannt, an. Hierzu wird intern ein Mapping zu den statisch erzeugten DOM-Elementen erzeugt, indem jedes Element des lokalen DOM-Templates, für welches eine ID definiert wurde, in dem `this.$`-Hash gespeichert wird. Hat nun also das Element <`div id="wrapper"></div`> in dem Template die ID `wrapper`, so kann es in Polymer mittels `this.$.wrapper` selektiert werden. Jedoch werden dem Hash nur die statisch erzeugten DOM-Knoten hinzugefügt, dynamisch mittels `dom-repeat` oder `dom-if` hinzugefügte Knoten allerdings nicht. Die dynamisch hinzugefügten Knoten können mit der `this.$$`-Funktion selektiert werden. So liefert die Funktion `this.$$(selector);` das erste Element, welches von den im Parameter `selector` enthaltenen CSS-Selektor selektiert wird.

4.2.3 Content Projection

Um nun Elemente des Light DOMs in das lokale DOM der Komponente zu injizieren, bietet Polymer ebenso das von den nativen Methoden bekannte <`content`>-Element an, welches einen Insertion Point des Light DOM im lokalen DOM der Komponente darstellt. Wie auch bei den nativen Insertion Points kann das <`content`>-Element auch nur selektierte Inhalte injizieren, indem das `select`-Attribut mit einem entsprechenden Selektor gesetzt wird. Falls das Shadow DOM in Polymer verfügbar ist, so wird eine Zusammenstellung des Shadow DOM und dem Light DOM gerendert. Ist das Shadow

DOM jedoch nicht verfügbar und das Shady DOM wird verwendet, so ist das zusammengesetzte DOM das tatsächliche DOM des Elements. Auch kann in Polymer mittels der `_observer`-Eigenschaft überwacht werden, ob Kind-Elemente der Komponente hinzugefügt oder von ihr entfernt werden. Dazu wird dieser die Funktion `observeNodes(callback)` zugewiesen, welche ausgeführt wird, wenn Elemente hinzugefügt oder entfernt werden. Der Parameter `callback` ist dabei eine anonyme Funktion, welche als Übergabewert das Objekt `info` hat, in welchem die hinzugefügten oder entfernten Knoten enthalten sind. Eine Implementierung der Überwachung des `contentNode`-Knotens auf Änderungen könnte dabei wie in Listing 4.2 dargestellt aussehen.

```
1 this._observer = Polymer.dom(this.$.contentNode).observeNodes(function(
      info) {
2   this.processNewNodes(info.addedNodes);
3   this.processRemovedNodes(info.removedNodes);
4 });
```

Listing 4.2: Überwachung eines `contentNode`-Knotens

4.2.4 CSS-Styling

Die in Abschnitt 2.4.5 gezeigten Regeln für das Stylen des Shadow DOMs sind auch unter Polymer und dessen Shady DOM gültig [Polymer - Documentation web]. Zusätzlich können Komponenten CSS-Properties (also Variablen) nach außen sichtbar machen, damit diese von außerhalb der Komponente gesetzt werden können. Somit kann das CSS in einer gekapselten Komponente bestimmt werden. Hierbei können auch Standardangaben gemacht werden, die von der Komponente übernommen werden, wenn die Variable nicht definiert wird. Um eine Variable bereitzustellen, muss diese der entsprechenden Eigenschaft mit der Syntax `var(--variable-name, default)` in den Style-Regeln der Komponente angegeben werden. Das Beispiel in Listing 4.3 zeigt das DOM eines `x-element`, welches die CSS-Variable `x-element-button-color` und dem zugewiesenen Standardwert `red` für einen Button in einem -Tag mit der Klasse `x-element-container` bereitstellt.

```
1  <dom-module id="x-element">
2    <template>
3      <style>
4        .x-element-container > button {
5          color: var(--x-element-button-color, red);
6        }
7      </style>
8      <div class="x-element-container">
9        <button>Ich bin ein Button</button>
10     </div>
11   </template>
12 </dom-module>
```

Listing 4.3: `x-element`-Komponente mit einer CSS-Variablen

Die Applikation, welche die Komponente benutzt, kann nun die Variable `--x-element-button-color` definieren, wie in Listing 4.4 gezeigt wird.

```
<style is="custom-style">
  x-element {
    --x-element-button-color: green;
  }
</style>
```

Listing 4.4: Definition der CSS-Variable

Das Attribut `is="custom-style"` des `<style>`-Tag dient dabei als Anweisung für den Polyfill, da CSS-Properties noch nicht von allen Browsern unterstützt werden. Nun soll jedoch nicht für jedes CSS-Attribut eine Variable angelegt werden, da dies schnell unübersichtlich werden kann. Um mehrere CSS-Attribute einer Komponente ändern zu können, können sogenannte Mixins erstellt werden. Diese sind eine Sammlung an Styles, die auf eine Komponente angewendet werden können. Sie werden wie eine CSS-Variable definiert, mit dem Unterschied, dass der Wert ein Objekt ist, welches ein oder mehrere Regeln definiert. Um ein Mixin nach außen bereitzustellen, muss es in der Komponente in den CSS-Regeln mit `@apply(--mixin-name)` bereitgestellt werden. Es kann dann von der Applikation verwendet werden. So kann das obige Beispiel um das `--x-element`-Mixin erweitert werden (siehe Listing 4.5).

```
<dom-module id="x-element">
  <template>
    <style>
      .x-element-container > button {
        color: var(--x-element-button-color, red);
        @apply(--x-element);
      }
    </style>
  </template>
</dom-module>
```

Listing 4.5: Erweiterung der Styles um ein Mixin

Wobei es wie in Listing 4.6 dargestellt von der einsetzenden Applikation konfiguriert werden könnte.

```
<style is="custom-style">
  x-element {
    --x-element-button-color: green;
    --x-element: {
        padding: 10px;
        margin: 10px;
    };
  }
</style>
```

Listing 4.6: Verwendung eines Mixins einer Komponente

4.2.5 Gemeinsame Styles mehrerer Komponenten

Um nun Style-Regeln auf mehrere Komponenten anzuwenden, stellt Polymer die sogenannten "Style Modules" bereit. Diese ersetzen die ab Version 1.1 nicht mehr unterstützte Möglichkeit, externe Stylesheets zu verwenden. Style Modules sind dabei nichts anderes als Komponenten, welche von allen Komponenten importiert werden können, die diese Styles anwenden sollen.

Style Module anlegen

Um eine Komponente mit geteilten Style-Regeln zu erstellen, genügt es, dass diese die Style-Regeln in einem <dom-module>-Tag mit einer beliebigen ID definiert. Dies wird in dem Listing 4.7 der Datei "shared-styles.html" gezeigt. Darin wird eine Style-Regeln definiert, die den Text aller Elemente mit der Klasse `wrapper` Rot anzeigen lässt.

```
1 <dom-module id="shared-styles">
2   <template>
3     <style>
4       .wrapper { color: red; }
5     </style>
6   </template>
7 </dom-module>
```

Listing 4.7: HTML-Datei "shared-styles.html"

Style Module benutzen

Damit eine Komponente diese Styles nutzen kann, muss sie sie zunächst importieren und anschließend einen <style>-Tag definieren, welcher als `include`-Attribut den Namen der Komponente mit den geteilten Style-Regeln hat. Die Styles werden darin importiert und auf die gesamte Komponente angewendet. Somit wird der Text des -Tags mit der Klasse `wrapper` rot dargestellt (siehe Listing 4.8).

```
1 <link rel="import" href="shared-styles.html">
2 <dom-module id="x-element">
3   <template>
4     <style include="shared-styles"></style>
5     <div class="wrapper">Wrapper mit rotem Text</div>
6   </template>
7   <script>Polymer({is: 'x-element'});</script>
8 </dom-module>
```

Listing 4.8: Benutzung eines Style-Moduls

4.3 HTML Imports

Um mehrere Polymer Komponenten oder Komponenten innerhalb anderer Komponenten zu benutzen, verwendet Polymer HTML Imports. Diese funktionieren analog zu der Verwendung mit der nativen HTML Imports Technologie (siehe Abschnitt 2.5), wobei die selben Vor- und Nachteile auftreten. Polymer kümmert sich dabei lediglich automatisch im Hintergrund um die korrekte Einbindung der HTML-Dateien und dessen Bereitstellung im Dokument, falls ein <`link rel="import"`> in einer Komponente enthalten ist. Das manuelle Einbinden mittels der speziellen JavaScript-Methoden oder Eigenschaften, wie beispielsweise der `import`-Eigenschaft des importierten Elements, wird somit hinfällig.

4.3.1 Dynamisches Nachladen von HTML

Falls HTML-Dateien dynamisch zur Laufzeit nachgeladen werden sollen, bietet Polymer zusätzlich eine Hilfsfunktion an, mit der HTML Imports nachträglich ausgeführt werden können [Polymer - Base Documentation web]. Die hierfür bereitgestellte Funktion `importHref(url, onload, onerror);` importiert beim Aufruf dynamisch die angegebene HTML-Datei in das Dokument. Sie erstellt dabei ein <`link rel="import"`>-Element mit der angegeben URL und fügt es dem Dokument hinzu, sodass dieser ausgeführt werden kann. Wenn der Link geladen wurde, also der `onload`-Callback aufgerufen wird, ist die `import`-Eigenschaft des Links der Inhalt des urückgegebenen, importierten HTML-Dokuments. Der Parameter `onerror` ist dabei eine optionale Callback-Funktion, die beim Auftreten eines Fehlers aufgerufen wird.

5 Zusätzliche Polymer-Funktionalitäten

Polymer bietet eine zusätzliche Applikations-Schicht mit einigen hilfreichen Funktionalitäten, die das Arbeiten mit den Komponenten vereinfachen. Die wichtigsten werden in diesem Kapitel dargestellt. Abschnitt 5.1 widmet sich dabei um das One-Way- und Two-Way-Data-Binding, Abschnitt 5.2 erläutert die Behaviors und in Abschnitt 5.3 erklärt abschließend die Events.

5.1 One-Way- und Two-Way-Data-Binding

Für den Transport von Daten zwischen Komponenten sieht Polymer das One-Way- und Two-Way-Data-Binding vor. Die Daten können dabei zwischen einer Eigenschaft eines Custom Elements (Host-Element) und dessen lokalen DOM (Kind-Element) gebunden und somit zwischen Komponenten ausgetauscht werden. Hierfür sieht Polymer das Mediator-Pattern vor, welches besagt, dass Daten zwischen 2 nebenstehenden Komponenten ihre Daten über eine übergeordnete Komponente propagieren müssen. Dies erfolgt mittels einer hierfür vorgesehenen Syntax für Attribute eines Elements, wie zum Beispiel: `<my-element some-property={{value}}></my-element>`. Durch die doppelten Klammern kann die Eigenschaft `value` des Host Elements Daten in das Attribut `some-property` des Kind Elements weitergeben. Hierzu wird von Polymer für jede Eigenschaft eines Elements ein `propertyEffect`-Objekt und ein entsprechender Setter angelegt. Wenn die Eigenschaft zur Laufzeit geändert wird, wird über das Array aus propertyEffects iteriert und bei entsprechender Eigenschaft mittels dem Setter der neue Wert gesetzt. Wird die Eigenschaft von einem Observer (siehe Abschnitt 4.1.5) überwacht, so wird dieser im Setter aufgerufen, statt die Werte direkt zu ändern. Für das Binden stehen unterschiedliche Annotationen zur Verfügung, welche nachfolgend erläutert werden [The Chromium Projects web].

5.1.1 One-Way-Data-Binding

Das One-Way-Data-Binding erlaubt Attributen nur das Lesen der entsprechenden Eigenschaft seiner Komponente, ein schreibender Zugriff wird jedoch untersagt. Meist wird es verwendet, um Texte basierend auf einer Eigenschaft anzuzeigen, der Text jedoch soll nicht verändert werden oder die Eigenschaft überschreiben. Erreicht wird das One-Way-Data-Binding mit der doppelten eckigen Klammer Syntax `[[]]`. Intern legt Polymer für die Eigenschaft keinen eventListener `attributeName-changed` für das Attribut `attributeName` an, somit wird die Eigenschaft, falls das Attribut geändert werden sollte, nicht upgedatet. Beim One-Way-Data-Binding kann dabei zwischen 2 Möglichkeiten unterschieden werden.

Host to Child Das Transportieren der Daten erfolgt nur von Host-Element zum Kind-Element. Hierzu muss zusätzlich der `readOnly`-Parameter (siehe Abschnitt 4.1.3) nicht definiert oder mit `false` initialisiert werden.

Child to Host Der Transport der Daten erfolgt nur von Kind-Element zu Host-Element. Hierzu müssen die `notify`- und `readOnly`-Parameter mit `true` initialisiert werden.

5.1.2 Two-Way-Data-Binding

Mittels dem Two-Way-Data-Binding, auch "automatic Binding" genannt, können Daten von Host- zu Kind-Element und umgekehrt geschrieben werden. Hierzu ist es zwingend notwendig, den `notify`-Parameter mit `true` zu initialisieren und zusätzlich die doppelte geschweifte Klammer Syntax {{}} zu verwenden. Sobald eine Eigenschaft des Kindes oder des Hosts geändert wird, wird von dem jeweiligen Element ein `propertyName-changed-event` ausgelöst. Wenn nun ein anderes Element an diese Eigenschaft gebunden ist, bekommt es das Event mit und ändert daraufhin den eigenen Wert. Hierdurch ist es möglich, eine API für eine Komponente zu erstellen, welche Daten nach außen sichtbar macht, um sie so zwischen mehreren Komponenten auszutauschen.

5.1.3 Binden von nativen Attributen

Um Werte an von HTML reservierte Attribute, also das `hostAttributes`-Objekt (siehe Anschnitt 3.2) statt an Eigenschaften der Komponente zu binden (was mit der normalen Attribut-Binding-Syntax `=` erreicht wird), muss die hierfür vorgesehene Syntax `=$` verwendet werden. So wird beispielsweise in dem Element `<div class=$"{{myClass}}">` die Eigenschaft `myClass` tatsächlich dem Attribut `class` statt der Eigenschaft zugewiesen. Polymer wandelt hierbei bei Verwendung der `=$` Syntax die Zuweisung in die Anweisung `<div>.setAttribute('class', myClass);` um. Das Binden von nativen Attributen ist somit automatisch immer nur in eine Richtung von Host- zu Kind-Element. Im Allgemeinen sollte die Syntax immer dann verwendet werden, wenn die Attribute `style`, `href`, `class`, `for` oder auch `data-*` gesetzt werden sollen.

5.2 Behaviors

Auch wenn in Polymer nur ein beschränktes Vererben mit Hilfe von Type Extensions möglich ist, gibt es dennoch die Möglichkeit Komponenten mit geteilten Code-Modulen, den sogenannten Behaviors, zu erweitern [Polymer - Documentation web]. Sie erlauben es, einen Code in mehrere Komponenten einzubinden, um diese mit einem gewissen Verhalten oder mit zusätzlichen Funktionalitäten auszurüsten. Durch sie haben Entwickler eine gute Kontrolle darüber, welche externen Codes in die eigene Komponente fließen. Im Elemente-Katalog sind sie unter anderem in den Iron-Elementen mit Input-Validierungen oder in den Neon-Elementen mit Animationsverhalten zu finden.

5.2.1 Syntax

Behaviors sind globale Objekte und sollten in einem eigenem Namespace, wie z.B. mit `window.MyBehaviors = window.MyBehaviors || {};`, definiert werden, da die von Polymer intern benutzten Behaviors im Polymer-Objekt verankert sind. Dadurch können Kollisionen mit zukünftigen Behaviors von Polymer verhindert werden. Behaviors haben eine starke Ähnlichkeit zu normalen Polymer-Elementen, sie besitzen ebenso Properties, Listener-Objekte und so weiter. Ein simples Behavior könnte beispielsweise wie in Listing 5.1 dargestellt definiert werden [Polymer - Pressed Behavior web].

```
1 MyBehaviors.HelloBehavior = {
2   properties: { ... },
3   listeners: {
4     mousedown: '_sayHello',
5   },
6   _sayHello: function() {
7     alert('Hallo von der anderen Seite!');
8   }
9 }
```

Listing 5.1: Definition eines Behaviors

Dieses Behavior würde, falls es von einer Polymer Komponente benutzt würde, einen Alert auslösen, wenn die Komponente geklickt wird. Um das Behavior nun einer Komponente hinzuzufügen, muss es dem Behaviors-Array hinzugefügt werden `Polymer({ is: 'super-element', behaviors: [HelloBehavior] });`. In diesem Array können so viele Behaviors mit unterschiedlichen Verhaltensweisen hinzugefügt werden, wie benötigt werden.

5.2.2 Behaviors erweitern

Wie auch Komponenten wiederum andere Komponenten erweitern können, können ebenso Behaviors erweitert werden. Somit können auch Behaviors untereinander geteilte Funktionalitäten einbinden. Um ein Behavior zu erweitern, müssen zunächst alle Behaviors importiert werden, welche das neue Behavior beinhalten soll. Anschließend wird das Verhalten des neuen Behaviors implementiert. Um das neue Behavior nun tatsächlich mit den anderen Behaviors zu erweitern, wird es als Array aus den importieren Behaviors und dem neu implementieren Behavior definiert, wie in Listing 5.2 dargestellt.

```
1 <link rel="import" href="oldbehavior.html">
2 <script>
3   NewBehavior = { ... }
4   NewBehavior = [ OldBehavior, NewBehavior ]
5 </script>
```

Listing 5.2: Erweiterung eines Behaviors

5.3 Events

Mit Polymer erstellte Komponenten können mit Hilfe des `listeners` Objekts auf bestimmte Interaktionen mit der Komponente reagieren [citeulike:13915080]. Hierzu wird die entsprechende Aktion mit der auszuführenden callBack Funktion in dem Objekt angegeben. Wird nun innerhalb der Komponente das angegebene Event registriert, wird die angegebene Funktion ausgeführt. Zu diesen Events zählen beispielsweise die Gesture Events, wie sie in Abschnitt 6.2 beschrieben werden. Um nun verschiedene Aktionen mit unterschiedlichen Kind Elementen zu verknüpfen, können dem `listeners` Objekt auch Kind Elemente und dessen callBack Funktion mit der Syntax `nodeID.eventName: callBack()` hinzugefügt werden. Dadurch reagiert nur das angegebene Element auf das Event, statt der gesamten Komponente. So kann beim "Tap" auf ein Element mit der ID `nodeID` die Funktion (der Event Handler) `callBack()` aufgerufen werden.

5.3.1 Deklarative Events

Alternativ zur imperativen Definition von Events mittels dem `listeners` Objekt, können diese auch deklarativ im Markup des Elements im lokalen DOM der Komponente angegeben werden. Will ein Element auf ein Event reagieren, so muss es dazu die `on-eventName` wie z.B. `<button on-tap="handleTap">Tap Button</button>` Annotation benutzen, wodurch bei Tap auf den Button die Funktion `handleTap` ausgeführt wird. Dies vermischt zwar innerhalb des Templates HTML und Javascript, jedoch muss mit dieser Methode dem Element keine ID zugewiesen werden, nur um dessen Event zu binden.

5.3.2 Selbst definierte Events

Falls das gewünschte Event nicht existieren sollte, können von Host Elementen und dessen Kind Elementen auch selbst definierte Events ausgelöst werden. Hierfür muss von der Komponente die Hilfsfunktion `fire(eventName, data);` aufgerufen werden, was wiederum von anderen Events oder programmatisch ausgelöst werden kann. Der Name des Events `eventName` muss dabei als String angegeben werden und wird, sobald die `fire`-Funktion aufgerufen wird, an alle auf das `fire`-Event hörende Elemente propagiert.

6 Best Practices beim Arbeiten mit Polymer

In den vorherigen Kapiteln wurde gezeigt, welchen Mehrwert Polymer für die Entwickler bringt und wie die Library eingesetzt wird. In diesem Kapitel werden in Abschnitt 6.1 einige UI-Performance-Patterns für eine möglichst performante Applikation sowie in Abschnitt 6.2 das Gesture-System erläutert. Abschnitt 6.3 erläutert die von Polymer bereitgestellten Mittel zum Entwickeln einer möglichst barrierefreien Applikation.

6.1 UI-Performance-Patterns

Applikationen und Webseiten in dem modernen Web müssen möglichst schnell Laden sowie ein flüssiges Arbeiten ermöglichen, damit sie von den Benutzern akzeptiert und genutzt werden. In Abbildung 6.1 wird deutlich, dass Applikationen mit einer Ladezeit von ca. 1000 Millisekunden oder mehr sich negativ auf die Zufriedenheit der Nutzer auswirken [Grigorik 2013].

Delay	User perception
0–100 ms	Instant
100–300 ms	Small perceptible delay
300–1000 ms	Machine is working
1,000+ ms	Likely mental context switch
10,000+ ms	Task is abandoned

Abbildung 6.1: Wahrnehmungen der Benutzer bei unterschiedlichen Ladezeiten

6.1.1 Ladezeiten und initiales Rendern optimieren

Ohne ein besonderes Augenmerk auf die Performance zu werfen, bleibt die Seite oder die Applikation beim initialen Laden mitunter zunächst weiß und der Benutzer kann keine Inhalte sehen. Wenn alle Ressourcen fertig geladen sind, wird der Inhalt dann plötzlich und unerwartet eingeblendet. Dies wird durch blockierende Elemente verursacht. Das erste blockierende Element sind bereits die webcomponents.js Polyfills, da diese in einem `<script>`-Tag stehen und somit sequenziell geladen werden müssen und die Ladezeit verlängern. Dies kann jedoch verhindert werden, indem dem Tag das `async`-Attribut hinzugefügt wird `<script src="webcomponents.js" async></script>`, alle anderen Ressourcen können dann asynchron geladen werden und die Applikation muss nicht auf die Polyfills warten. Um die Ladezeit weiter zu verkürzen, kann mittels überprüft werden, ob die Polyfills optional geladen werden müssen (Lazy Load) (siehe Listing 6.1).

```
1 var webComponentsSupported = ('registerElement' in document
2    && 'import' in document.createElement('link')
3    && 'content' in document.createElement('template'));
```

Listing 6.1: Browser-Feature-Detection

Neben den Polyfills blockieren importierte HTML-Dateien, also andere Komponenten, zwar nicht das Laden der Applikation, jedoch dessen Rendern, da diese selbst wiederum sowohl <script>-Tags als auch Stylesheets beinhalten. Diese blockieren ebenso das Rendern der Webseite, da der Browser die importieren CSS-Dateien erst parsen muss. Hier kann ebenfalls das async-Attribut gesetzt werden um dieses Problem zu lösen. Somit wird die Webseite sofort nach dem Laden aller Ressourcen gerendert, jedoch auch ohne Anwendung der Style-Regeln. Der hierdurch entstehende FOUC muss also manuell verhindert werden, indem grobe Container in einem <style>-Tag für das ungefähre Aussehen der Applikation definiert werden. Dadurch werden die Komponenten nach und nach in die Container geladen, es wird somit sofort eine Vorschau der Applikation angezeigt und die Applikation schnell geladen [Lewis 2015].

6.1.2 Optimierungen für eine flüssige Applikation

Wie in Abschnitt 4.2.1 gezeigt, implementiert Polymer eine eigene Version des Shadow DOMs, den Shady DOM, um eine möglichst breite Browserunterstützung zu gewährleisten. Im Gegensatz zum Shadow DOM, welcher im Kontext der Browser in der Programmiersprache C++ programmiert ist, ist der Shady DOM nur eine in JavaScript programmierte Nachbildung dessen. Somit ist dieser schon auf Plattform-Ebene langsamer. Zusätzlich ist das Scoping sowie das Rendern im nativen Shadow DOM einiges schneller. Vor dem Einbinden der Polymer-Library sollte im globalen Polymer-Settings-Objekt die Property dom auf den Wert 'shadow' gesetzt werden. Polymer überprüft daraufhin, ob der Browser das Shadow DOM unterstützt.

Um eine schnelle Reaktionszeit der App zu gewährleisten, sollten statt Click-Events immer die Touch-Events verwendet werden (siehe Abschnitt 6.2), da Click-Events auf mobilen Geräten eine Zeitverzögerung von 300 Millisekunden haben, bevor diese tatsächlich abgefeuert werden. Dies wird von den Browserherstellern so implementiert, da die Browser bei Click-Events zu erst prüfen müssen, ob das Event Teil eines Doppel-Taps ist, was einen Zoom auslösen würde.

Darüber hinaus sollten nicht benötigte Elemente nur dann tatsächlich gerendert werden, wenn diese auch wirklich im Viewport des Browsers zu sehen sind (Lazy Render). Wird dies nicht gemacht, werden alle Elemente beim initialen Laden der Seite gerendert, was die Applikation bei mehreren Tausend Elementen sehr träge machen kann. So setzt beispielsweise die <iron-list>-Komponente von Polymer diese Technik um, indem es nur die Elemente in das DOM lädt, welche aktuell sichtbar sein müssen. Ebenso wird das lokale DOM des dom-if-Template <template is='dom-if' if='{{open}}'> nur

dann generiert und gerendert, wenn das Element aktiv ist, also wenn `open` den Wert true annimmt. Dies kann unter Anderem bei Dropdown Elementen oder Akkordeon-Menüs umgesetzt werden. Um zu prüfen, wie viele Polymer Komponenten aktuell auf der Seite im DOM stehen und Informationen über dessen Performance zu erhalten, stellt Polymer die Erweiterung "Polymer DevTools Extension" für den Chrome-Browser zur Verfügung [Polymer - DevTools Extension web].

6.2 Gesture-System

Wie bereits in dem vorangegangenen Abschnitt 6.1 angeschnitten wird, sind flüssige Reaktionen auf Aktionen der Benutzer sehr wichtig für gute Applikationen. Um den Umgang mit Input auf verschiedenen Devices zu vereinfachen, stellt Polymer das Gesture-System bereit [Polymer - Gestures web]. Dieses vereinheitlicht die diversen Eingabemöglichkeiten von Desktops, Smartphones bzw. Tablets und Uhren, welche sich mit Maus und Touch unterschiedlich verhalten. Statt dass Entwickler beide Eingabearten gesondert implementieren müssen, kümmert sich Polymers Gesture-System automatisch darum, indem es die Events von Maus und Touch in 4 einheitliche, für alle Plattformen gleiche, vereint. Hierbei handelt es sich um die Events `down`, `up`, `tap` und `track`, welche konsistent auf Touch- und Klick-Umgebungen ausgelöst werden und statt den spezifischen Klick- und Touch-Event-Gegenstücken benutzt werden sollten. Um ein Element auf eines der Touch-Events hören zu lassen, kann entweder die imperative oder die deklarative Methode zum Hinzufügen von Events gewählt werden (siehe Abschnitt 5.3). Die Events können dem Polymer-Prototyp in dem `listeners`-Objekt hinzugefügt werden, wie anhand des Beispiels im folgenden Abschnitt unter **??** verdeutlicht wird.

6.2.1 Down und Up

Die `down`- und `up`-Events werden ausgelöst, wenn der Finger oder die Maus auf das Element drückt oder es loslässt. Sie bilden die einfachsten Gesten des Gesture-Systems, sind aber bei den meisten Anforderungen ausreichend. Diese Events können beispielsweise dazu eingesetzt werden, um zu visualisieren, welches Element gerade berührt oder geklickt wird. Ohne die `down`- und `up`-Events müssten die 4 nativen Events `touchstart`, `touchend`, `mousedown` und `mouseup` implementiert werden. Dabei hören die Touch-Events nur auf das berührte Element, die Maus-Events ändern ihr Ziel beim Bewegen des Mauszeigers, weshalb auf das gesamte Dokument gehört werden muss um auf das `mouseup`-Event zu warten, was besonders beim Scrollen zu Komplikationen führen kann. Mit Polymer sind hierfür nur die beiden Events `down` und `up` notwendig, wie in Listing 6.2 dargestellt.

```
1 Polymer({
2   is: 'my-element',
3   listeners: {
4     'down': 'startFunction',
5     'up': 'endFunction'
6   },
7   startFunction: function() { ... },
8   endFunction: function() { ... }
9 });
```

Listing 6.2: Polymer Events `down` und `up`

6.2.2 Tap

Das `tap`-Event verbindet die `down`- und `up`-Events und stellt ein Event für das Auswählen und Drücken eines Elements auf allen Devices dar. Es funktioniert dabei gleich für Maus und Touch, wobei sich das Gesture-System intern um die Plattform-Unterschiede kümmert. Das `tap`-Event ist das am meisten benutzte Event, da die häufigsten Aktionen der Benutzer der `tap` bzw. der `click` sind.

6.2.3 Track

Das `track`-Event ist eine Erweiterung des `tap`-Events und wird ausgelöst, wenn der Finger oder die Maus beim Drücken eines Elements bewegt wird. Es kommt bei allen Aktionen zum Einsatz, bei denen Drag'n'Drop benötigt wird. Drag'n'Drop kann bei der Maus auf native Weise mit der HTML5-API realisiert werden, jedoch haben viele Plattformen wie Smartphones oder Uhren Touch-Events, wofür die API nicht ausgelegt ist. Dadurch wird das gewünschte Verhalten nicht immer erreicht und ist nur kompliziert manuell zu implementieren. Das `track`-Event soll hier Abhilfe schaffen und das Drag'n'Drop einfacher relaisierbar machen, wobei auf einige Punkte geachtet werden muss.

Werden Elemente mit dem `track`-Event-Listener ausgestattet, so verhindern diese standardmäßig das Scrollen, da bei Touch-Geräten zwischen Scrollen und Draggen unterschieden werden muss. Beim Initialisieren des Elements muss anschließend die Scroll-Funktion mittels `this.setScrollDirection(direction, node);` wiederhergestellt werden. Wird für den Parameter `direction` der wert `'y'` angegeben, so kann über dem Element auf der vertikalen Achse gescrollt werden, wobei das Draggen des Elements in horizontaler Achse möglich ist. Für den `node`-Parameter wird hier standardmäßig `this` übergeben, also das auf `track` zu überwachende Element. Wird das `track` -Event registriert, so kann es auf 3 verschiedene Status aufgeschlüsselt werden: `start track` und `end`. Für jeden Status kann dann das entsprechend gewünschte Verhalten definiert werden.

6.3 A11y - Barrierefreiheit in Polymer

> The Web is an increasingly important resource in many aspects of life: education, employment, government, commerce, health care, recreation, and more. It is essential that the Web be accessible in order to provide equal access and equal opportunity to people with disabilities. An accessible Web can also help people with disabilities more actively participate in society. [W3C web]

Barrierefreiheit ist besonders im öffentlichen Sektor sehr wichtig, weshalb sie auch bei Polymer eine große Rolle spielt, so werden die gesamten UI-Elemente, die Paper Elements, sukzessive barrierefrei neu implementiert. Auch bieten die Iron-Elemente Behaviors für die Barrierefreiheit an (die Accessibility-Behaviors), welche von Custom Elements importiert werden können und das Programmieren von barrierefreien Elementen vereinfachen. Wenn Barrierefreiheit komplett unabhängig von Behaviors erreicht werden soll, bietet sich das Polymer `hostAttributes`-Objekt an (siehe Abschnitt 4.1.6). Um die Applikation auf verschiedene Faktoren, welche nachfolgend erläutert werden [Sutton 2015], bezüglich Barrierefreiheit zu überprüfen, bietet Polymer das Plugin "Accessibility Developer Tools" für den Chrome-Browser an. Dieses weist auf verschiedene Probleme bezüglich der Barrierefreiheit einer Webseite hin, bietet einen erweiterten Inspector für Accessibility-Eigenschaften von Elementen an.

6.3.1 Fokus / Tastatur

Der erste Punkt, der eine barrierefreie Applikation auszeichnet, ist deren Bedienbarkeit mit der Tastatur. So müssen alle Elemente, die eine Interaktion erlauben, mit der Tastatur mittels der Tab- oder den Pfeil-Tasten erreichbar sein. Dies muss gewährleistet werden, falls die Benutzer nicht in der Lage sind eine Maus zu benutzen, oder falls sie nur eine Tastatur benutzen wollen. Um dies zu erreichen, sollte das `tabindex`-Attribut deklarativ für die Elemente definiert werden. Für das Attribut sollte im lokalen DOM einer Komponente allerdings nur der Wert `0` verwendet werden, da nicht vorhersehbar ist, in welcher Reihenfolge das Element auf der einbindenden Webseite zum Einsatz kommt. Alternativ zu der deklarativen Definition des `tabindex`-Attributs kann es auch in Polymers `hostAttributes`-Objekt definiert werden (siehe Abschnitt 4.1.6).

Nachdem gewährleistet wird, dass das Element mit der Tastatur erreichbar ist, muss dies visuell dargestellt werden. Hierfür kann in den Styles des Elements die Pseudoklasse `:focus` definiert werden, oder dem Element das Behavior `PaperInkyFocusBehavior` dem Element hinzugefügt werden. Muss das Element nun eine Interaktion mit einer bestimmten Taste bereitstellen, so bietet Polymer hierfür das Iron Element `iron-a11y-keys-behavior` an, welches ein Mixin für Tastatur-Interaktionen darstellt. Es abstrahiert mit einer einfachen Syntax die Unterschiede der Implementierungen von Tastatureingaben der verschiedenen Browser.

6.3.2 Semantik

Der zweite Punkt ist der korrekte Einsatz der HTML-Elemente innerhalb der eigenen Komponenten. So bietet HTML über 100 Elemente an, mit denen der Inhalt semantisch strukturiert werden kann. Benutzer mit eingeschränktem Sehvermögen benutzen häufig unterstützende Technologien wie Screenreader, welche Benutzern den Inhalt der Webseite vorlesen. Wird die Webseite nun aus nur 2 unterschiedlichen HTML-Tags aufgebaut, z.B. dem <`div`>- oder <`span`>-Tag, kann der Benutzer sich nicht in der Webseite orientieren. Die Screenreader oder andere Assistive Technologies (AT) legen hierbei für jedes Element einen Accessibility-Node im Accessibility-Tree an [W3C 2014c]. Dieser Knoten kann die Accessible Rich Internet Applications (ARIA)-Attribute `role`, `value`, `state` und `properties` haben, welche von Custom Elements deklarativ definiert werden sollten. Die entsprechenden Werte werden für die nativen HTML-Elemente vom W3C definiert. Ebenso wie der `tabindex` können die ARIA-Attribute im `hostAttributes`-Objekt definiert werden, wie im Beispiel einer barrierefreien Komponente in Zeile 16. Wenn der Screenreader ein ihm unbekanntes Element findet und diese Attribute nicht definiert sind, wird standardmäßig die Rolle `group` angenommen, beim Vorlesen des Elements bekommt der Benutzer also nur `group` zu hören und weiß somit nicht, was das Element semantisch darstellt. Falls das Custom Element keinen Text beinhalten, keine sichtbare Beschreibung hat oder nur als Interaktions-Mittel bezüglich eines anderen Inhaltes dienen sollte, wie z.b. eine Checkbox oder ein Input-Feld, ist es wichtig, die Attribute `aria-label` oder `aria-labelledby` zu definieren, welche jenen Inhalt definiert, wie Beispiel einer barrierefreien Komponente in Zeile 9 dargestellt.

6.3.3 Flexibles UI

Der letzte Punkt ist das Erstellen einer UI, welche sich flexibel an die Bedürfnisse der Benutzer anpassen lässt, oder für Benutzer mit einer Farbschwäche angepasst ist. So sollten Farben nicht als einziges Medium zum Übertragen von Informationen dienen, stattdessen sollte eine weitere farbunabhängige Darstellung gewählt werden, wie beispielsweise ein Hinweistext bei einem falsch ausgefüllten Input-Feld. Abgesehen von Farbtönen können Benutzer auch Probleme mit Farbkontrasten haben, welcher zur Darstellung der Information und dem Hintergrund immer ausreichend groß gewählt werden sollte.

> Contrast (Minimum): The visual presentation of text and images of text has a contrast ratio of at least 4.5:1‘ [W3C 2008]

Darüber hinaus müssen Benutzer mit einer schwachen Sehstärke die Webseite unter Umständen vergrößern können, um sie lesen zu können. Es sollte deshalb immer gewährleistet werden, dass die Seite auch bei einem Zoom-Faktor größer 100 korrekt dargestellt wird.

7 Komponenten-Entwicklung

Die in den vorangegangenen Kapiteln dargestellte Methoden zum Erstellen einer Komponente mit Hilfe von Polymer werden in diesem Kapitel in einer Beispielimplementierung umgesetzt. Der Abschnitt 7.1 geht dabei auf die Entwicklung der Komponente selbst ein, in Abschnitt 7.2 wird diese mit einer ähnlichen Implementierung der Komponente mit Hilfe von AngularJS verglichen.

7.1 Entwicklung und Deployment einer Polymer-Komponente

Als Beispielimplementierung einer Polymer-Komponente wird eine responsive Single Page Application (SPA)-Navigation mit erster und zweiter Navigationsebene implementiert. Dabei soll sich die zweite Navigationsebene an die Auswahl der ersten Navigationsebene, sowie der Inhalt der Seite an die Auswahl der zweiten Navigationsebene anpassen.

7.1.1 Entwicklungsumgebung

Für die Entwicklung muss zunächst eine Entwicklungsumgebung eingerichtet werden. Hierzu wird ein Windows-System mit installiertem Node.js gewählt. Für Node werden zusätzlich die Pakete "Grunt", einem automatisierungs-Tool, das Package-Management-Tool "Bower", sowie das Scaffolding-Tool "Yeoman" installiert. Als Editor wird Sublime Text in Version 3 verwendet. Zum Testen der Komponente wird Chrome 48 eingesetzt.

7.1.2 Yeoman

Yeoman [Yeoman web] ist ein Node-Command Line Interface (CLI)-Modul, welches Entwicklern das Erstellen neuer Projekte erleichtert, indem es die dafür benötigten Strukturen und Dateien per Kommandozeile generiert. Es stellt hierfür 3 Arten von Tools zur Verfügung, ein Scaffolding-Tool (`yo`), ein Build-Tool (`Grunt`, `Gulp`, etc.) und ein Package-Manager (`Bower` und `npm`). Hierfür wird von Polymer wird ein entsprechender Generator angeboten, welcher das Polymer-Seed-Element-Boilerplate generiert. Mit dem Befehl `yo polymer:seed my-element` wird der Generator für das `my-element`-Element gestartet und die HTML-Datei mit der entsprechenden Polymer-Grundstruktur generiert (siehe Anhang B).

7.1.3 Die Multi-Navigation-App-Komponente

Zunächst werden die von der Komponente benötigten HTML Imports für die Paper Element `paper-tabs`, `paper-toolbar`, `iron-pages`, `paper-drawer-panel`, `paper-button`, `paper-icon-button` und `paper-card` sowie das Iron Element `iron-icons` mittels Bower

heruntergeladen und in der Komponente importiert. Dabei kann der Import für das `polymer`-Element entfernt werden, da dieser schon in den importierten Elementen vorhanden ist. Nun kann mit dem Aufbau des Layouts der Applikation mittels der `paper`-Elemente begonnen werden.

Als Wrapper für die gesamte Applikation dient das `paper-drawer-panel`-Element, welches die komplette Applikation in einen `drawer`-Bereich, einer Marginal-Spalte auf der linken Seite, sowie einem `main`-Bereich, dem Inhalts-Bereich aufteilt. Dem `main`-Bereich wird nun die erste Navigations-Ebene in Form einer `paper-tabs`-Komponente, welche in einer `paper-toolbar` geschachtelt wird, hinzugefügt. Als Inhalt der `paper-tabs` dient ein <`content`>-Tag, welcher alle <`paper-tab`>-Elemente des Light DOM selektiert und diese in die Komponente projiziert. Somit können die Navigationspunkte der ersten Ebene von außerhalb bestimmt und in die Komponente injiziert werden. Diese Navigations-Punkte werden anschließend mit einem `iron-pages`-Element verbunden, welches die mittels eines <`content`>-Tags ausgewählten Inhalte des Light DOM mit der CSS-Klasse `main` in die Komponente injiziert und zwischen diesen umschalten kann. In den `drawer`-Bereich wir nun, aus optischen Gründen, ebenfalls ein `paper-toolbar`-Element hinzugefügt. Ebenso wie der `drawer`-Bereich enthält auch der `main`-Bereich ein `iron-pages`-Element, welches als Inhalt einen <`content`>-Tag enthält, der alle <`iron-selector`>-Elemente des Light DOM selektiert und in die Komponente projiziert. Die Kinder des <`iron-selector`> bilden dabei die zweite Navigations-Ebene. Diese können jegliche Art von `paper`-Elementen oder nativen HTML-Elementen sein, zwischen denen hin- und hergewechselt werden können soll. Somit kann der Inhalt der gesamten Applikation dynamisch in Form von Kind-Elementen der Komponente übergeben werden. Dabei muss die Struktur des Light DOM jedoch einige Regeln befolgen. Die Elemente der ersten Navigations-Ebene (in der oberen Navigation) müssen `paper-tab`-Elemente sein, die Anzahl dieser ist hingegen variabel. Die Elemente der zweiten Navigations-Ebene müssen der Anzahl an Inhalts-Seiten entsprechen und nach Anzahl der Elemente der ersten Navigations-Ebene mittels `iron-selector`-Elementen gegliedert werden. Welche Elemente die Navigationspunkte selbst sind, spielt dabei keine Rolle. Der Inhalt selbst kann mit beliebigen Elementen definiert werden, jedoch muss für jedes Element die Klasse `main` definiert werden.

Damit die Applikation das gewünschte Verhalten aufzeigt, wurden die Properties `selectedTop` (das ausgewählte Element der ersten Navigations-Ebene) und `selectedContent` (das ausgewählte Element der zweiten Navigations-Ebene) definiert. Ebenso wurde der `ready`-Callback definiert, welcher beim abgeschlossenen Laden der Komponente die Hilfsfunktionen `_countLinks` und `_addSelectedHandler` ausführt. Die `_countLinks`-Methode ermöglicht es, zwischen den übergebenen Links der zweiten Navigations-Ebene zu wechseln, sodass immer der korrekte Inhalt angezeigt wird, wobei die `_addSelectedHandler`-Funktion dieses Verhalten auf die erste Navigations-Ebene abbildet. Sie bedient sich dabei des "Automatic Node Findings", um das jeweils aktive Element der Navigati-

onsebenen zu ermitteln. Abschließend werden in dem Template noch einige Style-Regeln definiert, welche die Applikation auf mobilen Geräten funktionsfähig machen und die Oberfläche optimieren. Die fertige Polymer-Komponente wird in Abbildung 7.1 dargestellt.

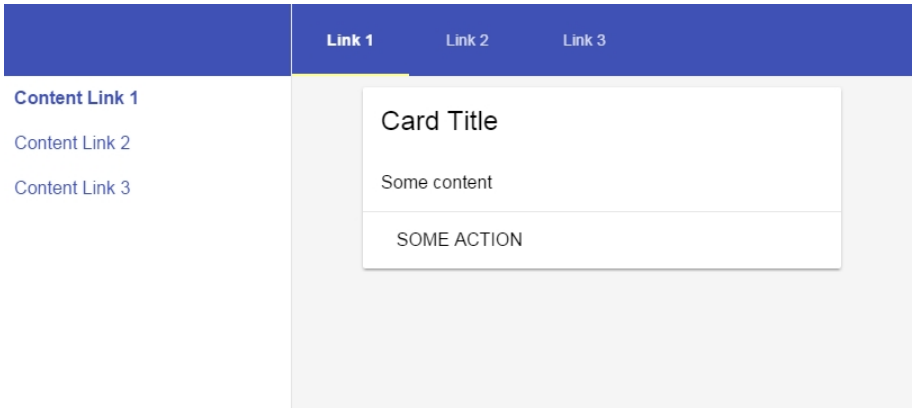

Abbildung 7.1: Darstellung der multi-navigation-app-Komponente

Die Komponente steht unter Versionskontrolle und ist auf GitHub unter der Adresse `https://github.com/glur4k/multi-navigation-app` sowie im Anhang B zu finden. Sie kann nun mittels dem Tag <`multi-navigation-app`> auf allen Webseiten und Applikationen eingebunden werden.

7.1.4 Deployment mit Bower

Um die Komponente für andere Entwickler verfügbar zu machen, damit diese sie einfach mit `bower install multi-navigation-app` installieren können, muss sie bei Bower registriert werden. Hierfür muss zunächst die `bower.json`-Datei um die korrekten Informationen ergänzt werden, da die Komponente sonst nicht von Bower indiziert werden kann. Entspricht die `bower.json` den Bower-Spezifikationen und wird auf GitHub gepusht, so wird die Komponente mittels dem Befehl `bower register multi-navigation-app git://github.com/glur4k/multi-navigation-app.git` bei Bower registriert. Sie ist somit unter der Adresse `http://bower.io/search/?q=multi-navigation-app` auf der Bower-Webseite zu finden. Mittels `bower info multi-navigation-app` kann die aktuelle Version der Komponente auf Bower ermittelt werden, welche dabei immer das aktuellste Release auf GitHub ist. Die Komponente kann nun von allen Entwicklern mittels Bower heruntergeladen und in die eigene Webseite oder Applikation eingebunden werden. Die `bower.json`-Datei ist zusätzlich um den Eintrag `"keywords": [ "web-components" ]` erweitert, somit ist sie ebenso in dem customelements.io-Katalog unter der Adresse `https://customelements.io/glur4k/multi-navigation-app/` zu finden.

7.2 Vergleich mit Komponenten-Entwicklung in AngularJS

Die mit Polymer in Abschnitt 7.1 implementierte Multi-Navigation-Application-Komponente wird möglichst ähnlich mit AngularJS nachgebaut (siehe Anhang C und auf GitHub unter `https://github.com/glur4k/angular-multi-navigation-app`). In diesem Abschnitt werden die durch die beiden unterschiedlichen Implementierungen deutlich gewordenen Unterschiede zwischen Polymer und AngularJS dargestellt.

7.2.1 Einstieg in AngularJS

AngularJS ist ein ebenfalls von Google entwickeltes, clientseitiges Open-Source-JavaScript-Framework zur Erstellung von dynamischen SPAs. Ebenso wie Polymer erlaubt sie es, eigene HTML-Elemente, unter AngularJS `Directives` genannt, zu erstellen, welche die native Sammlung an HTML-Elementen erweitern können. Sie ist ein "Fat-Client-Framework", welches die gesamte Logik, sowie die View-Schicht auf dem Client hält und an ein serverseitiges, Daten-haltendes Model angebunden werden kann [Tarasiewicz 2014]. AngularJS verfolgt dabei den Model-View-ViewModel (MVVM)-Ansatz, wie in Abbildung 7.2 dargestellt wird. Das MVVM-Model stellt eine Erweiterung des MVC-Ansatzes dar, wobei die Controller-Schicht durch eine ViewModel-Schicht ersetzt wird. Diese kann als eine Art Proxy aufgefasst werden, welche der View-Schicht nur die Daten des Models liefert, die sie tatsächlich benötigt. Hierbei kann die ViewModel-Schicht die Daten transformieren, damit sie von der View-Schicht ausgegeben werden können. Ebenso stellt sie die von der View-Schicht benötigten Funktionalitäten zum Ändern der Daten bereit.

Zu den Kern-Konzepten von AngularJS gehören unter anderen das aus Polymer bekannte Two-Way-Data-Binding, die Controller sowie Direktiven. Das Two-Way-Data-Binding erlaubt die Datenbindung in 2 Richtungen, von Model zu View und umgekehrt. Die dabei entstehenden Änderungen am Model werden automatisch im DOM abgebildet und Benutzerinteraktionen innerhalb des Views werden automatisch auf das Model angewendet. Dadurch entfällt die manuelle Manipulation des DOMs mithilfe von JavaScript bzw. jQuery, da diese von AngularJS intern mittels der jQuery-lite, einer vereinfachten, leichteren Version von jQuery automatisch vollzogen wird. Die Controllers definieren die Daten und die Logik (ViewModel), die für einen bestimmten View benötigt werden. Sie halten hierfür einen Scope, in dem die Variablen und Funktionen definiert werden, auf die aus dem View heraus zugegriffen werden können soll. Soll mit Hilfe von AngularJS HTML um eigene Elemente oder Attribute erweitert werden, so werden hierfür die Direktiven eingesetzt, welche die dafür notwendige Logik kapseln.

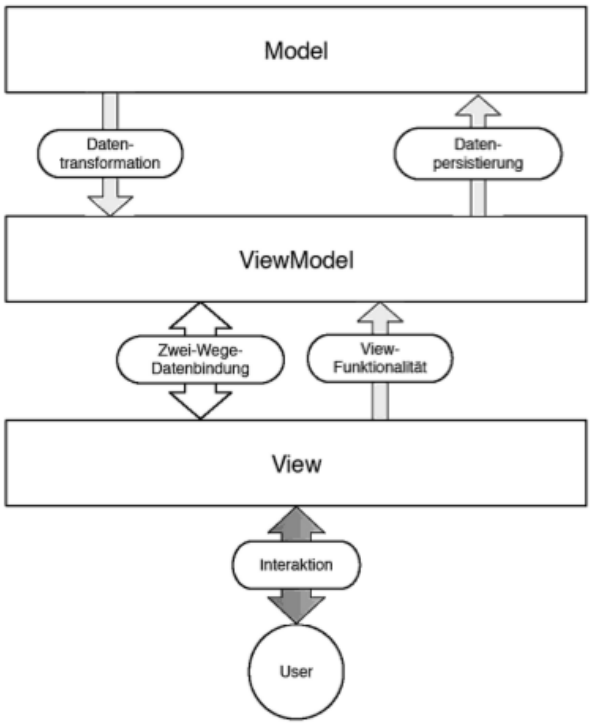

Abbildung 7.2: Darstellung der Model-View-ViewModel-Architektur von AngularJS

7.2.2 Vergleich der Intentionen hinter Polymer und AngularJS

Die Web Components sind eine Sammlung an sich noch in Entwicklung befindenden Technologien und APIs um eigene HTML-Elemente zu definieren. Polymer ist eine Library um dies zu vereinfachen und erweitern. Mit Hilfe von Polyfills und zusätzlichen Features kann es HTML-Elemente erstellen und sie auch auf Browsern zum Einsatz bringen, welche die nötigen Standards noch nicht unterstützen. AngularJS hingegen stellt APIs auf Framework-Ebene bereit wie Services, Routing oder auch Serverkommunikation. Die Polymer-Library bildet diese Funktionalitäten hingegen nicht ab. Stattdessen muss hierfür auf die mit Polymer entwickelten Iron Elements des Elemente Katalogs zurückgegriffen werden, da diese solche Funktionalitäten bieten. Es kümmert sich mehr darum, das Entwickeln solcher umfangreichen, mächtigen und wiederverwendbarer Komponenten zu ermöglichen. Mit diesen Komponenten wiederum können komplexe Applikationen realisiert werden, wie sie mit AngularJS umgesetzt werden können. Polymer muss daher als eine Library angesehen werden, wobei AngularJS ein komplettes Framework ist. Die durch Polymer erweiterten Web Components entsprechen vielmehr den Direktiven von Angular. Polymer ist also als ein Subset des AngularJS Funktionsumfangs, welcher für das Entwickeln von komplexen Applikationen entworfen ist, anzusehen.

7.2.3 Vergleich der Technischen Features

Dennoch gibt es Features, die sowohl Polymer als auch AngularJS mitbringen. Hierzu zählt das Two-Way-Data-Binding mit einer sehr ähnlichen Syntax sowie die deklarativen Templates zum Definieren von Komponenten-internen Markups. Jedoch überwiegen die Unterschiede beider Funktionsumfänge.

AngularJS ist ein Framework zum Erstellen von SPAs und stellt hierfür eine Reihe an Funktionalitäten bereit. Polymer hingegen bietet keine internen Mechanismen für das Strukturieren einer Applikation bereit, stattdessen können die hierfür mit Polymer entwickelten Komponenten eingesetzt werden. Eine mit Polymer realisierte Applikation ist dadurch deklarativ geschachtelt und hierarchisch aufgebaut, was durch das Mediator-Pattern erzwungen wird (siehe Abschnitt 5.1). AngularJS hingegen erfordert keine hierarchische Struktur, da es unter dem Framework nicht nur ein Typ von Komponenten gibt. Dies spiegelt sich auch in dessen Aufbau wieder, da die Definition von Elementen mit AngularJS imperativ in JavaScript erfolgt. Da Definition von Komponenten mit Polymer deklarativ orientiert ist, können diese auch einfacher mit anderen Komponenten, welche nicht mit Polymer erstellt werden, interagieren, da sie normales DOM sind. AngularJS hingegen ist nur schwer mit anderen SPA-Frameworks zu kombinieren. Des weiteren nutzt Polymer das Shadow DOM, bzw. das Shady DOM, um Stylesheets und JavaScripte in den Komponenten zu kapseln. In AngularJS wird hierauf verzichtet, da es nur die Daten in dem Scope kapselt. Auch ist das Deployment einer AngularJS-Direktiven einiges umständlicher als mit Polymer, da externe Templates immer relativ zum Projekt oder Domain-Root adressiert werden müssen. Hierzu ist ein zusätzlicher Build-Schritt nötig, der das Template in den Template-Cache schreibt, oder es in die Direktive serialisiert. Zuletzt ist die Browserunterstützung unter AngularJS deutlich breiter - beispielsweise wird der Internet Explorer in Version 8 unterstützt - wodurch es auch bei mehr Projekten in der Produktion eingesetzt werden kann.

8 Zukunftsprognose

Die Standards der Web-Component-Technologien sind noch nicht fertiggestellt und befinden sich momentan noch in Entwicklung. Um allgemeingültige Standards für alle Browserhersteller zu gewährleisten, werden sich diese vermutlich noch verändern. Jedoch werden sie dadurch ein breites Spektrum an Zustimmung und Implementierung bekommen, da sie eine zentrale Rolle spielen, wenn darum geht, auch das Web zu einer Plattform zu machen, für welche die Paradigmen von höheren Programmiersprachen gelten. So soll es auch im Web möglich sein, Applikationen aus mehreren verschiedenen Komponenten, die gekapselt, wartbar und interoperabel sind, zu entwickeln. Polymer wird dabei weiterhin eine wichtige Rolle innehaben, da die Bibliothek das Arbeiten mit den Web Components vereinfacht und zusätzliche Funktionalitäten dafür bereitstellt. Dabei könnte sogar ein Vergleich der Polymer Bibliothek und der jQuery Bibliothek vorstellbar sein. So wie jQuery zu einem De-Facto-Standard für das Arbeiten mit DOM-Elementen wurde, so könnte Polymer ein Standard für das Arbeiten mit Web Components werden. Da Google maßgeblich die Entwicklung der Standards der Web Component Technologien vorantreibt und sich die anderen Browserhersteller teilweise daran orientieren, könnte es sogar sein, dass zumindest Teile von Polymer zum offiziellen Standard der Web Technologien wird.

Wenn die Standards von allen Browsern akzeptiert und implementiert wurden, sinkt auch die Komplexität von Polymer um ein Vielfaches, da die komplette Schicht der Polyfills wegfällt. Neben den Polyfills kann Polymer dann auf die Nutzung des Bibliotheksinternen Shady DOM verzichten und stattdessen mit einer standardisierten, schnellen und leichtgewichtigen Version des Shadow DOM arbeiten. Dadurch wird Polymer um einiges schneller arbeiten und auch auf mobilen Geräten effizienten Einzug erhalten.

Neben Polymer basiert auch Angular ab Version 2.0 auf den Web-Component-Standards und wird dessen Technologien benutzen, jedoch verfolgt Angular einen anderen Ansatz als Polymer. Angular implementiert hierfür eine eigene Schicht, welche auf die nativen Technologien zugreift und das Framework komplexer macht. Da in Zukunft die Polyfills und der Shady DOM der Polymer-Library wegfallen, könnte Angular Polymer in sich integrieren, statt die Web-Components-Standards selbst zu implementieren. Hierfür würde beispielsweise die Micro-Schicht in Frage kommen, da diese nur die Grundfunktionalitäten für den Umgang mit den Web Components leistet. Jedoch werden beide Bibliotheken weiterhin koexistieren und weder Polymer Angular ersetzen noch umgekehrt, da Polymer nur eine erweiterbare Library und Angular ein vollständiges Framework ist. Beide Plattformen verfolgen daher unterschiedliche Ansätze und können unterschiedliche Probleme lösen. Jedoch kann durch die Entwicklung der Carbon-Elemente die Polymer-Library sukzessive als Framework erweitert werden, da diese eine neue Möglichkeit bieten, wie Applikationen strukturiert werden können. Durch sie können komplexe

Applikationen realisiert werden, da die Elemente sehr nah an der Plattform selbst sind, was einige Vorteile mit sich bringt. So haben sie wenige Abhängigkeiten, eine bessere Performance, können leicht ausgetauscht werden und bieten ein vereinfachtes Debugging an, da sie die Tools der Plattform statt die eines Frameworks benutzen. Polymer kann also als eine Library mit einem optionalen Framework-Plugin verstanden werden. Dieses bietet, je nach Anforderungen, Vor- und Nachteile wie jedes andere Framework auch. Es liegt somit weiterhin im Ermessen der Entwickler ob Angular oder Polymer eingesetzt werden soll, oder ob sogar - bis zu einem gewissen Grad - beides verwendet werden soll. Jedoch beschränkt sich Polymer nicht nur auf den Einsatz in Angular, sondern kann mit jedem beliebigem Framework kombiniert werden.

Wie genau die Entwicklung von Polymer weiter verläuft und ob sich diese auf Angular auswirken wird, liegt jedoch ganz im Ermessen von Google.

Auch die Seitenbau GmbH beobachtet die Entwicklung der Web Components mit Interesse, jedoch können sie in mittels Polymer momentan noch nicht produktiv eingesetzt werden. Die Ansprüche der Kunden für die unterstützen Browser sind noch sehr hoch, so soll in vielen Projekten noch der Internet Explorer in Version 8 unterstützt werden, was weder Polymer noch dessen Polyfills gewährleisten. Auch ist die Performance besonders auf mobilen Browsern noch nicht ausreichend, so dass auf alternative Frameworks wie AngularJS zurückgegriffen wird. Sollten diese Rahmenbedienungen in Zukunft jedoch erfüllt werden, so könnte die Polymer-Library auch bei der Seitenbau GmbH einen produktiven Einzug erhalten.

Anhang A - Native Web Component

HTML Struktur zum Benutzen einer Web Component mit den nativen APIs

index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Demo of a Custom Element</title>

  <!-- Import the web component -->
  <link rel="import" href="elements/custom-element.html">
</head>
<body>

  <!-- Use the web component -->
  <custom-element theme="style1">Reader</custom-element>

</body>
</html>
```

Implementierung einer Web Component mit den nativen APIs

custom-element.html:

```
<template id="myElementTemplate">
  <style>
    .outer { ... }
    .style1 { color: green; }
    .style2 { color: blue; }
    .name { font-size: 35pt; padding-top: 0.5em; }
  </style>

  <div class="outer">
    Welcome in the Web Component
    <div class="name">
      <content></content>
    </div>
  </div>
</template>

<script>
  // Refers to the "importee", which is index.html
  var importDoc = document.currentScript.ownerDocument;

  // Creates an object based in the HTML Element prototype
  var CustomElementProto = Object.create(HTMLElement.prototype);
```

```
  // Creates the "theme" attribute and sets a default value
  CustomElementProto.theme = 'style1';

  // Fires when an instance of the element is created
  CustomElementProto.createdCallback = function() {

    // Creates the shadow root
    var shadow = this.createShadowRoot();

    // Gets content from <template>
    var template = importDoc.querySelector('#myElementTemplate').content;

    // Adds a template clone into shadow root
    shadow.appendChild(template.cloneNode(true));

    // Caches .outer DOM query
    this.outer = shadow.querySelector('.outer');

    // Checks if the "theme" attribute has been overwritten
    if (this.hasAttribute('theme')) {
      var theme = this.getAttribute('theme');
      this.setTheme(theme);
    } else {
      this.setTheme(this.theme);
    }
  };

  // Fires when an attribute was added, removed, or updated
  CustomElementProto.attributeChangedCallback = function(attr, oldVal, newVal) {
    if (attr === 'theme') {
      this.setTheme(newVal);
    }
  };

  // Sets new value to "theme" attribute
  CustomElementProto.setTheme = function(val) {
    this.theme = val;
    this.outer.className = "outer " + this.theme;
  };

  // Registers <custom-element> in the main document
  document.registerElement("custom-element", {
    prototype: CustomElementProto
  });
</script>
```

Anhang B - Polymer Web Component

Polymer Seed-Element

my-element.html:

```
<link rel="import" href="../polymer/polymer.html">

<!--
An element providing a solution to no problem in particular.

Example:

    <my-element></my-element>

@group Seed Elements
@element my-element
@demo demo/index.html
@hero hero.svg
-->
<dom-module id="my-element">

  <template>
    <style>
      :host {
        display: block;
        box-sizing: border-box;
      }

      .author img {
        display: block;
        float: left;
        margin-right: 5px;
        max-height: 100px;
        max-width: 100px;
      }
    </style>
    <h1>&lt;my-element&gt;</h1>
    <content></content>
    <p class="author">
      <img src="{{author.image}}">
      Cheers,<br/>
      <span class="name">{{author.name}}</span>
    </p>
  </template>

</dom-module>

<script>

  Polymer({
```

```
is: 'my-element',

properties: {

  /**
   * `fancy` indicates that the element should don a monocle and tophat,
   * while checking its pocket watch.
   */
  fancy: Boolean,

  /**
   * Describes the author of the element, but is really just an excuse to
   * show off JSDoc annotations.
   *
   * @type {{name: string, image: string}}
   */
  author: {
    type: Object,
    // Use `value` to provide a default value for a property, by setting it
    // on your element's prototype.
    //
    // If you provide a function, as we do here, Polymer will call that
    // _per element instance_.
    //
    // We do that to ensure that each element gets its own copy of the
    // value, rather than having it shared across all instances (via the
    // prototype).
    value: function() {
      return {
        name:  'Dimitri Glazkov',
        image: 'http://addyosmani.com/unicorn.jpg',
      };
    }
  },

},

// Element Lifecycle

ready: function() {
  // `ready` is called after all elements have been configured, but
  // propagates bottom-up. This element's children are ready, but parents
  // are not.
  //
  // This is the point where you should make modifications to the DOM (when
  // necessary), or kick off any processes the element wants to perform.
},

attached: function() {
  // `attached` fires once the element and its parents have been inserted
  // into a document.
  //
  // This is a good place to perform any work related to your element's
  // visual state or active behavior (measuring sizes, beginning animations,
```

```
      // loading resources, etc).
    },

    detached: function() {
      // The analog to `attached`, `detached` fires when the element has been
      // removed from a document.
      //
      // Use this to clean up anything you did in `attached`.
    },

    // Element Behavior

    /**
     * The `my-element-lasers` event is fired whenever `fireLasers` is called.
     *
     * @event my-element-lasers
     * @detail {{sound: String}}
     */

    /**
     * Sometimes it's just nice to say hi.
     *
     * @param {string} greeting A positive greeting.
     * @return {string} The full greeting.
     */
    sayHello: function(greeting) {
      var response = greeting || 'Hello World!';
      return 'my-element says, ' + response;
    },

    /**
     * Attempt to destroy this element's enemies with a beam of light!
     *
     * Or, at least, dispatch an event in the vain hope that someone else will
     * do the zapping.
     */
    fireLasers: function() {
      this.fire('my-element-lasers', {sound: 'Pew pew!'});
    }

  });

</script>
```

Implementierung der Polymer Komponente Multi-Navigation-App

multi-navigation-app.html:

```
<link rel="import" href="../paper-tabs/paper-tabs.html">
<link rel="import" href="../paper-toolbar/paper-toolbar.html">
<link rel="import" href="../iron-pages/iron-pages.html">
<link rel="import" href="../paper-drawer-panel/paper-drawer-panel.html">
<link rel="import" href="../paper-button/paper-button.html">
<link rel="import" href="../paper-icon-button/paper-icon-button.html">
<link rel="import" href="../paper-card/paper-card.html">
<link rel="import" href="../iron-icons/iron-icons.html">

<dom-module id="multi-navigation-app">
  <template>
    <style>
      :host {
        display: block;
        box-sizing: border-box;
      }
      paper-tabs {
        color: #fff;
        height: 100%;
        margin: auto -16px;
      }
      #topnavi ::content paper-tab {
        min-width: 50px;
      }
      @media only all and (min-width: 600px) {
        #topnavi ::content paper-tab {
          min-width: 80px;
        }
      }
      #leftnavi ::content paper-button {
        margin: 0;
        width: 100%;
        text-align: left;
        color: #3f51b5;
        text-transform: none;
      }
      #leftnavi ::content paper-button.iron-selected {
        font-weight: bold;
      }
      iron-pages {
        position: relative;
      }
      #paperDrawerPanel [drawer]:not([style-scope]):not(.style-scope) {
        border-right: 1px solid #e0e0e0;
      }
      paper-drawer-panel {
        --paper-drawer-panel-main-container: {
          background-color: #f5f5f5;
```

```
        };
        --paper-drawer-panel-left-drawer-container: {
          border-right: 1px solid #e5e5e5;
        };
      }
      #contents ::content paper-card {
        margin: 0 10%;
        width: 80%;
      }
    </style>

    <paper-drawer-panel>
      <paper-header-panel drawer>
        <paper-toolbar></paper-toolbar>

        <!-- Left Navigation -->
        <iron-pages id="leftnavi">
          <content select="iron-selector"></content>
        </iron-pages>
      </paper-header-panel>

      <paper-header-panel main>
        <paper-toolbar>
          <paper-icon-button icon="menu" paper-drawer-toggle>
          </paper-icon-button>

          <!-- Top Navigation -->
          <paper-tabs id="topnavi" selected="{{selectedTop}}">
            <content select="paper-tab"></content>
          </paper-tabs>
        </paper-toolbar>

        <!-- Main Content -->
        <iron-pages id="contents" selected="{{selectedContent}}">
          <content select=".main"></content>
        </iron-pages>
      </paper-header-panel>
    </paper-drawer-panel>

  </template>
</dom-module>

<script>
  Polymer({
    is: 'multi-navigation-app',

    properties: {
      selectedTop: {
        type: Number,
        value: 0
      },
      selectedContent: {
        type: Number,
        value: 0
```

```
    }
  },

  // Element Behavior
  _countLinks: function() {
    var count = 0;
    this.$.leftnavi.getContentChildren().forEach(function (value, i) {
      var links = value.children;
      for (j = 0; j < links.length; j++) {
        count++;
        links[j].linkNum = (count - 1);
      }
    });
  },

  _getSelectedContent: function() {
    return this.$.leftnavi.getContentChildren()[this.$.topnavi.selected]
      .selectedItem.linkNum;
  },

  _addSelectedHandler: function() {
    var topNaviItems = this.$.topnavi;
    var leftNaviItems = this.$.leftnavi;
    var contents = this.$.contents;
    var that = this;

    topNaviItems.addEventListener('iron-select', function() {
      // init first selected
      leftNaviItems.select(that.selectedTop);
      var leftSelected = leftNaviItems.getContentChildren()[that
        .selectedTop].selectedItem;
      if (typeof leftSelected == 'undefined') {
        leftNaviItems.getContentChildren()[that.selectedTop].select(0)
      }

      contents.select(that._getSelectedContent());
    });

    leftNaviItems.getContentChildren().forEach(function (value, i) {
      value.addEventListener('iron-select', function (e) {
        contents.select(that._getSelectedContent());
      });
    });
  },
});
</script>
```

Anhang C - AngularJS Web Component

Controller der AnngularJS-Applikation angular-multi-navigation-application

main.js:

```
'use strict';

/**
 * @ngdoc function
 * @name multiNavigationApp.controller:MainCtrl
 * @description
 * # MainCtrl
 * Controller of the multiNavigationApp
 */
angular.module('multiNavigationApp')
  .controller('MainCtrl', function ($scope) {
    $scope.sections = [{
      title: 'Nav 1',
      subSections: [{
        title: 'Sub-Nav 1.1',
        content: 'Content 1.1'
      }, {
        title: 'Sub-Nav 1.2',
        content: 'Content 1.2'
      }, {
        title: 'Sub-Nav 1.3',
        content: 'Content 1.3'
      }, {
        title: 'Sub-Nav 1.4',
        content: 'Content 1.4'
      }]
    }, {
      title: 'Nav 2',
      subSections: [{
        title: 'Sub-Nav 2.1',
        content: 'Content 2.1'
      }, {
        title: 'Sub-Nav 2.2',
        content: 'Content 2.2'
      }, {
        title: 'Sub-Nav 2.3',
        content: 'Content 2.3'
      }, {
        title: 'Sub-Nav 2.4',
        content: 'Content 2.4'
      }]
    }, {
      title: 'Nav 3',
      subSections: [{
```

```
        title: 'Sub-Nav 3.1',
        content: 'Content 3.1'
      }, {
        title: 'Sub-Nav 3.2',
        content: 'Content 3.2'
      }, {
        title: 'Sub-Nav 3.3',
        content: 'Content 3.3'
      }, {
        title: 'Sub-Nav 3.4',
        content: 'Content 3.4'
      }]
    }];
  });
```

Direktive der AnngularJS-Applikation angular-multi-navigation-application

multi-navigation.js:

```
'use strict';

/**
 * @ngdoc directive
 * @name multiNavigationApp.directive:multiNavigation
 * @description
 * # multiNavigation
 */
angular.module('multiNavigationApp')
  .directive('multiNavigation', function () {
    return {
      scope: {
        sections: '='
      },
      link: function ($scope) {
        $scope.selectSection = function (section) {
          $scope.selectedSection = section;
          $scope.selectedSubSection = section.lastSelected ||
            section.subSections[0];
        };
        $scope.selectSubSection = function (subSection) {
          $scope.selectedSubSection = subSection;
          $scope.selectedSection.lastSelected = subSection;
        };
        $scope.selectedSection = $scope.sections[0];
        $scope.selectedSubSection = $scope.selectedSection.subSections[0];
      },
      templateUrl: 'views/multi-navigation.html'
    };
  });
```

View des Controllers der AnngularJS-Applikation angular-multi-navigation-application

main.html:

```
<div layout="column" layout-fill>
  <md-toolbar class="md-hue-3">
    <div class="md-toolbar-tools">
      <span>multi-navigation-app angular</span>
      <!-- fill up the space between left and right area -->
      <span flex></span>
    </div>
  </md-toolbar>
  <md-content>
      <multi-navigation sections="sections"></multi-navigation>
  </md-content>
</div>
```

View der Direktive der AnngularJS-Applikation angular-multi-navigation-application

multi-navigation.html:

```
<div layout="row">
  <md-sidenav md-component-id="left" md-is-locked-open="true"
    class="md-sidenav-left">
    <md-toolbar></md-toolbar>
    <md-content layout-padding>
      <md-list ng-if="selectedSection">
        <md-list-item
          ng-repeat="subSection in selectedSection.subSections"
          ng-click="selectSubSection(subSection)"
          ng-class="{'selected' : selectedSubSection === subSection}">
          {{subSection.title}}
        </md-list-item>
      </md-list>
    </md-content>
  </md-sidenav>
  <div flex>
    <md-toolbar>
      <md-tabs md-selected="0" md-border-bottom md-no-pagination>
        <md-tab
          ng-click="selectSection(section)"
          ng-repeat="section in sections" label="{{section.title}}"></md-tab>
      </md-tabs>
    </md-toolbar>
    <md-content layout-padding flex>
      <h1>{{selectedSubSection.title}}</h1>
      {{selectedSubSection.content}}
    </md-content>
  </div>
</div>
```

Abkürzungsverzeichnis

HTML Hypertext Markup Language

W3C World Wide Web Consortium

DOM Document Object Model

CSS Cascading Style Sheets

API Application Programming Interface

FOUC Flash Of Unstyled Content

LIFO Last In First Out

MVC Model View Controller

XSS Cross Site Scripting

CORS Cross Origin Resource Sharing

SOP Same Origin Policy

HTTP Hypertext Transfer Protocol

TCP Transmission Control Protocol

TLS Transport Layer Security

XHR XMLHttpRequest

ARIA Accessible Rich Internet Applications

UI User Interface

AT Assistive Technologies

SPA Single Page Application

MVC Model View Controller

MVVM Model-View-ViewModel

CLI Command Line Interface

HTTP Hypertext Transfer Protocol

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[Bateman 2014] Bateman, C. (2014). A No-Nonsense Guide to Web Components. http://cbateman.com/blog/a-no-nonsense-guide-to-web-components-part-2-practical-use/.

[Bidelman 2013a] Bidelman, E. (2013a). Custom Elements: defining new elements in HTML. http://www.html5rocks.com/en/tutorials/webcomponents/customelements/.

[Bidelman 2013b] Bidelman, E. (2013b). Shadow DOM 301: Advanced Concepts & DOM APIs. http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom-301/.

[Bidelman 2014] Bidelman, E. (2014). Shadow DOM 201: CSS and Styling. http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom-201/.

[Can I Use - Custom Elements web] Can I Use - Custom Elements (web). Custom Elements. http://caniuse.com/#feat=custom-elements.

[Can I Use - HTML Imports web] Can I Use - HTML Imports (web). HTML Imports. http://caniuse.com/#search=imports.

[Can I Use - HTTP/2 web] Can I Use - HTTP/2 (web). Http/2. http://caniuse.com/#search=http.

[Can I Use - Shadow DOM web] Can I Use - Shadow DOM (web). Shadow DOM. http://caniuse.com/#search=shadow%20dom.

[Cooney und Bidelman 2013] Cooney, D. and Bidelman, E. (2013). Shadow DOM 101. http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/.

[customelements.io web] customelements.io (web). customelements.io - Custom Elements. http://customelements.io/.

[Dodson 2014] Dodson, R. (2014). Shadow DOM CSS Cheat Sheet. http://robdodson.me/shadow-dom-css-cheat-sheet/.

[Gasston 2014] Gasston, P. (2014). A Detailed Introduction To Custom Elements. http://www.smashingmagazine.com/2014/03/introduction-to-custom-elements/.

[Glazkov 2015] Glazkov, D. (2015). Custom Elements. http://w3c.github.io/webcomponents/spec/custom/#concepts.

[Grigorik 2013] Grigorik, I. (2013). *High Performance Browser Networking: What every web developer should know about networking and web performance.* O'Reilly Media, 1 edition.

[Hongkiat 2014] Hongkiat (2014). Append And Reuse HTML Docs With HTML Import. http://www.hongkiat.com/blog/html-import/.

[HTML5Rocks 2013] HTML5Rocks (2013). HTML Imports: #include for the web. http://www.html5rocks.com/en/tutorials/webcomponents/imports/.

[Ihrig 2012] Ihrig, C. (2012). The Basics of the Shadow DOM. http://www.sitepoint.com/the-basics-of-the-shadow-dom/.

[Kitamura 2014a] Kitamura, E. (2014a). Introduction to Custom Elements. http://webcomponents.org/articles/introduction-to-custom-elements/.

[Kitamura 2014b] Kitamura, E. (2014b). Introduction to the template elements. http://webcomponents.org/articles/introduction-to-template-element/.

[Kitamura 2015] Kitamura, E. (2015). Introduction to HTML Imports. http://webcomponents.org/articles/introduction-to-html-imports/.

[Kröner 2014a] Kröner, P. (2014a). Das Web der Zukunft. http://webkrauts.de/artikel/2014/das-web-der-zukunft.

[Kröner 2014b] Kröner, P. (2014b). Web Components erklärt, Teil 1: Was sind Web Components? http://www.peterkroener.de/web-components-erklaert-teil-1-was-sind-web-components.

[Lewis 2015] Lewis, P. (2015). Polymer for the Performance-obsessed. https://aerotwist.com/blog/polymer-for-the-performance-obsessed/.

[MDN 2015a] MDN (2015a). Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[MDN 2015b] MDN (2015b). template - HTML. https://developer.mozilla.org/de/docs/Web/HTML/Element/template.

[Microsoft web] Microsoft (web). Developer Resources: Microsoft Edge Dev. https://dev.windows.com/en-us/microsoft-edge/platform/status/?filter=f3f0000bf&search=web%20components.

[Mozilla 2015] Mozilla (2015). The state of Web Components. https://hacks.mozilla.org/2015/06/the-state-of-web-components/.

[Namics 2014] Namics (2014). Web Components: HTML Templates. https://frontend.namics.com/2014/03/20/web-components-html-templates-2/.

[Overson und Strimpel 2015] Overson, J. and Strimpel, J. (2015). *Developing Web Components: UI from jQuery to Polymer*. O'Reilly Media, 1 edition.

[Perterkroener 2014] Perterkroener (2014). Fragen zu HTML5 und Co beantwortet 15 - Web Components versus Performance und Async, CSS-Variablen, Data-URLs. http://www.peterkroener.de/fragen-zu-html5-und-co-beantwortet-15-web-components-performance-css-variablen-data-urls-async/.

[Polymer - Base Documentation web] Polymer - Base Documentation (web). Base Documentation. http://polymer.github.io/polymer/.

[Polymer - DevTools Extension web] Polymer - DevTools Extension (web). DevTools Extension. https://chrome.google.com/webstore/detail/polymer-devtools-extensio/mmpfaamodhhlbadloaibpocmcomledcg.

[Polymer - Documentation web] Polymer - Documentation (web). Documentation. https://www.polymer-project.org/1.0/docs/devguide/feature-overview.html.

[Polymer - Element Catalog web] Polymer - Element Catalog (web). Element Catalog. https://elements.polymer-project.org/.

[Polymer - Gestures web] Polymer - Gestures (web). Gestures. https://github.com/Polymer/polymer-gestures.

[Polymer - Pressed Behavior web] Polymer - Pressed Behavior (web). Pressed Behavior. https://github.com/Polymer/polycasts/blob/master/ep21-behaviors/behaviors-demo/elements/pressed-behavior/pressed-behavior.html.

[Polymer - Shady DOM web] Polymer - Shady DOM (web). What is shady DOM? - Polymer 1.0. https://www.polymer-project.org/1.0/articles/shadydom.html.

[Polymer - Vulcanize web] Polymer - Vulcanize (web). Vulcanize. https://github.com/polymer/vulcanize.

[Satrom 2014] Satrom, B. (2014). *Building Polyfills*. O'Reilly Media, 1 edition.

[Savage 2015] Savage, T. (2015). Polymer Summit Recap. https://blog.polymer-project.org/announcements/2015/09/29/polymer-summit-recap/.

[Schaffranek 2014] Schaffranek, R. (2014). Web Components – eine Einführung. https://blog.selfhtml.org/2014/12/09/web-components-eine-einfuehrung/.

[Sharp 2010] Sharp, R. (2010). What is a Polyfill? https://remysharp.com/2010/10/08/what-is-a-polyfill.

[Sutton 2015] Sutton, M. (2015). Notes On Client-Rendered Accessibility. https://www.smashingmagazine.com/2015/05/client-rendered-accessibility/.

[Tarasiewicz 2014] Tarasiewicz, R. B. P. (2014). *AngularJS*. Dpunkt.Verlag GmbH.

[The Chromium Projects web] The Chromium Projects (web). Achieving data binding nirvana. https://www.chromium.org/developers/polymer-1-0#TOC-Achieving-data-binding-nirvana.

[W3C 2008] W3C (2008). Web Content Accessibility Guidelines (WCAG) 2.0. https://www.w3.org/TR/WCAG20/.

[W3C 2014a] W3C (2014a). Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/.

[W3C 2014b] W3C (2014b). HTML Templates. http://www.w3.org/TR/html5/scripting-1.html#the-template-element.

[W3C 2014c] W3C (2014c). The Accessibility Tree and the DOM Tree. http://www.w3.org/WAI/PF/aria-implementation/#intro_treetypes.

[W3C 2015a] W3C (2015a). CSS Custom Properties for Cascading Variables Module Level 1. http://dev.w3.org/csswg/css-variables/.

[W3C 2015b] W3C (2015b). CSS Scoping Module Level 1. https://drafts.csswg.org/css-scoping/.

[W3C 2015c] W3C (2015c). Shadow DOM. http://www.w3.org/TR/shadow-dom/.

[W3C web] W3C (web). Introduction to Web Accessibility. https://www.w3.org/WAI/intro/accessibility.php.

[W3Techs 2015] W3Techs (2015). Usage Statistics of HTTP/2 for Websites. http://w3techs.com/technologies/details/ce-http2/all/all.

[WebComponents - Polyfills web] WebComponents - Polyfills (web). Polyfills. http://webcomponents.org/polyfills/.

[WebComponents - webcomponents.js web] WebComponents - webcomponents.js (web). webcomponents.js. https://github.com/webcomponents/webcomponentsjs.

[WHATWG web] WHATWG (web). HTML Standard. https://html.spec.whatwg.org/multipage/dom.html#htmlunknownelement.

[Yeoman web] Yeoman (web). Yeoman Website. http://yeoman.io/.