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. Ing. 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. Ing. Andreas Maurer
Abgabedatum:	15.02.2016
Schlagworte:	Web Components, Polymer, AngularJS, JavaScript, HTML, Custom Elements, HTML Templates, Shadow DOM, HTML Imports

[Text der Zusammenfassung etwa 150 Worte. Es soll der Lösungsweg beschrieben sein.]

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

bei der Seitenbau GmbH unter Anleitung von Prof. Dr. Marko Boger 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 HTML zu erweitern. Dies ist bisher nicht möglich, da die HTML-Technologie - und somit die Möglichkeiten, HTML-Tags zu benutzen - vom W3C definiert und standardisiert wird. Unter den wichtigsten der neuen Standards sind die folgenden vier Technologien aufzuführen: Custom Elements, Shadow 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, CSS und JavaScript kapselt. HTML Templates stellen, wie der Name impliziert, einen Template-Mechanismus für HTML bereit und 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 API für Entwickler komfortabler gemacht werden. Um nun 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 Performanz 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 beschrieben. 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 3-6 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 auf die Problemlösungen der Web Components nach den Vorstellungen des W3C eingegangen. In Abschnitt 2.2 wird die erste Technologie vorgestellt, die Custom Elements, in Abschnitt 2.3 wird auf den Shadow DOM eingegangen, Abschnitt 2.4 widmet sich den HTML Templates und Abschnitt 2.5 zeigt die letzte Technologie, die HTML Imports. In Abschnitt 2.6 werden die Polyfills erklärt, welche für die Technologien noch zwingend notwendig sind. Abschließend wird in Kapitel 2.7 anhand der in diesem Kapitel erklärten Technologien eine exemplarische Komponente implementiert.

2.1 Problemlösung

In der heutigen Webentwicklung kommt es häufig vor, dass oftmals für diverse Probleme die gleiche, oder eine ähnliche Lösung programmiert werden muss. So muss auf vielen Seiten ein Slider, eine Navigation oder eine andere Komponente, welche das gewünschte Feature beinhaltet, eingebunden werden. Diese unterscheiden sich stets leicht, bringen im Kern aber dennoch meist die selben Funktionen mit sich. Um diese Funktionen auf der Webseite verfügbar zu machen, sind eine Reihe an verschiedenen Technologien notwendig. Wenn die gewünschte Komponente bereits existiert, muss für das Einbinden dieser Komponente ein bestimmtes HTML-Markup geschrieben werden. Damit die Komponente nun funktioniert, muss ein JavaScript eingebunden werden, welches zusätzlich noch anhand einer vordefinierten API konfiguriert werden muss. Diese API ist in der Regel nur für diese eine Komponente entworfen, so müssen für jede Komponente unterschiedliche APIs angesprochen werden, die sich mitunter stark unterscheiden können. Damit die Komponente dann auch in das visuelle Design der eigenen Webseite passt, muss ebenso 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. In diesem Fall muss auch das Stylesheet noch nachgebessert werden. Nimmt man diese Punkte zusammen, so wird deutlich, dass es in der Webentwicklung kein Plugin-System gibt, mit dem Webseiten schnell und einfach erweitert werden können. Diesem Problem widmen sich die Web Components. Sie sollen der Frontend-Entwicklung ein Plugin-System bereitstellen, welches es ermöglicht, fremde und eigene Komponenten schnell und einheitlich in die eigene Seite einzubinden. Eine Komponente steht dabei als eigenes HTML-Element, welches ihre gesamte innere Funktionalität in sich kapselt und nach außen unsichtbar macht. 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 den Umgang mit Plugins deutlich, da die einzige dafür notwendige Technologie HTML selbst ist. Dadurch können einzelne Komponenten verwendet werden wie jedes 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 die grundlegenden 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 ihre eigenen, von ihrer Applikation benötigten Elemente, definieren können. Betrachtet man in Abbildung 2.1 den Quelltext einer populären Webseite im Internet, wird schnell deutlich, worin das Problem liegt.

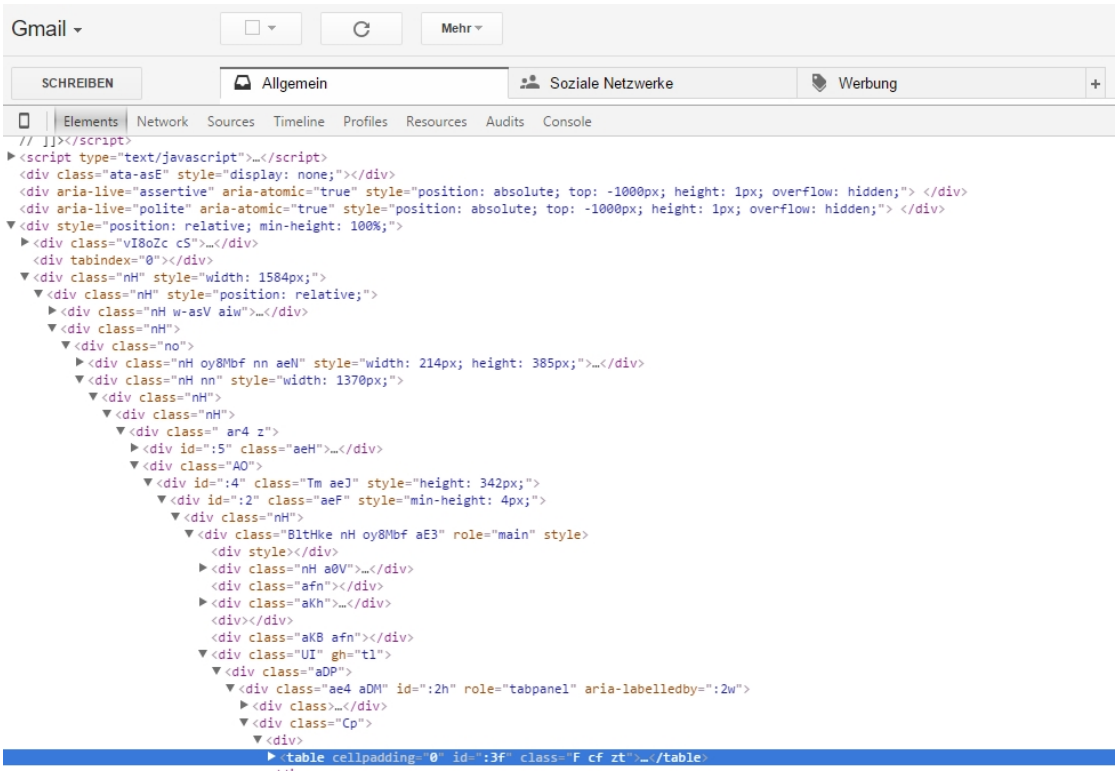

Abbildung 2.1: Webseite mit semantisch nicht aussagekräftigem Markup

Die Webseite der Google Mail Applikation ist stark geschachtelt in -Elemente. Diese sind notwendig, um der Webseite die gewünschte Funktionalität und Aussehen zu verleihen. Die Probleme dieser Struktur bzw. des DOMs sind deutlich: Es ist sehr schwer zu erkennen, welches Element nun was darstellt und welche Funktion hat. Abgesehen von der fehlenden schnell ersichtlichen Semantik - also der Zuordnung der Bedeutung zu einem Element - ist das gesamte DOM nur schwer wartbar. Dieser Problematik widmen sich die Custom Elements. Sie bieten eine neue API, welche es ermöglicht, eigene, semantisch aussagekräftige HTML-Elemente sowie deren Eigenschaften und Funktionen zu definieren. Würde das obige Beispiel nun also mit Hilfe von Custom Elements umgesetzt werden, so könnte das zugehörige DOM wie folgt aussehen [Bidelman 2013a].

```
<hangout-module>
  <hangout-chat from="Paul, Addy">
```

```
    <hangout-discussion>
      <hangout-message from="Paul" profile="profile.png"
          profile="118075919496626375791" datetime="2013-07-17T12:02">
        <p>Hier werden Web Components eingesetzt.</p>
      </hangout-message>
    </hangout-discussion>
  </hangout-chat>
  <hangout-chat>...</hangout-chat>
</hangout-module>
```

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. Dies ermöglicht es, gewünschte Funktionalitäten in selbsterstellten HTML-Elementen zu bündeln.

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, beispielsweise `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 `var MyElement = 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, auf dieses Thema wird jedoch in Abschnitt 2.2.4 gesondert eingegangen. Durch das Registrieren des Elements wird es in die Registry des Browsers geschrieben, welche dazu verwendet wird, die Definitionen der HTML-Elemente aufzulösen. Nachdem das Element registriert wurde, muss es zunächst mittels `document.createElement(tagName)` erzeugt werden, der `tagName` ist hierbei der Name des zuvor registrierten Elements. Danach kann es per JavaScript oder HTML-Deklaration im Dokument verwendet werden [Overson und Strimpel 2015] .

seitenzahl

Einbinden mit JavaScript:

```
document.body.appendChild(myelement);
```

Einbinden mit HTML:

```
<div class="some-html">
  <my-element><my-element>
</div>
```

2.2.2 Vorteile von Custom Elements

Ist ein Element noch nicht definiert und nicht beim Browser registriert, steht aber im Markup der Webseite, beispielsweise <myelement>, wird dies kein Fehler verursachen, da dieses Element das Interface von HTMLUnkownElement benutzen muss [Group 2015]. Ist es jedoch definiert oder beim Browser registriert worden, beispielsweise mit <my-element>, so benutzt es das Interface eines HTMLElement. Dies bedeutet, dass für dieses Element eigene APIs erzeugt werden können, indem eigene Eigenschaften und Methoden hinzugefügt werden [Bidelman 2013a]. Eigene Elemente mit einem spezifischen Eigenverhalten und Aussehen, wie beispielsweise ein neuer Video-Player, sind dadurch mit einem Tag statt mit einem Gerüst aus -Tags oder Ähnlichem umsetzbar.

2.2.3 Nachteil

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 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 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. Dadurch wird ein ungewolltes Anzeigen von ungestylten Inhalten verhindert [Gasston 2014].

```
my-element:unresolved {
  display: none;
}
```

2.2.4 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 erzeugt werden, muss Folgendes gemacht werden:

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

Das registrierte, erweiterte Element kann nun 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 [Kitamura 2014].

JavaScript:

```
var buttonExtended  = document.createElement('button', 'button-extended');

// Alternativ
var buttonExtended = new ButtonExtendedProto();
```

Um es nun im DOM zu benutzen, muss der Name des erweiterten Elements via dem Attribut `is="elementName"` des erweiternden Elements angegeben werden.

HTML:

```
<div class="wrapper">
  <button is="button-extended"></button>
</div>
```

Verwendung bei Github

Eine Umsetzung der Type extensions ist auf der Webseite von GitHub zu finden. Dort werden die "Latest commit" Angaben eines Repositories als ein erweitertes `time`-Element dargestellt. Statt des Commit-Datums und der Zeit, wird die berechnete Zeit seit dem letzten Commit angezeigt, wie in Abbildung 2.2 dargestellt.

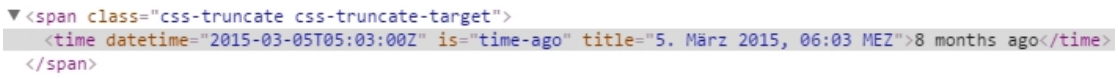

Abbildung 2.2: Github Einsatz eines Custom Element

GitHub verwendet hierzu ein selbst erzeugtes `time-ago`-Element, welches eine Type extension auf Basis des 'time'-Elements umsetzt. Mittels dem `datetime`-Attribut wird die absolute Zeit des Commits an das interne JavaScript weitergegeben. Als Inhalt des `time`-Elements wird dann die mit JavaScript berechnete relative Zeit ausgegeben. Falls der Browser nun keine Custom Elements unterstützt oder JavaScript deaktiviert ist, wird dennoch das nicht erweiterte, native HTML `time`-Element mit der absoluten Zeit angezeigt.

2.2.5 Eigenschaften und Methoden definieren

Anhand des Beispiels auf GitHub 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] . 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.

Seitenzahl

```
<script>
// Methode definieren
ButtonExtendedProto.alert = function () {
  alert('foo');
};

// Eigenschaft definieren
ButtonExtendedProto.answer = 42;
</script>

<!-- Beispiel einer deklarativen Konfiguration -->
<button-extended answer="41">Ich bin ein Button</button-extended>
```

2.2.6 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

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

attachedCallback

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

detachedCallback

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

attributeChangedCallback

Die `attributeChangedCallback`-Funktion 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 folgt definiert werden [Schaffranek 2014].

```
var ButtonExtendedProto = Object.create(HTMLElement.prototype);

ButtonExtendedProto.createdCallback = function() {...};
ButtonExtendedProto.attachedCallback = function() {...};

var ButtonExtended = document.registerElement('button-extended', {
  prototype: ButtonExtendedProto
});
```

2.2.7 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. Erweiterte Elemente können mittels dem Attribut-Selektor in CSS angesprochen werden [Overson und Strimpel 2015] .

Seitenzahl

```
/* Eigenes Custom Element */
my-element {
  color: black;
}

/* Erweitertes natives HTML-Element*/
[is="button-extended"] {
  color: black;
}
```

2.2.8 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 2.3) [Can I Use - Custom Elements web].

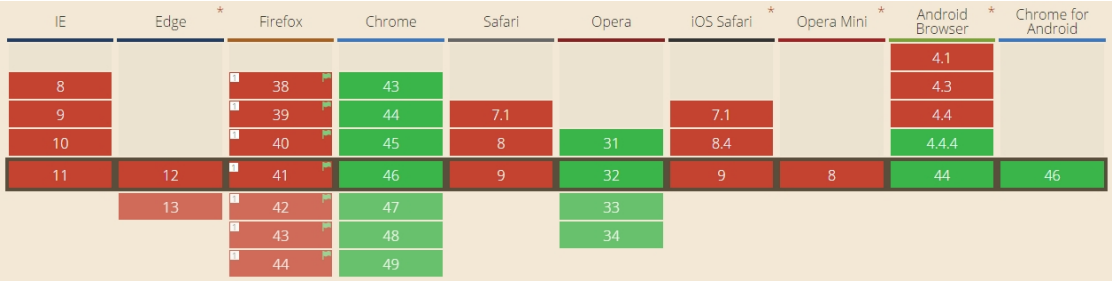

Abbildung 2.3: Browserunterstützung von Custom Elements

2.3 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 des Hauptdokuments auch oft als "Light DOM" bezeichnet. Das Shadow DOM wird bereits in HTML5 standardmäßig eingesetzt, wie beispielsweise im <video>-Tag. Beim Inspizieren des Elements mit Hilfe der Chrome Developer Tools oder den Firefox-Entwicklungswerkzeugen wird deutlich, dass das <video>-Tag ein Shadow DOM beinhaltet, welcher die Steuerelemente des Videos erzeugt. Neben dem <video>-Tag sind auch die verschiedenen <input>-Elemente, wie z.B. das <input type="password"> (siehe Abbildung 2.4) mit einem Shadow DOM ausgestattet [Overson und Strimpel 2015, S. 109-126].

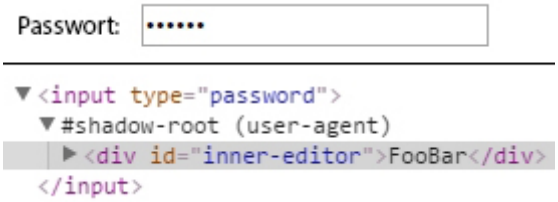

Abbildung 2.4: Passwort-Input-Element

2.3.1 Shadow DOM 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, den Shadow DOM, und alles was dieser beinhaltet, zu kapseln [Ihrig 2012]. Dies verhindert, dass externes CSS oder JavaScript das interne Markup oder umgekehrt internes CSS oder JavaScript den 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

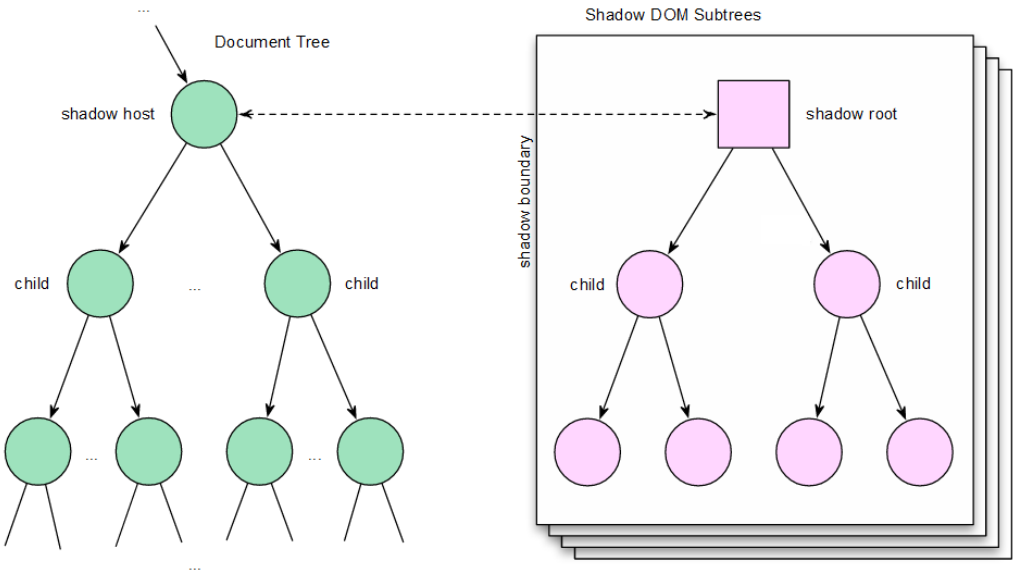

Abbildung 2.5: Shadow DOM und Shadow Boundary nach W3C

dieser zum Rendern einen 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.

2.3.2 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 zwei Arten von Insertion Points unterschieden werden.

2.3.3 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 [Cooney und Bidelman 2013].

```
<div id="shadow">Inhalt</div>
<template id="myTemplate">
  <div class"hiddenWrapper">
    <content></content>
  </div>
</template>
```

```
var shadow = document.querySelector('#shadow').createShadowRoot();
var template = document.querySelector('#myTemplate');
var clone = document.importNode(template.content, true);
shadow.appendChild(clone);
```

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 außen gestylt werden können, mehr dazu im Abschnitt 2.3.5. 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')); 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.

2.3.4 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 sogenannten “Younger Tree” - ein <shadow>-Tag stehen, dieser rendert den zuvor hinzugefügten Shadow DOM - den sogenannten “Older Tree”. Somit wird eine Schachtelung mehrerer Shadow DOMs ermöglicht [Bidelman 2014].

2.3.5 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.

2.3.6 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`.

>>> (ehem. /deep/)

Der >>> 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 (ehem. ::content)

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.

2.3.7 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, besonders 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.

2.3.8 Beispiel eines Shadow DOMs mit Template und CSS

Das folgende Beispiel zeigt eine exemplarische Implementierung eines Shadow DOMs, welcher gekapseltes CSS und JavaScript beinhaltet.

```
<style>
  .styled {
    color: green;
  }
  .content {
    background-color: khaki;
  }
</style>

<div id="hello">
  <div>Hello</div>
  <div class="styled">Styled</div>
  <p class="hidden">Inhalt, der vom Shadow DOM überschrieben wird.</p>
</div>

<template id="myTemplate">
  <style>
    .wrapper {
      border: 2px solid black;
      width: 120px;
    }
    .content {
      color: red;
    }
  </style>

  <div class="wrapper">
```

```
    Shadow DOM
    <div class="content">
      <content select="div"></content>
    </div>
  </div>
</template>

<script>
  var root = document.querySelector('#hello').createShadowRoot();
  var template = document.querySelector('#myTemplate');
  var clone = document.importNode(template.content, true);
  root.appendChild(clone);
</script>
```

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

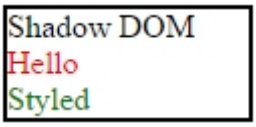

Abbildung 2.6: Shadow DOM Beispiel

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, welcher per `var root = document.querySelector('#hello').createShadowRoot()` erzeugt wird, gerendert. 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 <`div`>-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 <`div`>-Element ohne eine zugehörige Klasse mit der Regel `.content { color: red; }` auch dementsprechend in Rot gerendert.

2.3.9 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].

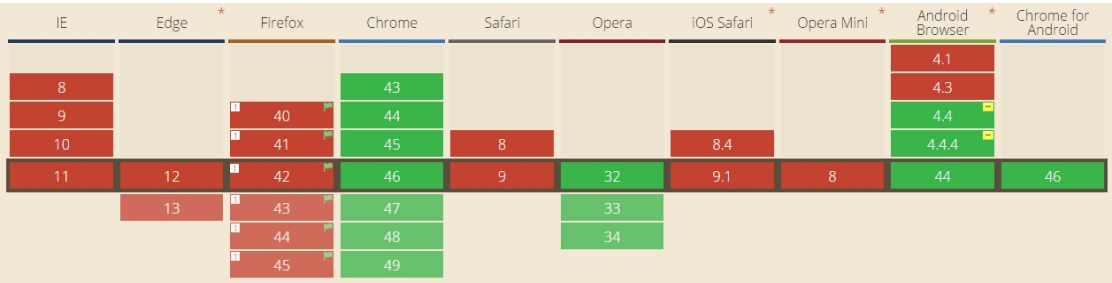

Abbildung 2.7: Browserunterstützung des Shadow DOMs

2.4 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 MVC-Applikation ist der Mechanismus der Darstellung der Präsentations-Schicht, auch View genannt, 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.

> 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. [Network 2015b]

2.4.1 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 [Webcomponents 2014].

Via verstecktem -Element

Das folgende Beispiel zeigt die Umsetzung eines Templates mit Hilfe eines -Blocks, der via CSS versteckt wird.

```
<div id="mydivtemplate" style="display: none;">
  <div>
    <img src="myimage.jpg">
  </div>
</div>
```

Der entscheidende Nachteil dieser Methode ist, dass alle Ressourcen, also alle verlinkten Dateien, beim Laden der Webseite auch heruntergeladen werden. Zwar werden sie

nicht angezeigt, dennoch verursachen sie eine große Datenmenge, welche initial übertragen werden muss. Dies geschieht in diesem Fall selbst wenn die Ressourcen eventuell erst später oder gar nicht benötigt werden, was eine massive Einschränkung der verfügbaren Bandbreite und Browser-Performance mit sich bringen kann. 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 wird hier somit nicht vorgesehen.

Via <script>-Element:

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

```
<script type="text/template">
  <div>
    <img src="myimage.jpg">
  </div>
</script>
```

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

2.4.2 Das <template>-Tag

Den Problemen der oben genannten Methoden widmet sich der <template>-Tag, welcher eine native und sichere Methode für das Einbinden von dynamischen Inhalten etabliert. Das Template und die darin enthaltenen Inhalte werden beim Rendern des Webseite vollständig ignoriert, sie werden weder angezeigt, noch werden ihre benötigten Inhalte beim Laden der Webseite mitgeladen. Ebenso werden enthaltene JavaScripts nicht ausgeführt, auch kann JavaScript von außen nicht in das Template hinein traversieren. Im folgenden wird die grobe Struktur eines einfachen Templates, das mit Hilfe des <template>-Tags umgesetzt wird, dargestellt.

```
<template id="mytemplate">
  <style>/* Styles */</style>
  <script>// JavaScript</script>
```

```
  <img src="bild.jpg"> <!-- Kann zur Laufzeit dynamisch gesetzt werden -->
  <p class="text">Hier steht ein Text.</p>
</template>
```

2.4.3 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`) und als zweiter Parameter ein Boolean für `deep`, welcher angibt ob auch Kinderknoten geklont werden sollen. Nun kann der Inhalt des Templates mittels `document.body.appendChild(templateClone);` an einer beliebigen Stelle des DOM eingefügt werden.

2.4.4 Vorteile

Die Vorteile dieser nativen Implementierung für Templates sind vielfältig. So sind HTML Templates ein fertiges Gerüst an HTML, das nicht nachträglich mit JavaScript modifiziert werden muss, es kann aus dem Quelltext kopiert und beliebig oft und an beliebiger Stelle in den DOM der Webseite eingefügt werden. Erst beim einfügen in den DOM werden die Inhalte tatsächlich gerendert und Abhängigkeiten nachgeladen. Darunter fallen auch enthaltene Styles oder JavaScript-Codes, welche erst beim Einfügen angewendet und ausgeführt werden. So werden auch externe Stylesheets, JavaScript-Dateien oder Bilder und Videos erst dann geladen und abgespielt, wenn sie tatsächlich benötigt werden. Dadurch können auch 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. Des Weiteren sind Templates komplett vor dem DOM versteckt, will man beispielsweise mit JavaScript in das Template mittels `document.getElementById('#mytemplate .text')` hinein traversieren, so gibt die Funktion 'null' zurück. Der abschließende und wohl auch größte Vorteil ist, dass mit JavaScript auf das Template zugegriffen werden und es an anderer Stelle dynamisch eingebunden werden kann. Falls nun jedoch in einem Template mehrere weitere Templates geschachtelt sind, so muss jedes dieser Templates einzeln aus dem aktiven Template im DOM kopiert und wieder eingefügt werden um es zu aktivieren.

2.4.5 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 2.8). Sie sind des Weiteren die einzige Technologie der Web Components, die bisher von Microsofts Edge ab Version 13 unterstützt werden.

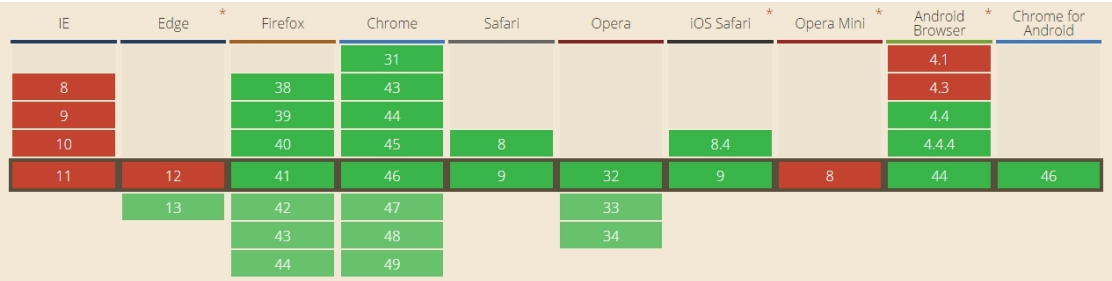

Abbildung 2.8: Browserunterstützung des HTML Template Tags

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]

```
<head>
  <link rel="import" href="/my-import.html">
</head>
```

Sollte nun ein HTML Import mehrfach aufgeführt sein, oder eine HTML-Datei anfordern, die schon geladen wurde, so wird die Abhängigkeit automatisch ignoriert und die Datei nur ein einziges Mal übertragen. Dadurch wird eventuell in den HTML-Dateien enthaltenes JavaScript auch nur ein mal ausgeführt. 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 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. [Network 2015a]

Sollte das jedoch dennoch erlaubt werden, so muss das 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 HTTP requests that can be automatically launched toward destinations that differ from the running application's origin. [W3c 2014a]

2.5.2 Vorteil

Durch HTML Imports ist es möglich, komplette Applikationen mit mehreren oder gar nur einer einzigen Anweisung zu importieren. Dies gilt sowohl für eigene Applikationen oder kleinere Komponenten, wie beispielsweise einem Slider oder ähnlichem, als auch fremde Frameworks oder Komponenten. Durch die HTML Imports wird die Abhängigkeiten-Verwaltung stark vereinfacht und automatisiert.

> Using only one URL, you can package together a single relocatable bundle of web goodness for others to consume. [Rocks 2013]

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 folgt aussehen.

```
<link rel="stylesheet" href="bootstrap.css">
<link rel="stylesheet" href="fonts.css">
<script src="jquery.js"></script>
<script src="bootstrap.js"></script>
<script src="bootstrap-tooltip.js"></script>
<script src="bootstrap-dropdown.js"></script>
```

Stattdessen kann dieses Markup nun in ein einziges HTML Dokument, welches alle Abhängigkeiten verwaltet, geschrieben werden. Dieses wird dann mit einem einzigen Import in das eigene HTML Dokument importiert.

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

2.5.3 HTML Imports verwenden

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 und auf sie zuzugreifen muss auf die `.import` Eigenschaft des `<link>`-Tags mit dem Import zugegriffen werden `var content = document.querySelector('link[rel="import"]').import;`. Nun kann mit auf das DOM des Imports zugegriffen werden. Beispielsweise 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 nun jedoch mehrere Imports geben sollte, können den Imports IDs zugewiesen werden, anhand derer die Imports voneinander unterschieden werden können [Hongkiat 2014]. Der komplette Prozess wird in dem folgenden Beispiel skizziert.

```
<head>
  <link rel="import" href="my-import.html">
</head>
<body>
  <script>
    var content = document.querySelector('link[rel="import"]').import;
    var el = content.querySelector('.element');
    document.body.appendChild(el.cloneNode(true));
  </script>
</body>
```

Import mit ID machen (mehrere), Code Highlighting korrigieren

2.5.4 Abhängigkeiten verwalten

Doch wie sieht es nun mit Abhängigkeiten von Imports aus? So kommt es des öfteren vor, dass verschiedene Bibliotheken, welche auf der Seite eingebunden werden, die gleichen Abhängigkeiten haben und diese verwaltet werden müssen. Sollte dies nicht sauber gemacht werden, können unerwartete Fehler auftreten, die mitunter nur schwer zu identifizieren sind. HTML Imports verwalten diese automatisch. Um das zu erreichen, wird jede Abhängigkeit in eine zu importierende HTML-Datei geschrieben, welche diese bündelt. Haben nun mehrere solcher Imports die gleichen Abhängigkeiten, werden diese vom Browser erkannt und nur einmal heruntergeladen und eingebunden. Mehrfachdownloads und Konflikte der Abhängigkeiten werden so verhindert. Das folgende Beispiel von [Webcomponents 2015] verdeutlicht dieses Szenario.

index.html

```
<link rel="import" href="component1.html">
<link rel="import" href="component2.html">
```

component1.html

```
<script src="jQuery.html"></script>
```

component2.html

```
<script src="jQuery.html"></script>
```

jQuery.html

```
<script src="js/jquery.js"></script>
```

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 auf die Reihenfolge der Imports geachtet werden, da diese selbst ihre Abhängigkeiten beinhalten.

2.5.5 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 [Rocks 2013].

2.5.6 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].

2.5.7 HTTP/2

Dennoch sind viele Einzel-Requests ein gewünschtes Verhalten von Web Components, sie sind also ein Feature und kein Bug. Dieses Feature kommt jedoch erst zum Tragen, wenn HTTP in Version Zwei in allen Browsern Standard ist. HTTP/2 bietet eine Reihe an Vorteilen gegenüber dem heute benutzen HTTP/1.1. So können mehrere Requests in einer TCP-Verbindung übertragen werden, wobei die Reihenfolge der Antworten auf die Requests keine Rolle spielt. Des Weiteren kann der Client, also der Absender der Requests, die Priorität der angeforderten Dateien bestimmen, somit kann der Server Dateien mit einer hohen Priorität vor Dateien mit niedrigerer Priorität schicken. Um die Größe der zu sendenden Pakete zu reduzieren, setzt HTTP/2 eine drastische Header-Kompression ein, welche die Bruttogröße der zu übertragenden Pakete 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. Durch diese Reihe an neuen Features bietet es also keinen Vorteil mehr, möglichst wenige Requests

an den Server zu schicken um die Ladezeit der Webseite zu optimieren. HTTP/2 sieht es also vor, seine Webseite in mehrere kleine Dateien zu unterteilen, sodass bei kleinen Änderungen einer Datei, wie beispielsweise einem Icon, nicht mehr das gesamte Bild-Sprite, also das konkatenierte Bild, welches sich aus allen Einzelbildern zusammensetzt, sondern eben nur das neue, geänderte Icon neu übertragen werden muss. Ebenso können einzelne Teile einer Webseite wie Bild-Slider, Navigations-Menüs, komplexe Widgets, etc. schnell ausgewechselt werden, ohne die komplette Seite bei jeder Anfrage übertragen zu müssen. Das Zusammenfassen von Dateien wird folglich auf Protokollebene von HTTP/2 übernommen, die Entwickler einer Webseite müssen sich nicht selbst um dieses Problem kümmern. 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, hier muss allerdings darauf hingewiesen werden, dass auch dort TLS als Verschlüsselung von Webseiten verwendet werden muss, damit das HTTP/2 Protokoll benutzt werden kann [citeulike:13879562]. Auch wenn das Protokoll schon standardisiert werden ist, liegt der Market Share an Verbindungen mit HTTP/2 momentan bei ca. 3% [W3Techs und W3Techs web], es muss de facto darauf verzichtet werden und eine alternative Lösung für das Problem der vielen Requests gefunden werden.

2.5.8 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.

2.5.9 Request-Minimierung mit “Vulcanize”

Webseiten können viele verschiedene, modular aufgebaute Stylesheets, JavaScript-Dateien 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 können die

konkatenierten Dateien 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 [Google web] bereit, welches serverseitig ermöglicht, einzelne kleine Web Components eine einzige große Web Component zu konkatenieren. 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 Request gebündelt und Ladezeiten und Bandbreite minimiert.

2.5.10 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 einer Anwendung oder von 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 einen enormen Performance-Schub einher geht [Rocks 2013].

2.5.11 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 2.9). 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 [Hacks 2015]. Auf Details zu ECMAScript 6 Modules wird an dieser Stelle nicht eingegangen, da diese den Umfang dieser Arbeit überschreiten.

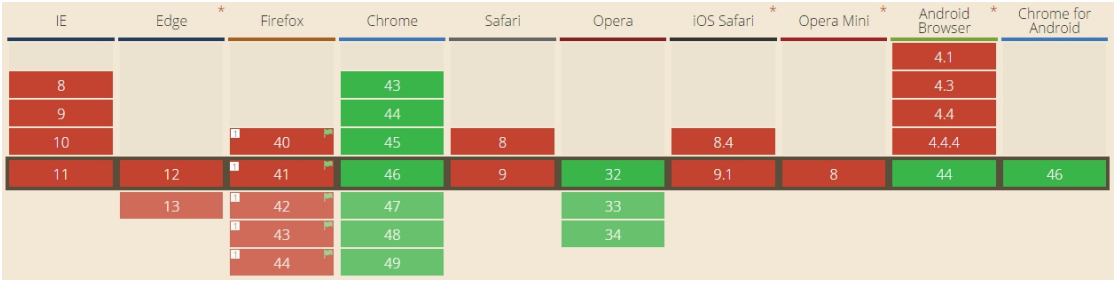

Abbildung 2.9: Browserunterstützung der HTML Imports

2.6 Polyfills mit webcomponents.js

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, sowie deren Browserunterstützung und Performance, und wie sie die Browserunterstützung der Web-Components-Technologien verbessern.

2.6.0.1 Native Browserunterstützung von Web Components

In den einzelnen Unterkapiteln zu den Technologien wurde jeweils kurz gezeigt, ob sie 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 manuell mit dem Flag `dom.webcomponents.enabled` in den Entwicklereinstellungen aktiviert werden. HTML Imports werden, wie in Kapitel erwähnt, bis auf weiteres nicht unterstützt (Siehe Kapitel 2.5).

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 Unterstützung der modernen Browser ist also noch verhalten, wird sich aber stark verbessern. Das bedeutet jedoch nicht, dass die Web Components noch nicht verwendet werden können. Mittels JavaScript besteht die Möglichkeit, deren Funktionalitäten den aktuellen Browsern, welche Web Components nicht unterstützen, sowie noch älteren

Browsern beizubringen. Das hierfür benutzte JavaScript wird Polyfill genannt, mit dessen Hilfe können die Funktionen auf alle relevanten Browser portiert werden.

2.6.1 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. Flattening the API landscape if you will. [Sharp 2010]

Mit Hilfe von JavaScript kann eine Technologie also auch in Browsern benutzt werden, welche die Technologie nicht unterstützen. Mit Hilfe von Polyfills können Technologie-Lücken in Browsern auf mehrere, unterschiedliche Arten ("Poly") gefüllt ("fill") werden [Satrom 2014]. Eine Sammlung an Polyfills für die verschiedenen 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 entschlossen wurde, es auszugliedern, damit es auch unabhängig von der Benutzung von Polymer eingesetzt werden kann [Webcomponents web].

2.6.2 Browserunterstüzung

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 2.10 [Webcompontents web] dargestellt.

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

Abbildung 2.10: Browserunterstützung der Web Components Technologien mit webcomponents.js

Jedoch werden auch trotz Einsatz des Polyfills nur die aktuelleren Versionen des jeweiligen Browsers unterstützt. Darunter fallen weiterhin nicht 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 XHR, und somit asynchron heruntergeladen, selbst wenn das `async`-Attribut (siehe Abschnitt 2.5.8) nicht gesetzt ist.

2.6.3 Performance

Das webcomponents.js-JavaScript bringt mit seiner Größe von 116KB [Webcompontents web] 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 Browsern - drastisch sinkt und mitunter nicht tolerierbar ist [Polymer - Shady DOM web].

2.7 Implementierung einer Komponente mit den nativen Web Component APIs

Anhand der vorhergehenden Abschnitte wird in diesem Abschnitt die Implementierung der Web 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. Die gerenderte Komponente wird in Abbildung 2.11 dargestellt.

Abbildung 2.11: Gerenderte Web Komponente mit nativen APIs

2.7.1 Custom Element erstellen und 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.

```
CustomElementProto.createdCallback = function() {
  if (this.hasAttribute('theme')) {
    var theme = this.getAttribute('theme');
    this.setTheme(theme);
  } else {
    this.setTheme(this.theme);
  }
};

CustomElementProto.attributeChangedCallback = function(attr, oldVal, newVal) {
  if (attr === 'theme') {
```

```
    this.setTheme(newVal);
  }
};
```

Die `createdCallback`-Funktion soll zunächst prüfen ob das Attribut `theme` beim Verwenden 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 Komponente 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.

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

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, welches die für die Komponente notwendige HTML Struktur beinhaltet.

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

Das Template enthält dabei einen Insertion Point <`content`>, in welchem die Kind-Elemente der Komponente in das interne Markup projiziert werden. Zusätzlich werden

zwei Hilfs-Wrapper und Text definiert, damit die Elemente schneller mittels JavaScript selektierbar sind und das gewünschte Aussehen erreicht wird. Um nun die verschiedenen Styles, welches mittels dem `theme`-Attribut ausgewählt werden kann, zur Verfügung zu stellen, werden diese in einem <`style`>-Tag in dem Template definiert. In diesem Beispiel werden zwei Optionen, `style1` und `style2` zur Verfügung gestellt, sowie weitere Styles für die gesamte Komponente definiert.

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

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 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 der Shadow DOM sofort initialisiert werden, wenn 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.

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 auf der Seite eingebettet werden, wie beispielsweise der Konfiguration <`custom-element theme="style1">Reader</custom-element`>.

Das vollständige Beispiel der Komponente, sowie dessen Einbindung in ein HTML-Dokument sind im Anhang zu finden.

Abkürzungsverzeichnis

Abbildungsverzeichnis

Listings

TODOs

Literaturverzeichnis

[Bateman 2014] Bateman, C. (2014). A No-Nonsense Guide to Web Components.

[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). Can I use - Custom Elements. http://caniuse.com/#feat=custom-elements.

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

[Can I Use - Shadow DOM web] Can I Use - Shadow DOM (web). Can I use - 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/.

[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). W3C - Custom Elements. http://w3c.github.io/webcomponents/spec/custom/#concepts.

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

[Group 2015] Group, W. H. A. T. W. (2015). HTML Standard. Technical report.

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

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

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

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

[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.

[Microsoft web] Microsoft (web). Developer Resources : Microsoft Edge Dev.

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

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

[Network 2015b] Network, M. D. (2015b). ¡template¿ - HTML. https://developer.mozilla.org/de/docs/Web/HTML/Element/template.

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

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

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

[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?

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

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

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

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

[Webcomponents web] Webcomponents (web). Polyfills.

[Webcompontents web] Webcompontents (web). webcomponentsjs.