BLOG

Web Components

NATIVE KOMPONENTEN IM WEB OHNE FRAMEWORK

21 Apr 2020

Manuel Rauber

Aktuell helfen uns viele Frameworks, im Web Komponenten zu entwickeln. Dabei hat jedes Framework seine individuelle Ausprägung, wie der Code zu strukturieren ist, welche Features oder Lifecycle-Methoden die Komponente hat. Wechseln wir von Framework A zu Framework B, müssen wir mitunter einiges Neues lernen und uns auf die Beschaffenheit des neuen Frameworks einlassen. Mit Web Components zieht ein natives Komponentenmodell in den Browser ein. Ist das die Abhilfe? Und damit der Untergang der Frameworks?

Komponenten – ein Wort, das wir Windows-Entwickler seit vielen Jahren kennen und zu schätzen wissen. Auch im Web ist eine Komponente nichts Neues. Damals, zu jQuery-Zeiten, half uns jQuery, UI-Komponenten zu entwickeln. Heute nutzen wir hier Frameworks wie Angular, React oder Vue. Doch was leistet so eine Komponente eigentlich?

Eine Komponente kapselt Funktion in Form von Code, UI-Struktur und UI-Design. Heruntergebrochen auf das Web bedeutet das z. B. Code in Form von JavaScript oder TypeScript, UI-Struktur mit HTML und UI-Design mit CSS. Mit Hilfe dieser Programmier- und Auszeichnungssprachen erhalten wir wiederverwendbare Komponenten, aus denen wir unsere finale Anwendung komponieren. Oftmals besteht eine Anwendung aus vielen kleinen Komponenten. Jede mit einer eigenen, ganz bestimmten Funktion und oft auch einer Schnittstelle nach außen, sodass der Entwickler Daten in die Komponente geben kann, aber auch Daten aus ihr erhält. Wirft man einen Blick auf die drei großen Single Page Application (SPA) Frameworks, Angular, React und Vue, geben alle drei dem Entwickler ein Komponentenmodell an die Hand. Sei es bei Angular via *@Component*-Dekorator, bei React via Ableitung von

React.Component und bei Vue die Funktion *Vue.component()*. All das hilft uns, im jeweiligen Framework unsere Business Use Cases in Komponenten zu gießen.

Auch wenn alle drei genannten Frameworks mit dem Komponentenmodell eine Gemeinsamkeit haben, ist die Entwicklung danach grundverschieden. Jedes Framework hat ein eigenes Konzept, wie Daten an das UI übermittelt werden oder auf Benutzereingaben in Form von Events reagiert werden können. Auch die Lifecycle-Methoden unterscheiden sich etwas. Alle Frameworks bieten Methoden an, die in der jeweiligen Komponente aufgerufen werden, wenn diese z. B. zur Anzeige gebracht oder vom Framework wieder zerstört wird. Daneben existieren allerdings weitere Framework-spezifische Lifecycle-Methoden. Für uns Entwickler bedeutet das, dass wir einen Teil unseres Basiswissens, dem Verständnis über das Web, Framework-übergreifend verwenden können. Alles andere müssen wir für jedes Framework immer wieder neu lernen, was gerne auch mal mit dem einen oder anderen Fallstrick verbunden ist.

(https://basta.net/mainz/)

WEB COMPONENTS – EIN NATIVES KOMPONENTENMODELL

Bricht man alle Frameworks auf ihre Basisidee herunter, entstanden sie alle aus einem Grund: Dem Nachrüsten eines Komponentenmodells im Browser, da dieser keines zur Verfügung stellt. Das manche Frameworks dabei etwas größer wurden und Mehrwertdienste anbieten, wie z. B. eine Dependency Injection, ist eine individuelle Entscheidung der Framework-Entwickler, um dem Entwickler mehr Funktion an die Hand zu geben. Seit geraumer Zeit entwickeln und etablieren sich drei Standards im Web, die es ermöglichen, dass uns der Browser ein natives Komponentenmodell zur Verfügung stellt: Custom

Elements, Shadow DOM und HTML Templates. Vor einiger Zeit hätte man noch einen vierten Standard, HTML Imports, hinzugezählt. Dieser wird allerdings zugunsten von ES Module Imports nicht mehr benötigt. Sehen wir uns diese drei Standards einmal genauer an.

CUSTOM ELEMENTS

 – kommt Ihnen das bekannt vor? In früheren Zeiten der Webentwicklung hat diese Angabe, nebst der Einbindung einer JavaScript-Bibliothek und CSS-Dateien, gereicht, um aus einem -Element einen Datepicker zu machen. Zur Laufzeit der Anwendung wurde dieses
 dann um weitere Elemente und Funktionen erweitert. Viel schöner wäre es doch, wenn wir stattdessen <my-datepicker></my-datepicker> schreiben könnten. Und genau das ermöglicht uns das Custom Elements API. Es erlaubt uns, dem Browser neue HTML-Tags beizubringen, die dann mit einem von uns definierten Inhalt gerendert werden. Damit das klappt, brauchen wir die nächsten zwei Standards.

SHADOW DOM

Wir alle kennen das Problem im Web: Man entwickelt eine schöne Komponente, bindet sie in eine Drittanbieterwebseite ein und in der Regel passieren zwei Dinge. Erstens sieht unsere Komponente meist nicht mehr so aus, wie wir es wollten. Zweitens sehen plötzlich

Dinge auf der Website nicht mehr so aus, wie sie eigentlich sein sollten. Das liegt oft daran, dass zu generische CSS-Selektoren genutzt wurden, die der Browser natürlich auf alle gefundenen Elemente der kompletten Website anwendet. Genau hier setzt Shadow DOM an und bietet Abhilfe. Shadow DOM ist ein Set von JavaScript APIs, die es uns ermöglichen, einen sogenannten Shadow DOM Tree einem HTML-Element anzuhängen, der vom Browser separat und außerhalb des Main Document DOM gerendert wird. Sind im Shadow DOM Tree CSS-Angaben enthalten, werden diese auch nur auf die Elemente innerhalb des Shadow DOM Trees angewendet. CSS-Angaben außerhalb, also sprich im Main Document DOM, haben keine Auswirkungen auf die Elemente innerhalb des Shadow DOM. Dadurch können Komponenten genauso gestylt werden, wie wir es gerne hätten, und wir brauchen keine Sorge haben, dass jemand unsere CSS-Stile überschreibt. Jetzt bleibt nur noch die Frage offen, wie wir eigentlich definieren, welche HTML-Elemente im Shadow DOM angezeigt werden? Dazu benötigen wir den letzten Standard.

HTML TEMPLATES

Im Wesentlichen besteht dieser Standard aus zwei HTML-Tags, nämlich <template> und <slot>. Das <template&gt gibt an, welche HTML-Elemente gerendert werden sollen. Den <slot> kann man quasi als Platzhalter im Template sehen, dazu später mehr. Der Unterschied ist allerdings, dass der Browser alles innerhalb des <template>-Tags nicht rendert. Er parst den Inhalt und baut das passende DOM dazu auf, bringt es aber nicht zur Anzeige. Erst mit der eigentlichen, und auch wiederverwendbaren Nutzung des Templates wird es vom Browser gerendert.

DER ZÄHLER – EIN BEISPIEL EINER WEB COMPONENT

Mit Hilfe der drei genannten Technologien sind wir in der Lage, eine eigene wiederverwendbare Web Component zu entwickeln. Und genau das wollen wir in den nächsten Abschnitten machen, um uns so Schritt für Schritt den wichtigsten APIs der Standards zu nähern. Das fertige Beispiel finden Sie auf GitHub (https://github.com/thinktecture-labs/windows-developer-web-components). Zur Entwicklung benötigen Sie einen Codeeditor, Node.js und Google Chrome. Auch wenn die Evergreen-Browser mittlerweile beinahe alle APIs unterstützen, empfiehlt es sich für diesen Artikel, Google Chrome zu verwenden. Gemeinsam werden wir eine kleine Zählerkomponente entwickeln, mit einer Anzeige des aktuellen Wertes sowie einem Plus- und Minus-Button. Die **Abbildung 1** zeigt den universellen Einsatz unserer Implementierung.

Abb. 1: Demo der fertigen Zählerkomponente

Zum Start erstellen wir uns eine Datei mit dem Namen package.json. Den Inhalt entnehmen Sie Listing 1.

Listing 1: Inhalt der package.json-Datei

```
{
  "name": "windows-developer-web-component
  "version": "1.0.0",
  "scripts": {
    "start": "browser-sync start --server
  },
  "dependencies": {},
  "devDependencies": {
    "browser-sync": "^2.26.7"
  }
}
```

Die wichtigste Angabe hier ist der Startbefehl. Dieser startet die Anwendung browser-sync. Diese Anwendung bietet uns zwei Dinge: Zum einen erhalten wir einen

Webserver, der später unsere Web Component ausliefern wird, sodass sie im Browser angezeigt werden kann. Zum anderen bemerkt die Anwendung, wenn innerhalb des Entwicklungsordners eine Änderung stattfindet, und lädt automatisch den Browser neu. So können wir sehr schnell unsere Änderung ansehen, nämlich in dem Moment, in dem wir im Codeeditor auf Speichern drücken.

Damit es wie gewünscht funktioniert, erstellen wir zunächst noch den Ordner src. Ist das geschehen, öffnen wir eine Kommandozeile im Ordner, in dem auch die package.json-Datei liegt. In der Kommandozeile führen wir das Kommando npm install aus, um browser-sync herunterzuladen. Danach führen wir npm start aus, um die Anwendung zu starten. Zu sehen ist natürlich noch nichts. Das wollen wir mit dem nächsten Schritt ändern. Dazu legen wir im Ordner src die Datei index.html an. Den Inhalt der Datei finden Sie in Listing 2.

Listing 2: Inhalt der Datei src/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Windows Developer Web Component
    <img src="
  </head>
  <body>
  </body>
</html>
```

Neben den HTML-Dokument-typischen Angaben sehen wir auch die Angabe eines <*script*>-Tags, was auf eine Datei *counter.js* verweist. Hinweis: Diese Datei haben wir noch nicht erstellt. Interessant ist die Angabe von type=*"module"*. Sie teilt dem Browser mit, dass er die Datei als JavaScript-Modul laden soll. Auch hierbei handelt es sich um eine mittlerweile native Funktion des Browsers. Früher hätte man sein JavaScript als CommonJS- oder AMD-basierte Module bereitgestellt.

Heute reicht die Angabe im Browser aus, um jede JavaScript-Datei als Modul zu laden. Module ermöglichen es uns, unsere JavaScript-Programme in kleinere Teile zu zerstückeln und nur bei Bedarf zu laden.

Gut! Wenn wir jetzt im Google Chrome den URL http://localhost:3000 aufrufen, sollten wir zumindest eine weiße Seite mit dem Titel „Windows Developer Web Components Demo“ sehen. Ist das nicht der Fall, prüfen Sie zuerst, ob in Ihrer Kommandozeile beim Ausführen von *npm start* ein Fehler angezeigt wird. Falls nicht, prüfen Sie, ob auf Ihrer Maschine *browser-sync* ggf. auf einem anderen Port gestartet wurde. Auch das ist in der Logausgabe der Kommandozeile ersichtlich.

COUNTER.JS – DAS HERZSTÜCK UNSERER KOMPONENTE

Mit den zwei Dateien zuvor haben wir das Grundgerüst unserer Anwendung geschaffen. Zum einen eine kleine Entwicklungsumgebung, die sich automatisch bei Änderung an Dateien diese neu lädt. Als auch eine kleine Demo-Applikation mit der *index.html*. Denn diese benötigen wir später für unsere eigentliche Komponente nicht. Sie dient nur dazu, dass wir unsere eigene Komponente sehen, um sie besser entwickeln zu können. Als Nächstes legen wir im Ordner *src* die Datei *counter.js* an. In dieser Datei werden wir unsere Web Component entwickeln. In Listing 3 finden wir das Grundgerüst einer jeden Web Component.

Listing 3: Inhalt der Datei src/counter.js

```
1  class MyCounter extends HTMLElement {
2    constructor() {
3      super();
4    }
5  }
6
7  window.customElements.define('my-counter',
```

Die Datei startet mit der Definition der Klasse *MyCounter*. Sie leitet von einer Klasse *HTMLElement* ab. Die Klasse *HTMLElement* wird vom Browser zur Verfügung gestellt. Sie dient als Basisklasse für jedes im Browser befindliche HTML-Element, daher auch für unsere. Übrigens, wollten wir z. B. ein spezielles Eingabefeld entwickeln, könnten wir auch von der Klasse *HTMLInputElement* ableiten. Im MDN (https://developer.mozilla.org/en-US/docs/Web/API) findet sich eine ganze Reihe von Interfaces, von denen wir unsere eigene Komponente ableiten können. Damit wir die Vererbungskette ordnungsgemäß einhalten, nutzen wir einen *super()*-Aufruf in unserem Konstruktor, ähnlich wie *base()* bei C#.

Am Ende von Listing 3 findet sich das erste API unseres Standards Custom Elements wieder. Es handelt sich hier um die Definition eines neuen HTML-Elements für den Browser. Daher ist dieses API auf dem Objekt *window* zu finden. Die Methode *define()* definiert das neue Element. Der erste Parameter gibt den Namen des Elements an. Der zweite Parameter ist eine Konstruktor-Funktion, in dem Fall unsere Klasse *MyCounter*.

Der Typ von *customElements* ist eine *CustomElementRegistry*. Diese ist global für das aktuelle Browsertab. Neben der Definition von neuen Elementen, könnte man prüfen, ob ein bestimmtes HTML-Element registriert wurde, in dem man bspw. *customElements.get('name-des-elements')* aufruft. Entweder man erhält die Konstruktor-Funktion oder null. Alternativ kann mit *customElements.whenDefined('name-des-elements')* gewartet werden, bis ein HTML-Element mit dem gewünschten Namen zur Verfügung steht. Die Methode gibt ein Promise zurück, das erfüllt wird, sobald das HTML-Element registriert wurde. Um das Element zu nutzen, können wir im <*body*>-Bereich in der index.html

unser Element via *<my-counter></my-counter>* aufrufen. Es ist zu beachten, dass Web Components generell ein schließendes Tag benötigen.

HTML TEMPLATE ZUR VISUALISIERUNG

Durch das Speichern im Codeeditor hat sich unser Browser automatisch neu geladen. In den DevTools von Chrome sehen wir zwar unser HTML-Element, allerdings ist visuell auf der Seite immer noch nichts sichtbar. Es fehlen schließlich noch unser HTML Template und der Shadow DOM. In Listing 4 finden wir ein erstes kleines Template. Dieses wird noch vor der Definition unserer Klasse abgelegt.

Listing 4: HTML Template

```
const template = document.createElement('t

template.innerHTML = `
  </p>
<style>
    .counter-container {
      --default-height: var(--height, 50px

      width: calc(var(--default-height) *
      height: var(--default-height);
      display: flex;
    }

    .counter-container > div {
      color: black;
      font-size: 2.2em;

      display: flex;
      align-items: center;
      justify-content: center;
    }

    .button {
      padding: 1rem;
      border: 1px solid black;
    }

    .value {
      margin: 0 1rem;
    }
  </style>
<p>

  <slot name=&quot;header&quot;>
```

```
35      </p>
36    <h1>My Counter</h1>
37    <p>
38      </slot>
39
40      </p>
41    <div class=&quot;counter-container&quot;><
42      </p>
43    <div class=&quot;button decrement&quot;>-<
44    <p>
45
46      </p>
47    <div class=&quot;value&quot;>
48          <slot name=&quot;value-prefix&quot;>
49          <span class=&quot;value-display&quot
50          <slot name=&quot;value-postfix&quot;
51      </div>
52    <p>
53
54      </p>
55    <div class=&quot;button increment&quot;>+<
56    <p>
57      </div>
58    <p>`;
59
60    // class MyCounter extends HTMLElement ...
```

Bitte beachten Sie, dass zur besseren Darstellung im Artikel das Styling der Komponente nicht dem der **Abbildung 1** entspricht, sondern abgeändert wurde. Das originale Styling finden Sie im GitHub Repository (https://github.com/thinktecture-labs/windows-developer-web-components). Es ist daher auch zu empfehlen, das Styling aus dem Repository zu übernehmen.

Schauen wir uns Listing 4 genauer an. In der ersten Zeile wird via *document.createElement()* ein HTML Template erstellt. Über den Zugriff auf *innerHTML* können wir unser Template definieren. Hier können wir neben dem eigentlichen Markup eben auch CSS-Stile hinterlegen, um unsere Komponente zu stylen. Im Kasten „Struktureller Aufbau von Web Components“ finden sich weitere Informationen über den strukturellen Aufbau von Web Components. Im CSS selbst wird zunächst die Variable *–default-height* definiert. Sie prüft, ob eine Variable *–height* gesetzt wurde. Falls nicht, wird 50px als

Standardwert übernommen. Auf Basis dieser Variable wird die Größe unserer Komponente bestimmt. Im Originalstyling hat die Variable Auswirkung auf weitere Elemente innerhalb der Web Component.

Spannend wird das HTML Markup, da wir hier ein Element sehen, das wir nur innerhalb eines HTML Templates verwenden können. Es handelt sich hierbei um das Element <*slot*>. Es dient als Platzhalter für Inhalt, den wir später von außen, also in unserer *index.html* bestimmen können. Wird der Inhalt nicht von außen überschrieben, wird der Inhalt vom <*slot*>-Element selbst gerendert, in diesem Fall ein

<*h1*>-Tag. Der name erlaubt uns, gezielt einen bestimmten Slot anzusprechen, den wir überschreiben. Dieses Konzept ist in anderen Frameworks oft unter den Begriffen Transclusion, Content Projection oder Higher-Order Component zu finden. Das weitere Markup ist gewohntes HTML.

Struktureller Aufbau von Web Components

Je nach Framework sind Webentwickler es gewohnt, Code von Template und Styling zu trennen, in zwei oder mehrere Dateien. Im einfachsten Fall von Web Components, wie in diesem Artikel dargestellt, befindet sich alles innerhalb einer Datei. Bei größeren oder komplex gestylten Web Components kann diese Datei daher recht groß werden. Auch der Einsatz von CSS-Prozessoren ist meist schwierig, wenn sich das CSS innerhalb von JavaScript-Dateien wiederfindet. Möchte man bereits erste produktive Web Components entwickeln, empfiehlt sich der Einsatz von Frameworks, die sich mittlerweile auf Web Components spezialisieren. Herausstechend ist hier Stencil.js (https://stenciljs.com). Stencil.js lagert das Styling in eine separate Datei aus und hat von Haus aus Unterstützung von verschiedenen CSS-Prozessoren, sodass wir hier auf gewohntem Weg

entwickeln können. Des Weiteren bietet Stencil.js einigen syntaktischen Zucker, um wiederkehrende Aufgaben in Web Components einfacher zu gestalten, bspw. das Binden von Daten in die UI oder das Reagieren auf Events wie einen Button-Klick.

SHADOW DOM

Falls wir zwischenzeitlich unseren Codeeditor gespeichert haben, hat der Browser zwar ein Reload ausgeführt, dennoch ist nichts zu sehen. Das hat auch einen guten Grund: Wir haben zwar ein Template definiert, aber wir nutzen es noch nicht. Das wollen wir ändern. Dazu benötigen wir den Inhalt aus Listing 5. Dieser wird unmittelbar nach dem *super()* im Konstruktor unserer Klasse eingefügt.

Listing 5: Shadow DOM

```
1  constructor() {
2    super();
3
4    this.shadow = this.attachShadow({ mode: '
5    this.shadow.appendChild(template.content.
6  }
```

Der wohl wichtigste Aufruf ist *attachShadow()*. Hierdurch wird unserer Komponente ein Shadow DOM angehängt, in das wir etwas platzieren dürfen. Die Angabe des ersten Parameters ist obligatorisch und gibt den Modus des Shadow DOM an. Hier im Beispiel nutzen wir den Modus *open*. Alternativ gäbe es auch eine Variante *closed*.

Der Modus bestimmt, ob wir via JavaScript von außen auf das Shadow DOM zugreifen dürfen. Jede Web Component hat ein JavaScript Property shadowRoot. Der Modus bestimmt, ob wir beim Zugriff auf dieses Property den tatsächlichen Shadow DOM Tree erhalten (*open*) oder ob wir beim Zugriff null erhalten (*closed*). Jetzt stellt sich die Frage, warum wir uns von außen Zugriff erlauben wollten. Gerade im Hinblick auf die Entwicklung

größerer Komponentenbibliotheken kann es durchaus sinnvoll sein, Zugriff auf das Shadow DOM kleinerer interner Komponenten zu erlauben. Allerdings gilt es dann, den Zugriff auf das Shadow DOM zu sperren bei den Komponenten, die später von den Entwicklern eingesetzt werden. Übrigens, JavaScript wäre nicht JavaScript, wenn man nicht über kleine Tricks auch beim closed-Modus an den Shadow DOM Tree kommen könnte. Eine Suchmaschine Ihrer Wahl hilft beim Auffinden solcher kleinen Gemeinheiten.

Nachdem wir nun einen Shadow DOM Tree erzeugt haben, fügen wir mit der letzten Zeile im Listing 5 unser HTML Template ein, in dem wir es klonen. Der Parameter gibt an, ob ein *deepClone* stattfinden soll, sprich der komplette Baum kopiert wird. Ansonsten erhalten wir nur den Wurzelknoten. Wenn wir jetzt unseren Codeeditor speichern, werden wir zum ersten Mal ein Rendering unserer Komponente sehen. Zugegebenermaßen nicht ganz so hübsch mit dem CSS direkt aus dem Artikel, allerdings umso hübscher mit dem CSS aus dem Repository (https://github.com/thinktecture-labs/windows-developer-web-components).

ZURÜCK IN DIE ZUKUNFT: PLAIN OLD JAVASCRIPT

Auch wenn unsere Komponente gerendert wird, können wir noch nicht mit ihr interagieren. Um unserer Komponente Leben einzuhauchen, benötigen wir den Code aus Listing 6.

Listing 6: Event Handling

```
constructor() {
  // super & shadow

  this.decrementButton = this.shadow.query
  this.incrementButton = this.shadow.query
  this.valueDisplay = this.shadow.querySel

  this.decrementButton.addEventListener('c
  this.incrementButton.addEventListener('c
}

connectedCallback() {
  this.render();
}

get value() {
  return +this.getAttribute('value') || 0;
}

set value(v) {
  this.setAttribute('value', v);
  this.render();
}

increment() {
  this.value++;
  this.dispatchEvent(new CustomEvent('valu
}

decrement() {
  this.value--;
  this.dispatchEvent(new CustomEvent('valu
}

render() {
  this.valueDisplay.textContent = this.val
}
```

Zunächst holen wir, fast schon etwas altertümlich in Hinblick auf moderne SPA-Frameworks, über den *querySelector()* alle Elemente, die wir benötigen. Das wären beide Buttons sowie das Element, das den aktuellen Wert der Komponente angeben soll. Danach registrieren wir zwei Event Handler via *addEventListener()*. Beim Klick auf die jeweiligen Buttons soll eine Funktion ausgeführt werden, nämlich *decrement()* und *increment()*. Danach folgt die Implementierung der Lifecycle-Methode *connectedCallback()*. Sie wird immer dann aufgerufen, wenn der Browser die Komponente im DOM platziert. Sie kann daher für erste Initialisierungen genutzt werden. In unserem Fall ein Aufruf der Methode *render()*, die

wiederum den aktuellen Wert *this.value* in unser HTML-Element *valueDisplay* schreibt. Neben dem *connectedCallback()* existiert auch ein *disconnectedCallback()*, sprich, wenn die Komponente wieder aus dem DOM entfernt wird, bspw. bei einer konditionalen Anzeige.

Im nächsten Schritt wird ein Getter/Setter *value* definiert. Dieser schreibt und liest den aktuellen Wert des HTML-Attributs *value* aus. Beim Lesen des Wertes wenden wir einen kleinen JavaScript-Trick an. HTML-Attribute sind generell als Strings abgebildet. Da wir aber einen numerischen Wert benötigen, nutzen wir das Plus-Zeichen vor dem Aufruf von *getAttribute()* zur Konvertierung eines Strings zu einer Nummer. Alternativ kann hier auch die Methode *parseInt* genutzt werden. Beim Setzen des Wertes wird zusätzlich die Methode *render()* aufgerufen, um den gesetzten Wert zur Anzeige zu bringen. Im Kasten „JavaScript Properties vs. HTML-Attribute“ sind nützliche Informationen zu JavaScript Properties und HTML-Attributen enthalten.

Zu guter Letzt werden unsere *increment()*– und *decrement()*-Methoden definiert. Sie erhöhen bzw. verringern den Wert von *value*. Danach schicken sie ein Event nach außen, das meldet, dass sich der Wert geändert hat. So kann der Nutzer dieser Komponente darauf reagieren, wenn sich der Wert der Komponente ändert. Dazu wird ein *CustomEvent* erzeugt. Der erste Parameter ist der Name des Events, der zweite dient zur Übermittlung von weiteren Daten. Wichtig hierbei ist, dass es sich um ein Objekt handeln muss, auf dem die Eigenschaft *detail* definiert ist. Hier – und nur hier – können wir einen einzelnen Wert oder ein Objekt übergeben.

JavaScript Properties vs. HTML-Attribute

Bei Web Components unterscheiden wir zwei verschiedene States. Den JavaScript State in Form von Properties und den HTML State in Form von HTML-Attributen. Mit dem in Listing 6 und Listing 7 gezeigten Pattern synchronisieren wir automatisch beide States. Das bedeutet, wenn wir im HTML das Attribut value setzen, ändern wir auch das Property value in JavaScript. Andersrum, wenn wir im JavaScript this.value setzen, ändern wir auch das HTML-Attribute value. Wir sind nicht gezwungen, den State zu synchronisieren. Standardelemente wie z. B. ein <input>-Element synchronisiert diesen State auch nicht.

Bei einfachen Werten können wir uns eine Synchronisierung erlauben, bei komplexen Werten eher nicht. Denn JavaScript Properties können natürlich nicht nur einfache Werte repräsentieren, sondern auch Arrays und Objekte. HTML-Attribute sind allerdings immer Strings. Würde man daher bspw. die Datenquelle einer tabellarischen Web Component als HTML-Attribut ausgeben, würden wir unter Umständen große Arrays als String darstellen und dem Browser übermitteln. Das zehrt an der Performance.

Das Pattern der Synchronisierung nennt sich Reflecting properties to attributes oder auch Reflected DOM Attributes.

ATTRIBUTE BEOBACHTEN

Ein kleines Detail fehlt uns noch. Wir können zwar auf unsere Buttons klicken und der angezeigte Wert unserer Komponente wird sich ändern. Ändern wir aber den Wert unserer Komponente im HTML via DevTools, wird nichts passieren. Warum das so ist, entnehmen Sie der Info in Kasten „JavaScript Properties vs. HTML-Attribute“. Um das Problem zu lösen, müssen wir noch eine dritte

Lifecycle-Methode anbinden. Diese finden wir in Listing 7 und sie kann bspw. direkt nach dem (aber nicht im) Konstruktor implementiert werden.

Listing 7: Lifecycle-Methode: attributeChangedC

```
1  static get observedAttributes() {
2    return [ 'value' ];
3  }
4
5  attributeChangedCallback(name, oldVal, new
6    if (oldVal === newVal) {
7      return;
8    }
9
10   if (name === 'value') {
11     this.value = newVal;
12   }
13 }
```

Es handelt sich hier um die Lifecycle-Methode *attributeChangedCallback()*. Sie hat drei Parameter. Den Namen des geänderten Attributs als String, sowie den alten und den neuen Wert, ebenfalls als Strings, da HTML-Attribute generell als String repräsentiert werden. Zusätzlich benötigen wir den statischen Getter *observedAttributes*. Dieser liefert ein String-Array mit allen Attributen, die der Browser überwachen soll, um bei einer Änderung die Methode *attributeChangedCallback()* aufzurufen.

Wie man dem Code der Lifecycle-Methode entnehmen kann, führt jeder Zugriff auf das Attribut zu einer Änderung. So kann es durchaus sein, dass der alte und der neue Wert übereinstimmen. Daher prüfen wir das zunächst und machen nichts, falls sich der Wert nicht ändert. Ändert sich allerdings der Wert und der Name des geänderten Attributs ist *value*, übernehmen wir diesen Wert.

Achtung: Wie der Kasten „JavaScript Properties vs. HTML-Attribute“ erläutert, nutzen wir hier das Pattern Reflected DOM Attributes. Durch das Setzen des Property

value setzen wir auch das HTML-Attribut *value*. Das hat zur Folge, dass der Browser, da sich das Attribut ändert, wieder *attributeChangedCallback()* aufruft. Ohne die Prüfung, ob sich der Wert geändert hat, würden wir hier in eine Endlosschleife geraten. Daher müssen wir bei der Anwendung des Patterns sehr vorsichtig sein, um keine Endlosschleife zu erzeugen. Damit wäre die Entwicklung unserer Komponente abgeschlossen.

ANPASSUNG VON INDEX.HTML

Um die Entwicklung abzurunden, nehmen wir noch eine kleine Änderung an der index.html vor. In Listing 8 finden wir den kompletten Inhalt der Datei, den wir auch genau so in unseren Editor übernehmen.

Listing 8: Inhalt der Datei index.html

```
<!DOCTYPE html>
<html lang=&amp;quot;en&amp;quot;>
  <head>
    <meta charset=&amp;quot;UTF-8&amp;quot;>
    <title>Windows Developer Web Component

    </p>
<style>
      my-counter.special {
        --height: 150px;
      }

      my-counter.special h1 {
        color: red;
      }
    </style>
<p>

    <script src=“counter.js&amp;quot; type=&amp;qu


</em></p>
&amp;nbsp;
<h1>Value pre- &amp; postfix</h1>
>
€

<script>
      var counter = document.querySelector

      counter.addEventListener('valueChang
    </script>
```

Wir sehen drei Änderungen: Es wurde etwas CSS definiert, unser Zähler ist zweimal definiert, einmal recht einfach und einmal mit Inhalt. Und zuletzt ein kleines Skript, das auf Werteänderungen des ersten Zählers reagiert. Um diese zu sehen, müssen wir die DevTools öffnen und die Konsole anschauen. Beim Klicken auf den ersten Zähler wird der aktuelle Wert auf der Konsole ausgegeben. Im CSS-Bereich sehen wir, dass wir eine Klasse special erstellt haben. Sie setzt die CSS-Variable *height* auf 150 Pixel. Daher sollte unser zweiter Zähler im Browser deutlich größer erscheinen als der erste. Wir sind also in der Lage, über CSS-Variablen das Aussehen unserer Komponente von außen zu beeinflussen. Wollten wir das eigentlich nicht vermeiden? Ja – und nein. Wir wollen vermeiden, dass uns jemand ungewollt CSS-Stile überschreibt, weil sie zufällig den gleichen Namen haben. Mit den CSS-Variablen bietet unsere Komponente ein API an, das wir bewusst nach außen geben, um das Aussehen zu ändern. Wir als Komponentenentwickler haben es also damit in der Hand, welche Aspekte unserer Komponente geändert werden dürfen.

Die zweite CSS-Angabe ändert die Textfarbe des <h1>-Tags der zweiten Zählerkomponente. Auch hier können wir uns die Frage stellen, warum wir Zugriff darauf haben, da doch die Überschrift im Shadow DOM enthalten ist. Das ist korrekt, solange die Elemente nicht über einen Slot überschrieben werden, was auch nur möglich ist, wenn wir es als Komponentenentwickler genau so definieren. Um einen Slot zu benutzen, müssen wir das Element oder die Elemente in das Tag unserer Web Component schreiben und das Attribut *slot* auf einen Namen setzen, den wir in unserer Web Component definiert haben. Ein Slot ändert die Zusammensetzung unseres Shadow DOM. Die geslotteten Elemente werden nämlich nicht in das Shadow DOM kopiert, sondern das Shadow DOM verweist auf die Elemente im Main Document DOM. Das bedeutet, dass die, in diesem

Beispiel drei, geslottete Elemente gar nicht zum Shadow DOM gehören, sondern zum Main Document DOM, und damit haben wir per CSS ganz normal Zugriff darauf (mit allen Vor- und Nachteilen).

In **Abbildung 2** sind die Chrome DevTools zu sehen. Dort sieht man zwei Dinge: Die überschriebenen Slots liegen außerhalb des Shadow DOM und im Shadow DOM ist ein Verweis auf ein <h1>-Tag zu sehen. Klickt man den Verweis an, landet man bei dem Element, das in den Slot gesetzt wurde. Versuchen Sie einmal via CSS die Buttonfarbe zu ändern. Auch mit !important werden Sie es nicht schaffen, da genau diese Elemente im Shadow DOM liegen und wir kein API dafür definiert haben.

Abb. 2: Verweis von Slots aus dem Shadow DOM in das Main Document DOM

FAZIT

Web Components sind eine tolle Sache, die uns in Zukunft immer mehr begleiten wird. Kleinere Komponenten lassen sich so gänzlich ohne Framework entwickeln. Wir haben aber auch gesehen, dass wir viel händisch machen müssen, was wir von SPA Frameworks als bereits vorhanden gewohnt sind, bspw. Datenbindung oder das Reagieren auf Events. Auch müssen wir uns Gedanken machen, wann wir welchen Teil der UI aktualisieren. In größeren Projekten dürfte das kaum praktikabel sein, da man sehr viel Boilerplate

benötigt. Hier lohnt es sich, einmal einen Blick in Stencil.js zu werfen, ein Framework speziell für Web Components mit sinnvollen Features wie CSS-Prozessoren, Dokumentationserzeugung oder Datenbindung, aber ohne weiteren Schnickschnack.

Wir haben auch gesehen, dass wir mit Web Components in der Lage sind, sehr bewusst zu definieren, was wir von außen zugänglich machen und was nicht, wir können hier ein richtiges API für den Entwickler definieren. Das bedeutet aber auch, dass wir unsere Komponenten gut dokumentieren müssen, da aktuell sehr viele Dinge stringbasiert sind bspw. Namen von Slots oder Attributen.

Alles in allem helfen uns die Standards rund um Web Components, native Komponenten zu entwickeln, die in jedem Browser genutzt werden können. Es gibt noch viel mehr zu entdecken, wie bspw. CSS Shadow Parts oder weitere Funktionen der HTML Slots und deren Styling via *::slotted()*. Viel Spaß beim Erkunden der Welt von Web Components!

Web Development Sessions auf der BASTA!

- Cross-Plattform-Apps für Desktop, Mobile und Web mit Electron, Cordova & Angular (https://basta.net/web-development/cross-plattform-apps-fuer-desktop-mobile-und-web-mit-electron-cordova-angular/)
- Angular-Architekturen für Geschäftsanwendungen mit Nx Monorepos (https://basta.net/web-development/angular-architekturen-fuer-geschaeftsanwendungen-mit-nx-monorepos/)

Share Post:

(http://pinterest.com/pin/create/button/?
(http://digg.com/submit?phase=2&url=https://basta.net/blog/einstieg-in-die-fabelhafte-welt-der-web-components/)
(http://www.facebook.com/sharer.php?u=https://basta.net/blog/einstieg-in-die-fabelhafte-welt-der-web-components/)
(http://www.linkedin.com/shareArticle?mini=true&url=https://basta.net/blog/einstieg-in-die-fabelhafte-welt-der-web-components/)
(http://reddit.com/submit?url=https://basta.net/blog/einstieg-in-die-fabelhafte-welt-der-web-components/)
(http://www.stumbleupon.com/submit?url=https://basta.net/blog/einstieg-in-die-fabelhafte-welt-der-web-components/)
(http://www.tumblr.com/share/link?url=https://basta.net/blog/einstieg-in-die-fabelhafte-welt-der-web-components/)
(http://www.pinterest.com/pin/create/button/?url=https://basta.net/blog/einstieg-in-die-fabelhafte-welt-der-web-components/&media=https://basta.net/wp-content/uploads/2022/03/Einstieg-in-die-fabelhafte-Welt-der-Web-Components-1.jpg)

(http://twitter.com/home?status=Einstieg%20in%20die%20fabelhafte%20Welt%20der%20Web%20Components%20-%20https://basta.net/blog/einstieg-in-die-fabelhafte-welt-der-web-components/)

‹ Oberflächendimensionen – alles nur UI? Mitnichten! (https://basta.net/blog/oberflaechendimensionen-alles-nur-ui-mitnichten/)

Die Realisierung von Microfrontends › (https://basta.net/blog/realisierung-von-microfrontends/)

BEHIND THE TRACKS

.NET Framework & C# (https://basta.net/net-framework-c/)
Visual Studio, TFS, C# & mehr

Agile & DevOps (https://basta.net/agile-devops/)
Best Practices & mehr

Web Development (https://basta.net/web-development/)
Alle Wege führen ins Web

Data Access & Storage (https://basta.net/data-access-storage/)
Alles rund um´s Thema Data

HTML5 & JavaScript (https://basta.net//html5-javascript/)
Leichtegewichtig entwickeln

User Interface (https://basta.net//user-interface/)
Alles rund um UI- und UX-Aspekte

Microservices & APIs (https://basta.net//microservices-apis/)
Services, die sich über APIs via REST und JavaScript nutzen lassen

Security (https://basta.net//
Tools und Techniken für sicherere Applikationen

Cloud & Azure (https://basta.net//cloud-azure-serverless/)
Cloud-basierte & Native Apps

Locations

BASTA! 2022

Die BASTA! ist die führende unabhängige Konferenz für Microsoft-Technologien im deutschsprachigen Raum.

LASSEN SIE SICH INSPIRIEREN

Nehmen Sie an inspirierenden Talks und ausführlichen Workshops teil, um Ihre .NET-Entwickler-Fähigkeiten auf die nächste Stufe zu heben.

SPONSOR WERDEN

Ihr erfolgreicher Auftritt auf der BASTA! 2022.

SPONSOR WERDEN (/SPONSOR-WERDEN/)

NEWSLETTER ANMELDEN

Stay Tuned! Registrieren Sie sich für unseren Newsletter und erhalten Sie regelmäßig News zur Konferenz!

JETZT ANMELDEN (/NEWSLETTER/)

(http://www.facebook.com/BASTAcon)
(https://twitter.com/bastacon)
(http://www.youtube.com/user/BastaConference)