2 Einführung in Web Components

In diesem Kapitel werden wir sehen, was Web Components grundsätzlich sind und welche Probleme sie lösen. Nach einem Vergleich bisheriger Techniken wie jQuery UI, Bootstrap und Handlebars geht es schrittweise darum, die neuen Techniken anhand von Codebeispielen kennenzulernen. Als Kapitelabschluss fassen wir das Gelernte in einem Grundgerüst zusammen, das wir für alle Komponenten einsetzen, die wir im weiteren Verlauf des Buches selbst erstellen.

2.1 Von Dokumenten zu Anwendungen: neue Ansätze für neue Anforderungen im Web

Webanwendungen bestehen üblicherweise aus drei Ebenen:

1. Struktur und Inhalte als HTML-Dokumente
2. Festlegungen zur Darstellung und Gestaltung als CSS
3. Anwendungslogik und Interaktionen als JavaScript-Code

Die Kombination aus HTML, CSS und JavaScript bildet eine solide Grundlage für die Erstellung von Webdokumenten und dank vieler neuer Elemente in HTML5 auch zunehmend für Webanwendungen.

Allerdings entwickeln sich die Anforderungen an Webanwendungen und die Bedürfnisse von Nutzern und Entwicklern schnell und stetig weiter. Daher werden oft neue Elemente für zusätzliche Funktionalität notwendig, zumeist neue UI-Elemente.

Dies ist keineswegs neu, und das Gros der Entwicklung besteht schon längst darin, existierende HTML-Elemente mittels CSS und JavaScript neu zu kombinieren und zu gestalten und damit neue Elemente zu schaffen. Beispiele sind etwa jQueryUI-Komponenten wie Schieberegler, Fortschrittsanzeigen oder Datepicker, die inzwischen als Teil von HTML5 spezifiziert wurden.

Elemente, die besonders oft benötigt werden, sollte man natürlich nicht jedes Mal neu entwickeln müssen. Stattdessen sollten bestehende Implementierungen wiederverwendbar gestaltet werden. Da solche Elemente meist aus HTML, CSS

und JavaScript bestehen und nur zusammengefasst genutzt werden können, sind die Komponenten ein Bündel aus mehreren Dateien, die entsprechend in die Anwendung eingebettet werden müssen.

Problematisch wird es, wenn eine Vielzahl solcher Komponenten eingesetzt wird, da es hier schnell zu Konflikten kommen kann. Zudem verlangen Komponenten oft einen bestimmten, häufig tief verschachtelten Aufbau von HTML-Elementen. Beispielsweise erfordert die Nutzung von jQueryUI Tabs folgenden Code:

```
<!-- Stylesheet und Scriptdateien einbinden -->
 <link rel="stylesheet" href="//code.jquery.com/ui/1.11.4/themes/basic/jquery-ui.
 css">
   <script src="//code.jquery.com/jquery-1.10.2.js"></script>
   <script src="//code.jquery.com/ui/1.11.4/jquery-ui.js"></script>
 <!-- Markup der Tabs definieren -->
 <div id="tabs">
   <ul>
     <li><a href="#tabs-1">Erster</a></li>
     <li><a href="#tabs-2">Zweiter</a></li>
     <li><a href="#tabs-3">Dritter</a></li>
   </ul>
   <div id="tabs-1">
     <p>Inhalt des ersten Tabs</p>
   </div>
   <div id="tabs-2">
     <p>Inhalt des zweiten Tabs</p>
   </div>
   <div id="tabs-3">
     <p>Inhalt des dritten Tabs</p>
   </div>
 </div>
 <!-- Tabs aktivieren -->
 <script>
   $("#tabs").tabs();
 </script>
```

Listing 2–1 *Beispielanwendung mit jQuery Tabs*

Dieses Beispiel zeigt, dass das HTML-Markup nicht nur an die Bedürfnisse einer Komponente angepasst werden muss (hier ein Div mit einer Liste und je einem Div mit passender ID pro Tab). Darüber hinaus kann auch die semantische Klarheit schnell verloren gehen, denn das Markup im Beispiel ist eher eine »Div-Suppe« als eine Tab-Komponente.

Besser wäre es, eigene Komponenten als neue HTML-Elemente bereitzustellen. Nehmen wir etwa das Beispiel für Tabs, dann könnte eine gute Lösung so aussehen:

```
<tab-box>
     <tab-content title="Erster">
```

```
            <p>Inhalt des ersten Tabs</p>
        </tab-content>
        <tab-content title="Zweiter">
            <p>Inhalt des zweiten Tabs</p>
        </tab-content>
        <tab-content title="Dritter">
            <p>Inhalt des dritten Tabs</p>
        </tab-content>
    </tab-box>
```

Listing 2–2 *Beispielanwendung mit Web Components*

Hier ist das Markup zwar auch spezifisch für die Komponente, aber leichter zu lesen und semantisch auch eindeutiger. Wichtig ist hierbei natürlich, dass ein Browser, der diese Elemente nicht unterstützt, dennoch eine Darstellung der Inhalte bietet. Dies ist der Fall, vergleichbar mit der jQueryUI-Lösung, da ein Browser unbekannte Elemente im Zweifel ähnlich wie Div-Elemente behandelt.

Web Components bieten genau diese Lösung in Form von vier Webstandards, mit denen man Komponenten entwickeln kann, die genau wie HTML-Elemente funktionieren und genutzt werden können. In den folgenden Abschnitten lernen wir alle vier Standards und ihren Einsatz kennen und erarbeiten uns dabei eine Grundlage für die Entwicklung mit Web Components.

Hinweis

Die populären Frameworks versuchen, Web Components durch ähnliche Funktionalität zu emulieren. Angular.js beispielsweise durch Directives, React.js mit Components. In diesem Buch werden wir diese Lösungen jedoch nicht weiter betrachten, sondern uns auf die Standard-Web-Components und Polymer beschränken.

2.2 Web-Component-Standards im Überblick

2.2.1 Template-Element

Ein neues HTML-Element, `<template>`, ersetzt Template-Lösungen wie Handlebars, Mustache oder `<script type="text/plain">` durch eine standardisierte Lösung, um Inhalte im HTML-Markup für die spätere Verwendung vorzubereiten und wiederzuverwenden. Dies führt auch zu teils deutlich besserer Performance gegenüber vorherigen Lösungen.

2.2.2 Shadow DOM

Das Shadow DOM erlaubt die Isolierung von Teilen des HTML-Markups, wodurch unerwünschte Nebenwirkungen von CSS oder JavaScript begrenzt werden können.

2.2.3 Custom-Elements

Der Custom-Elements-Standard erlaubt es Entwicklern, eigene HTML-Elemente mit zugehörigem, eigenen Verhalten zu definieren und zu verwenden oder bestehende Elemente zu erweitern.

2.2.4 HTML-Imports

HTML-Imports ermöglichen es mittels einer neuen Direktive für `<link>`-Elemente, nicht nur CSS in ein HTML-Dokument einzubinden, sondern auch ganze HTML-Elemente. Dies ermöglicht die einfache Nutzung eigener Komponenten, aber auch die Wiederverwendung von Komponenten von anderen Entwicklern.

2.3 Templating

Ein Merkmal jeder Webanwendung ist die Notwendigkeit, HTML dynamisch aus aktuellen Daten der Anwendung zu erzeugen.

Da die Erzeugung von HTML direkt aus JavaScript heraus (z. B. mittels `document.createElement`) schnell unübersichtlich wird und die Trennung von Markup und Anwendungslogik verletzt, kommen üblicherweise sogenannte Template-Engines zum Einsatz. Sie können die Vorlagen als HTML separat definieren und innerhalb der Anwendungslogik leicht mit Inhalt versehen und einbinden.

2.3.1 Konventionelle Templating-Systeme

Nehmen wir zum Beispiel Handlebars. Handlebars-Templates werden in `<script type="text/x-handlebars-template">`-Elementen definiert, in denen HTML mit Platzhaltern definiert wird, die dann später mit Daten gefüllt und in das aktive HTML-Dokument eingebunden werden.

Ein einfaches Beispiel sieht so aus:

```
<!doctype html>
 <html>
 <body>
   <script id="tpl" type="text/x-handlebars-template">❶
     <h1>{{title}}</h1> ❷
     <div class="body">{{body}}</div>
   </script>
   <script>
     var source = document.getElementById("tpl").textContent,
         template = Handlebars.compile(source); ❸
     document.body.innerHTML += template({  ❹
         title: 'Hallo Welt',
         body: 'Ich komme aus einem Template'
```

```
    });
  </script>
</body>
</html>
```

Listing 2–3 *Beispielanwendung mit Handlebars-Template*

❶ Templates werden in `<script>`-Blöcken mit entsprechendem `type` definiert.

❷ Mit {{…}} werden Platzhalter definiert, die später durch konkreten Inhalt ersetzt werden sollen.

❸ Mittels `compile` wird der Inhalt des Templates in eine Funktion umgewandelt, die Daten für die Platzhalter als Parameter entgegennimmt.

❹ Schließlich kann die Funktion aus Schritt 3 mit konkreten Daten aufgerufen werden, um das finale Markup zu erhalten und es einzufügen.

2.3.2 Das Template-Element

Einer der vier Web-Components-Standards trägt den Namen »Templates« und adressiert das im vorherigen Abschnitt behandelte Problem. Dank des neuen `<template>`-Elements ist es ohne externe Bibliotheken oder Tools möglich, Templates zu definieren und zu nutzen. Wir werden später sehen, dass Templating große Unterschiede in der Performance aufweisen kann, je nachdem, wie mit dem DOM gearbeitet wird.

Das Template-Element kann und wird von den Browser-Engines früher oder später optimiert werden können und ist bereits jetzt unter den schnelleren Kandidaten für DOM-Manipulation.

Hier ein Beispiel zur Verwendung:

```
<!doctype html>
 <html>
 <body>
   <template id="tpl"> ❶
     <h1 class="title"></h1>
     <div class="body"></div>
   </template>
   <script>
     var tplContent = document.getElementById("tpl").content; ❷
     var node = tplContent.importNode(true); ❸
     node.querySelector("h1").textContent = "Hallo Welt"; ❹
     node.querySelector("div").textContent = "Ich komme aus einem Template";
     document.body.appendChild(node); ❺
   </script>
 </body>
 </html>
```

Listing 2–4 *Verwendung von <template>*

❶ Das Template wird als HTML innerhalb eines `<template>`-Elements definiert.

❷ Mit der DOM-API lässt sich das Template-Element auswählen, und über dessen `content`-Attribut kann auf das `DocumentFragment` innerhalb des Templates zugegriffen werden.

❸ Der Inhalt des Templates gehört nicht zum Anwendungsdokument, da es sich um ein sogenanntes Dokumenten-Fragment handelt. Ein solches Dokumenten-Fragment ist eine DOM-Struktur, die vom Dokument selbst losgelöst ist. Mittels `importNode` kann das Dokumenten-Fragment in das `document` der Anwendung importiert werden und ist nun als aktiver Teil des DOM-Baumes verfügbar. Der erste Parameter ist das zu importierende Dokumenten-Fragment, der zweite Parameter legt fest, ob das gesamte Fragment oder nur die direkten Kindelemente importiert werden sollen.

❹ Das DOM-Element, das wir aus dem Template erzeugt haben, kann nun mit üblichen DOM-APIs manipuliert und befüllt werden.

❺ Um den Inhalt in die Anwendung zu bringen, genügt ein entsprechender DOM-Aufruf (z. B. `appendChild`).

Ein Template-Element ist dabei einem `DocumentFragment` gleichzusetzen. Das bedeutet, dass die Inhalte in einem Template vom eigentlichen Dokument losgelöst und inaktiv sind, bis sie durch Aufruf von DOM-Methoden wie `appendChild` in das aktive Dokument eingefügt werden. Eine Konsequenz daraus ist beispielsweise, dass auch Skripte innerhalb von Templates erst dann ausgeführt werden, wenn das Template in das aktive Dokument eingefügt wird.

Hinweis

Das `DocumentFragment`-Konstrukt ist eine leichtgewichtigere Alternative zum vollständigen `document`-Objekt und ermöglicht schnelleres Klonen von Elementen.

2.3.3 Performance von Templates

Die Performance von unterschiedlichen Template-Lösungen hängt neben der Implementierung der eigentlichen Templating-Logik auch maßgeblich von der Effizienz der Methode ab, mit der die Template-Inhalte in das Dokument eingefügt werden. Hierbei kann es enorme Unterschiede geben. In diesem Abschnitt schauen wir kurz auf folgende Varianten:

1. Manipulation mittels `innerHTML`
2. Erstellung neuer Elemente mittels `createElement`
3. Nutzung von `createDocumentFragment`
4. Einsatz des `template`-Elements

Die Ergebnisse variieren von Browser zu Browser und sogar zwischen unterschiedlichen Browser-Versionen, aber die generelle Tendenz sieht in etwa so aus:

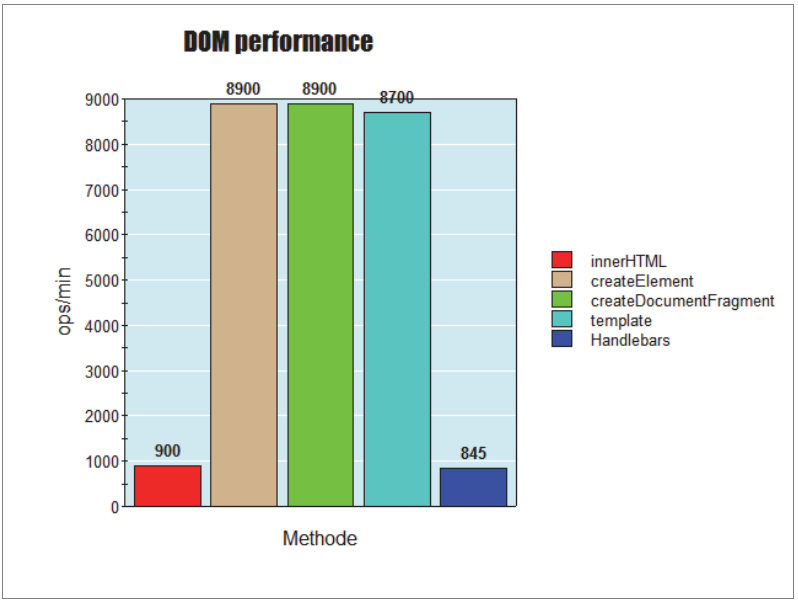

Abb. 2–1 *DOM-Performance beim Einfügen von Elementen*

Auch wenn die absoluten Zahlen ungenau sind und vom System und Browser abhängen, so sind die Größenordnungen hier doch beachtlich und relativ konstant über alle getesteten Browser hinweg.

Hinweis

Für das `<template>`-Element sind nur Werte aus Firefox und Chrome gemessen worden, alle anderen Methoden wurden in IE 11, Chrome, Firefox und Safari gemessen.

Das bedeutet, dass `createElement`, `createDocumentFragment` und `Template` ähnlich leistungsfähig sind, während `innerHTML` um den Faktor 10 langsamer ist. Einige Template-Engines, wie etwa Handlebars, nutzen `innerHTML` und zeigen daher eine vergleichbar geringe Leistung.

2.4 CSS für Komponenten

Ein Template kann außerdem CSS enthalten, um die Darstellung von Elementen beim Einfügen in das aktive Dokument anzupassen, etwa so:

```
<!doctype html>
 <html>
```

```
<body>
  <template id="tpl">
    <style>
      h1 { ❶
        color: red;
      }
    </style>
    <h1></h1>
    <div></div>
  </template>
  <h1>Zuerst war alles schwarz</h1> ❷
  <script>
    var tplContent = document.getElementById("tpl").content;
    var node = tplContent.importNode(true);
    node.querySelector("h1").textContent = "Doch dann kam das Template";
    node.querySelector("div").textContent = "Und fast alles wurde rot";
    document.body.appendChild(node);
  </script>
</body>
</html>
```

❶ Templates können auch `<style>`-Blöcke oder `<link>`-Elemente, die auf CSS verweisen, enthalten.

❷ Bestehende Elemente außerhalb des Templates können dabei versehentlich durch CSS beeinflusst werden.

Die Verwendung von globalem Styling wie im letzten Beispiel, in welchem die Farbe aller `<h1>`-Elemente auf Rot gesetzt wurde, ist daher problematisch, da dies ungewollt das ganze Dokument beeinflussen kann. IDs sind ebenfalls ungeeignet, da sie im gesamten Dokument einmalig sein müssen. Beim mehrfachen Klonen und Einfügen in das Dokument werden sie aber dupliziert, was zu Konflikten führt.

Eine mögliche Lösung ist die Verwendung von Klassen als CSS-Selektor:

```
<!doctype html>
 <html>
 <body>
   <template id="tpl">
     <style>
       h1.hot {
         color: red;
       }
     </style>
     <h1 class="hot"></h1>
     <div></div>
   </template>
   <h1>Zuerst war alles schwarz</h1>
   <script>
     var tplContent = document.getElementById("tpl").content;
     var node = tplContent.importNode(true);
     node.querySelector("h1").textContent = "Doch dann kam das Template";
```

```
    node.querySelector("div").textContent = "Und alles schien friedlich";
    document.body.appendChild(node);
  </script>
</body>
</html>
```

Damit ist zwar das ursprüngliche Problem umgangen, nicht aber gelöst worden, denn das CSS aus dem Template gilt weiterhin für das gesamte Dokument. Deshalb könnte eine Verwendung der Klasse `hot` im umgebenden Dokument wieder zu unerwünschten Nebenwirkungen führen.

Denken wir zurück an unser ursprüngliches Problem: die Verwendung und Verschachtelung von Komponenten, die von der Anwendung, in der sie genutzt werden, möglichst unabhängig sind. Der Workaround mit Klassen ist nicht ausreichend robust. Falls eine andere Komponente die Klasse `hot` ebenfalls benutzt, aber andere Style-Eigenschaften verwendet, kommt es zu Konflikten und unerwünschten Darstellungsproblemen.

2.4.1 Isolierte Styles dank Shadow DOM

Erfreulicherweise nimmt sich der zweite Standard aus der Web-Components-Standardfamilie, Shadow DOM, dieses Problems an. Das Shadow DOM bietet uns eine Isolationsschicht für Inhalte aus Templates an, mit der solche Konflikte vermieden werden können.

Dabei kann man sich das Shadow DOM als eine Art eigenes Dokument vorstellen, das in das Dokument der Anwendung so integriert wird, dass sein Inhalt für das äußere Dokument unsichtbar und (nahezu) unerreichbar ist. Durch diese Beschaffenheit wirken weder CSS-Selektoren noch JavaScript-DOM-Methoden auf den Inhalt eines Shadow DOM. Das bedeutet, dass keine CSS-Konflikte zwischen unserer Anwendung und Inhalten innerhalb eines Shadow DOM auftreten und dass wir nicht versehentlich von außen mittels `removeChild` oder `querySelector` und ähnlichen Methoden auf solche Inhalte zugreifen und diese manipulieren können. Es sei denn, wir haben direkten Zugriff auf die Shadow-DOM-Instanz oder nutzen eine der Techniken, um die Grenze des Shadow DOM gezielt zu überschreiten.

Warnung

Die Isolation des Shadow DOM ist kein Sicherheitsfeature. Es dient lediglich der Isolation gegen **versehentliche** Manipulationen, kann aber gezielt durchbrochen werden. Eine Verwendung als Sicherheitsbarriere gegen unerwünschte externe Skripte oder Zugriffe auf das Dokument ist daher nicht wirksam!

Ein Shadow-DOM-Objekt gehört immer zu einem Objekt innerhalb des Anwendungsdokuments und wird mit der `createShadowRoot`-Methode erzeugt. Es verhält sich dann wie ein `DocumentFragment` und kann Elemente in sich aufnehmen.

Ein Beispiel für die Nutzung des Shadow DOM:

```
<!doctype html>
 <html>
 <body>
   <template id="tpl">
     <style>
       h1 {
         font-style: italic;
         color: darkgreen;
       }
     </style>
     <h1></h1>
   </template>

   <h1>Alt und langweilig</h1>

   <script>
     var tplContent = document.getElementById("tpl").content;
     var node = document.importNode(tplContent, true);
     node.querySelector("h1").textContent = "Neu und aufregend"; ❶

     var container = document.createElement("div"), ❷
         shadowContainer = container.createShadowRoot(); ❸

     shadowContainer.appendChild(node); ❹
     document.body.appendChild(container); ❺
   </script>
 </body>
 </html>
```

❶ Zunächst importieren und modifizieren wir das Template wieder.

❷ Dann erzeugen wir ein neues Element, in das wir unser Shadow DOM einfügen wollen.

❸ Innerhalb dieses `container`-Elements erzeugen wir das Shadow DOM mit `createShadowRoot()`.

❹ Der Inhalt des Templates wird in das Shadow DOM eingefügt (statt wie zuvor direkt in das Anwendungsdokument).

❺ Durch Einfügen des Container-Elements wird selbiges zusammen mit seinem Shadow DOM in das Anwendungsdokument eingefügt.

Das Ergebnis zeigt, dass die CSS-Eigenschaften nur innerhalb des Shadow DOM wirken:

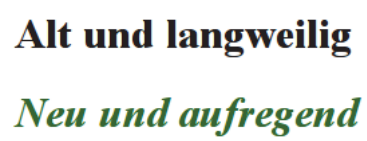

Abb. 2–2 *Styles wirken nur innerhalb des Shadow DOM*

Betrachten wir nun das erzeugte DOM:

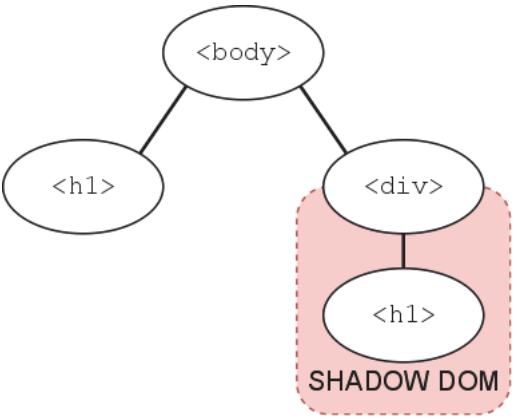

Abb. 2–3 *Shadow DOM Baumstruktur*

Nun haben wir den Inhalt aus unseren Templates isoliert und Konflikte in unserer Anwendung ausgeschlossen.

2.4.2 Eingefügte Inhalte, Shadow und CSS

Das Shadow DOM kontrolliert die Darstellung eines Elements komplett. Das heißt, im folgenden Beispiel wird der Browser nur den Text »Stille Wasser sind tief.« anzeigen:

```
<template>
   Stille Wasser sind tief.
 </template>
 <div>Hier lauern Drachen.</div> ❶
 <script>
 var tplContent = document.querySelector('template').content
 var container = document.querySelector('div')
 var root = container.createShadowRoot()
 root.appendChild(document.importNode(tplContent, true)) ❷
 </script>
```

❶ Dieses `<div>` enthält bereits Text, wird aber später ein eigenes Shadow DOM erhalten.

❷ Wir fügen den Inhalt des Templates (den Text »Stille Wasser sind tief.«) in das Shadow DOM ein.

Da der existierende Text (»Hier lauern Drachen.«) nicht im Shadow DOM existiert, wird er nicht dargestellt.

Es gibt aber einen Weg, existierende Inhalte in ein Shadow DOM zu »projizieren«. Das bedeutet, dass diese Inhalte im Dokument verbleiben und nicht Teil

des Shadow DOM werden, aber innerhalb des Shadow DOM angezeigt und auch über JavaScript innerhalb des Shadow DOM angesprochen werden können.

Im folgenden Beispiel erscheint sowohl der Text »Hier lauern Drachen.« als auch der Text »Stille Wasser sind tief.«. Ersterer wird an der Stelle im Shadow DOM angezeigt, an der das `<content>`-Element steht.

```
<template>
   Stille Wasser sind tief.
   <content></content>
 </template>
 <div>Hier lauern Drachen.</div>
 <script>
 var tplContent = document.querySelector('template').content
 var container = document.querySelector('div')
 var root = container.createShadowRoot()
 root.appendChild(document.importNode(tplContent, true))
 </script>
```

Wenn wir das `<content>`-Element nutzen, um Inhalte vom Anwendungsdokument in das Shadow DOM zu projizieren, verhalten sich diese Inhalte beim Stylen mit CSS nicht unbedingt erwartungsgemäß:

```
<!doctype html>
 <html>
 <head>
   <style>
     h1 { color: red; } ❶
   </style>
 </head>
 <body>
   <template>
     <style>
       h1 {
         color: green; text-decoration: underline; ❷
       }
     </style>
     <content></content>
     <h1>Drinnen</h1>
    </template>

    <h1>Draußen</h1>
    <div> ❸
      <h1>Dazwischen</h1> ❹
    </div>
    <script>
       var root = document.querySelector('div').createShadowRoot()
       var tplContent = document.querySelector('template').content

       root.appendChild(document.importNode(tplContent, true))
    </script>
  </body>
  </html>
```

❶ Im Anwendungsdokument werden <h1>-Elemente rot.

❷ Im Shadow DOM werden <h1>-Elemente grün und unterstrichen.

❸ Dieses Element ist die Wurzel des Shadow DOM.

❹ Was geschieht hier? Das Elemente ist selbst nicht Teil des Shadow DOM, wird aber per <content> in das Shadow DOM projiziert.

Was geschieht mit <h1>-Elementen, die aus dem Anwendungsdokument stammen, aber im Shadow DOM dargestellt werden? Legen wir die Erwartung zugrunde, dass alles im Shadow DOM isoliert wird, dann sollte das <h1>-Element, das in das <content>-Element eingefügt wird, grün und unterstrichen sein. Aber dem ist nicht so:

Abb. 2–4 *Eine Überraschung beim Stylen von Elementen im Shadow DOM*

Es zeigt sich also, dass eingefügte Elemente trotz ihrer Verteilung auf das Shadow DOM die CSS-Eigenschaften des Anwendungsdokuments übernehmen. Zur Behebung dieses Umstands kann man das ::content-Pseudoelement nutzen, um explizit Elemente, die in ein Shadow DOM eingefügt wurden, zu adressieren:

```
<!doctype html>
 <html>
 <head>
   <style>
     h1 { color: red; } ❶
   </style>
 </head>
 <body>
   <template>
     <style>
       h1 {
         color: green; text-decoration: underline; ❷
       }

       ::content > h1 {
         color: orange; font-style: oblique; ❸
       }
     </style>
     <content></content>
     <h1>Drinnen</h1>
    </template>
```

```
    <h1>Draußen</h1>
    <div>
      <h1>Dazwischen</h1> ❹
    </div>
    <script>
       var root = document.querySelector('div').createShadowRoot()
       var tplContent = document.querySelector('template').content

       root.appendChild(document.importNode(tplContent, true))
    </script>
  </body>
  </html>
```

❶ Im Anwendungsdokument werden `<h1>`-Elemente rot.

❷ Im Shadow DOM werden `<h1>`-Elemente grün und unterstrichen.

❸ Eingefügte Elemente werden orange und kursiv.

❹ Das Element hat jetzt einen expliziten Style erhalten.

Und das Ergebnis sieht erwartungsgemäß aus:

Abb. 2–5 *Jedes Element hat eine klar zugewiesene Style-Regel.*

2.5 Eigene Tags für Komponenten

Wir haben am Anfang des Kapitels das Beispiel einer Tabs-Komponente angesprochen, bei dem wir HTML-Elemente benutzt haben, die uns komplexe Elemente so einsetzen ließen, als wären sie normale HTML-Elemente:

```
<tab-box>
   <tab-pane title="Erster Tab">
     <h1>Der erste Tab</h1>
   </tab-pane>
   <tab-pane title="Zweiter Tab">
     <h1>Der zweite Tab</h1>
   </tab-pane>
 </tab-box>
```

Um solche neuen Elemente definieren zu können, wurde der `Custom Elements`-Standard spezifiziert.

2.5.1 Bisherige Ansätze

Bislang sind eigene UI-Komponenten oftmals eine Mischung aus vordefiniertem HTML-Markup, JavaScript und CSS. Es gibt aber auch Komponenten, die lediglich aus CSS bestehen und üblicherweise über bestimmte CSS-Klassen angesprochen werden:

```
<button type="button" class="btn btn-primary">Primary</button>
```

In diesem Beispiel, entnommen aus der Bootstrap-Bibliothek von Twitter, wird ein Button als »primary« gekennzeichnet. Je nach verwendetem Farbschema kann ein solcher Button unterschiedlich hervorgehoben sein, er wird aber immer gegenüber anderen Buttons betont.

Bei komplexeren Komponenten ist jedoch fast immer JavaScript im Spiel, wie hier im Beispiel von jQuery UI Tabs:

```
<!doctype html>
 <html>
 <body>
   <div id="tabs">
     <ul> ❶
       <li><a href="#tabs-1">Nunc tincidunt</a></li>
       <li><a href="#tabs-2">Proin dolor</a></li>
       <li><a href="#tabs-3">Aenean lacinia</a></li>
     </ul>
     <div id="tabs-1">❷
       <p>Tab 1</p>
     </div>
     <div id="tabs-2">
       <p>Tab 2</p>
     </div>
     <div id="tabs-3">
       <p>Tab 3</p>
       <p>Tab 3, fortgesetzt</p>
     </div>
   </div>
   <script>
     $("#tabs").tabs(); ❸
   </script>
 </body>
 </html>
```

❶ Zunächst wird die Liste mit Tabs angelegt, die hinterher als Navigation dient.

❷ Es folgen die eigentlichen Tabs mit dem jeweiligen Inhalt.

❸ Und schließlich folgt JavaScript, um das Markup entsprechend mit Event-Handlern auszustatten und umzuformatieren.

Das Problem ist hierbei das inflexible und relativ sperrige Markup und seine schwache semantische Bedeutung. Der obige Code macht nicht auf den ersten

Blick klar, wie die Darstellung später wirkt oder aus welchem Grund sich gerade dieses Markup später in ein Tab-Panel verwandelt. In der Praxis ist das Problem deutlich größer, da dort weitere Komponenten innerhalb des Tab-Panels verschachtelt werden können und das JavaScript üblicherweise nicht direkt im Zusammenhang mit dem Markup definiert wird.

2.5.2 Custom Elements

Der Kern des Custom-Elements-Standards ist die neue Methode `document.registerElement`, mit der es möglich ist, dem HTML-Parser des Browsers neue Elemente hinzuzufügen.

Dazu benötigen wir ein Prototyp-Objekt und einen Namen. Gemäß des Standards müssen Custom Elements einen Bindestrich im Namen tragen, damit der Browser solche Elemente leichter identifizieren kann.

Die Registrierung eines eigenen Elements ist einfacher als vielleicht vermutet:

1. Ein Prototyp-Objekt erzeugen, das von einem existierenden HTML-Element-Prototypen erbt
2. Nötigenfalls Methoden für die Lifecycle-Callbacks definieren
3. `document.registerElement` mit dem Namen des neuen Elements und dem Prototypen-Objekt aufrufen

Hier ein Beispiel der Registrierung des `name-tag`-Elements:

```
<!doctype html>
 <html>
 <body>
   <template>
     <style>
       :host {
         display: inline-block;
         background: rgb(255, 64, 64);
         height: 6em;
         border-radius: 1em;
       }
       h1 {
         background: white;
         color: black;
         border-bottom: 1px solid black;
         font-weight: bold;
         margin: 1em 0;
         height: 1em;
       }
     </style>
     <h1>Hi, I'm <span class="name"></span></h1>
   </template>
   <name-tag></name-tag>
   <script>
```

```
    "use strict"
    var NameTagProto = Object.create(HTMLDivElement.prototype); ❶
    NameTagProto.createdCallback = function() { ❷
      this._root = this.createShadowRoot()
      var tplContent = document.querySelector('template').content,
          node       = document.importNode(tplContent, true)
      this._root.appendChild(node)
    }
    NameTagProto.attachedCallback = function() { ❸
      console.log('name-tag was inserted')
      this._root.querySelector('.name').textContent = window.prompt() ❹
    }
    window.NameTag = document.registerElement('name-tag', { prototype:
  NameTagProto }) ❺
   </script>
 </body>
 </html>
```

❶ Hier erzeugen wir ein Prototyp-Objekt für unser Element, abgeleitet von `<div>`.

❷ Diese Funktion wird aufgerufen, sobald der Browser die Registrierung unseres Elements abgeschlossen hat.

❸ Diese Funktion wird aufgerufen, wenn unser Element tatsächlich im HTML-Dokument genutzt wird.

❹ Sobald unser Element eingefügt wurde, erfragen wir den Namen per `window.prompt`.

❺ Hier findet die eigentliche Registrierung unseres Elements statt. Die globale Variable erlaubt es, später Elemente dieses Typs auch aus JavaScript heraus, z. B. mit `document.createElement`, zu erzeugen.

2.5.3 Das <content>-Element

In HTML ist es üblich, dass ein Element seinen Inhalt aus dem Anwendungsdokument (deklarativ) erhält. Bislang haben wir den Inhalt unserer Elemente aber stets aus JavaScript heraus generiert oder direkt im Template festgelegt.

Letztendlich wollen wir jedoch unsere Elemente folgendermaßen nutzen können:

```
<!doctype html>
 <html>
 <body>
   <name-tag>Martin</name-tag>
 </body>
 </html>
```

Dazu können wir uns des `<content>`-Elements bedienen, das einen sogenannten *Insertion Point* bietet. An einer solchen Stelle werden Elemente aus dem Anwen-

dungsdokument in das Shadow DOM eingebettet und stehen dort für Skripte zur Verfügung.

Die Verwendung ist vergleichsweise einfach:

```
<!doctype html>
 <html>
 <body>
   <template>
     <style>
       :host {
         display: inline-block;
         background: rgb(255, 64, 64);
         height: 6em;
         border-radius: 1em;
       }
       h1 {
         background: white;
         color: black;
         border-bottom: 1px solid black;
         font-weight: bold;
         margin: 1em 0;
         height: 1em;
       }
     </style>
     <h1>Hi, I'm <content></content></h1> ❶
   </template>
   <name-tag>Martin</name-tag>❷
   <script>
     var NameTagProto = Object.create(HTMLDivElement.prototype);
     NameTagProto.createdCallback = function() {
       this._root = this.createShadowRoot()
       var tplContent = document.querySelector('template').content,
           node       = document.importNode(tplContent, true)
       this._root.appendChild(node)
     }
     NameTagProto.attachedCallback = function() { ❸
       console.log('name-tag was inserted')
     }
     NameTag = document.registerElement('name-tag', { prototype: NameTagProto })
   </script>
 </body>
 </html>
```

❶ Das `<content>`-Element kann im Template verwendet werden.

❷ Inhalt, der im Anwendungsdokument definiert wurde, wird an das `<content>`-Element weitergegeben und so in das Shadow DOM importiert.

❸ Der `attachedCallback` ist in diesem Beispiel unnötig geworden.

2.5.4 Auf eingebettete Inhalte zugreifen mit getDistributedNodes

Inhalte, die in das Shadow DOM importiert worden sind, lassen sich auch von Skripten innerhalb des Shadow DOM auslesen. Dazu muss die Methode `getDistributedNodes` des entsprechenden `<content>`-Elements aufgerufen werden; sie liefert dann die enthaltenen Nodes (Elemente, Text und so weiter) zurück.

Um beispielsweise auf den Namen aus unserem letzten Beispiel zugreifen zu können, ist dieser Code nötig:

```
<!doctype html>
 <html>
 <body>
   <template>
     <style>
       :host {
         display: inline-block;
         background: rgb(255, 64, 64);
         height: 6em;
         border-radius: 1em;
       }
       h1 {
         background: white;
         color: black;
         border-bottom: 1px solid black;
         font-weight: bold;
         margin: 1em 0;
         height: 1em;
       }
     </style>
     <h1>Hi, I'm <content></content></h1>
   </template>
   <name-tag>Martin<span>XYZ</span></name-tag> ❶
   <script>
     var NameTagProto = Object.create(HTMLDivElement.prototype);
     NameTagProto.createdCallback = function() {
       this._root = this.createShadowRoot()
       var tplContent = document.querySelector('template').content,
           node       = document.importNode(tplContent, true)
       this._root.appendChild(node)
     }
     NameTagProto.attachedCallback = function() {
       console.log('name-tag was inserted')
       var nodes = this._root.querySelector('content').getDistributedNodes(), ❷
           name  = nodes[0].textContent; ❸
       console.log('The name was ' + name);
     }
     NameTag = document.registerElement('name-tag', { prototype: NameTagProto })
   </script>
 </body>
 </html>
```

❶ In diesem Beispiel wird nur der Text des Namens ins Shadow DOM eingefügt.

❷ Mittels `getDistributedNodes` erhalten wir eine Liste der Nodes im `<content>`-Element.

❸ Wir erwarten nur ein einzelnes Element innerhalb des `<content>`-Elements. Also greifen wir direkt auf das erste Element zu und lesen den `textContent` aus.

Hinweis

Im letzten Beispiel würde ein `<name-tag><span>Martin</span></name-tag>` denselben Effekt haben, da wir nach wie vor auf den Text des ersten Elements innerhalb des `<content>` zugreifen können.

Achtung

Die `getDistributedNodes`-Methode liefert alle `Nodes` (das sind Elemente, aber auch Text-Nodes und Kommentar-Nodes). Das folgende Beispiel ist also **nicht** identisch zu den vorherigen Beispielen:

```
<name-tag>
   <span>Martin</span>
 </name-tag>
```

Das Verhalten von `getDistributedNodes` ist auf den ersten Blick nicht immer ganz intuitiv. Der HTML-Parser erzeugt für den Zeilenumbruch am Ende der ersten Zeile und um die Leerzeichen vor dem öffnenden `<span>` eine **Text-Node**, die jetzt als erster Eintrag von `getDistributedNodes` zurückgegeben wird. Demnach enthält `name` dann nur noch einen Zeilenumbruch und zwei Leerzeichen.

Das `<content>`-Element ist sogar in der Lage, eingefügte Inhalte in eine feste Reihenfolge zu bringen, indem man das `select`-Attribut nutzt. Nehmen wir als Beispiel ein `<latest-news>`-Element, das so funktionieren soll, dass Überschriften mit der Klasse `breaking` zuerst angezeigt werden.

Eine solche Anwendung sieht aus wie in Listing 2–13 beschrieben.

```
<!doctype html>
 <html>
 <body>
   <h1>Newsfeed</h1>
   <latest-news>
     <h2>Test 1, non-breaking</h2>
     <h2 class="breaking">Test 2, breaking</h2>
     <h2>Test 1, non-breaking</h2>
     <h2 class="breaking">Test 4, breaking</h2>
   </latest-news>
```

```
</body>
</html>
```

Um jetzt sicherzustellen, dass selbst in diesem Fall die `breaking`-Überschriften zuerst in unser Element importiert werden und dann erst alle weiteren, können wir das `select`-Attribut mit dem passenden CSS-Selektor nutzen:

```
<!doctype html>
 <html>
 <body>
   <template>
     <style>
       h2 {
         background: white;
         color: black;
       }
       ::content .breaking { ❶
         font-size: 1.2em;
         font-weight: bold;
         color: red;
       }
     </style>
     <content select=".breaking"></content> ❷
     <content></content> ❸
   </template>
   <h1>Newsfeed</h1>
   <latest-news>
     <h2>Test 1, non-breaking</h2>
     <div class="breaking">Test 2, breaking</div> ❹
     <h2>Test 1, non-breaking</h2>
     <h2 class="breaking">Test 4, breaking</h2>
   </latest-news>
   <script>
     var NameTagProto = Object.create(HTMLDivElement.prototype);
     NameTagProto.createdCallback = function() {
       this._root = this.createShadowRoot()
       var tplContent = document.querySelector('template').content,
           node       = document.importNode(tplContent, true)
       this._root.appendChild(node)
     }
     NameTag = document.registerElement('latest-news', { prototype:
  NameTagProto })
   </script>
 </body>
 </html>
```

❶ Die importierten `<h2>`-Elemente lassen sich über `::content` ansprechen und stylen.

❷ Dieses `<content>`-Element nimmt alle Elemente mit der Klasse `breaking` auf und importiert sie in das Shadow DOM.

❸ Dieses <content>-Element importiert dann alle verbliebenen Elemente.

❹ Da der Selektor nicht besonders wählerisch ist, können wir auch andere Elemente importieren.

2.5.5 Auf Elemente innerhalb von <content> zugreifen

In manchen Fällen ist es nützlich, auf Elemente zuzugreifen, die mittels <content> in das Shadow DOM eingefügt wurden. Ein Beispiel wäre eine Tab-Box, in der wir für jedes eingefügte Tab-Element einen Reiter erstellen wollen.

Einen solcher Zugriff lässt sich mit der Methode `getDistributedNodes()` des jeweiligen <content>-Elements umsetzen. Der Rückgabewert der Methode ist eine `NodeList`, in der alle Elemente innerhalb des zugehörigen <content>-Elements aufgelistet werden, etwa so:

```
<!doctype html>
 <html>
 <body>
   <x-tabs>
     <div title="Tab 1">Erster Tab</div>
     <div title="Tab 2">Zweiter Tab</div>
     <div title="Tab 3">Dritter Tab</div>
   </x-tabs>
   <template>
     <style>
       ul {
         padding: 0;
       }
       ul li {
         display: inline-block;
         border: 1px solid #666;
         background: #ccc;
         margin-right: 2px;
         padding: 5px
       }
     </style>
     <ul></ul>
     <content></content>
   </template>
   <script>
     var TabsProto = Object.create(HTMLDivElement.prototype);
     TabsProto.createdCallback = function() {
       this._root = this.createShadowRoot()
       var tplContent = document.querySelector('template').content,
           node       = document.importNode(tplContent, true)
       this._root.appendChild(node)
     }
     TabsProto.attachedCallback = function() { ❶
       var tabsNodes = this._root.querySelector('content').
```

```
    getDistributedNodes() ❷
          var navList = this._root.querySelector('ul')
          Array.from(tabsNodes).filter(function(node) { ❸
            return node.title
          }).forEach(function(tab) {
            var li = document.createElement('li')
            li.textContent = tab.title
            navList.appendChild(li)
          })
        }
        Tabs = document.registerElement('x-tabs', { prototype: TabsProto })
      </script>
    </body>
    </html>
```

❶ Wir verwenden den `attachedCallback`, um Code auszuführen, sobald unser Element eingesetzt wird, d. h., sobald wir es in der Anwendung eingebunden und mit Inhalt versehen haben.

❷ Mittels der `getDistributedNodes`-Methode des `<content>`-Elements erhalten wir alle Kindelemente.

❸ Da wir auch Elemente ohne `title`-Attribute in diesem `<content>`-Element haben (Zeilenumbrüche erzeugen `#text`-Knoten), filtern wir unerwünschte Elemente heraus.

Das Ergebnis sieht dann etwa so aus:

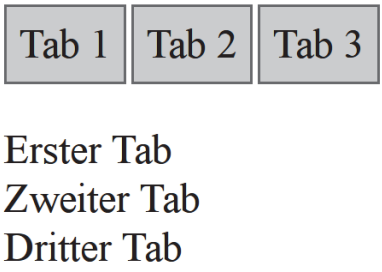

Abb. 2–6 *Tabs mit Reitern*

2.5.6 Eigene Attribute und Methoden

Wenn wir Elemente definieren, ist es oft sinnvoll oder notwendig, auch eigene Attribute oder Methoden einzuführen. Nehmen wir wieder ein Beispiel zur Hand: ein Wetter-Element, das uns das Wetter für einen bestimmten Ort anzeigt. Für dieses Element ist es sicherlich sinnvoll, das Attribut `city` und eine Methode `update()` zu definieren, um den Ort für das Wetter angeben und die Wetterdaten aktualisieren zu können.

Attribute lassen sich mit der ES5-Syntax für Objekteigenschaften per `Object.defineProperty` erzeugen. Methoden können einfach direkt auf dem Prototyp-Objekt definiert werden. Darüber hinaus können wir mithilfe einer Wrapper-Funktion private Hilfsmethoden und Variablen definieren.

Die Verwendung von eigenen Attributen lässt sich an einem Beispiel verdeutlichen: Nehmen wir an, wir möchten ein Element für die Darstellung des aktuellen Wetters in einer beliebigen Stadt erstellen. Dieses Element soll folgendermaßen verwendet werden können:

```
<weather-box id="weather" city="Zuerich"></weather-box> ❶
 <script>
 // ...
 window.setInterval(function() {
   document.getElementById('weather').update() ❷
 }, 60000)
 </script>
```

❶ Das Element zeigt das aktuelle Wetter für Zürich an.

❷ Alle 60 Sekunden wird mittels `update()` das Wetter aktualisiert.

Unsere `weather-box`-Komponente wird etwa so aussehen:

```
<template>
   <p></p>
 </template>
 <script>
   var WeatherBox = null;
   (function() { ❶
     var elemPrototype   = Object.create(HTMLDivElement.prototype)
     // Private Attribute
     var _city  = '' ❷
     // Öffentliche Attribute
     Object.defineProperty(elemPrototype, 'city', { ❸
       get: function() { return location },
       set: function(newCity) {
         _city = newCity
         this.update()
       }
     })
     // Helfer
     function loadWeather(location, units, callback) { ❹
       var xhr = new XMLHttpRequest()
       xhr.open('get', 'http://api.openweathermap.org/data/2.5/
 weather?q=' + location + '&units=' + units)
       xhr.onload = function() {
         try {
           var weather = JSON.parse(this.responseText)
           callback(weather)
         } catch(e) { console && console.error && console.error(e); }
```

```
      }
      xhr.send()
    }
    // Lifecycle callbacks
    elemPrototype.createdCallback = function() {
      this.root = this.createShadowRoot()
      var tplContent = document.querySelector('template').content,
          node       = document.importNode(tplContent, true)
      this.root.appendChild(node)
    }
    elemPrototype.attachedCallback = function() {
      _city = this.attributes.city && this.attributes.city.value ❺
      this.update()
    }
    // Öffentliche methods
    elemPrototype.update = function() { ❻
      var self = this
      if(_city) {
        loadWeather(location, 'metric', function(weather) {
          if(!weather || !weather.weather || weather.weather.length < 1) return
          var weatherDesc = weather.weather[0]
          self.root.querySelector('p').textContent = weatherDesc.
description + ' bei ' + weather.main.temp.toFixed(2) + '°C'
        })
      }
    }
    Element = document.registerElement('weather-box', { prototype:
 elemPrototype })
  })()
</script>
```

❶ Wrapper-Funktion, um private Attribute und Hilfsfunktionen zu ermöglichen.

❷ Privates Attribut, in dem wir den Wert von `_city` zwischenspeichern. Dieses Property ist hier nur zur Verdeutlichung definiert, die Komponente würde auch ohne es funktionieren.

❸ Öffentliches Attribut `city`. Damit ist es auch zur Laufzeit möglich, z. B. mittels `document.querySelector("weather-box").city = "Berlin"`, den Ort zu verändern. Gemäß `set`-Funktion wird automatisch `update()` aufgerufen.

❹ Hilfsfunktion, die nicht von außen aufgerufen werden kann

❺ Beim Einfügen in das Anwendungsdokument prüfen wir, ob bereits ein Wert für `city` definiert ist. Falls ja, setzen wir den Wert in `_city` ein.

❻ Öffentliche Methode `update`. Diese Methode sendet einen Aufruf an die OpenWeatherMap-API, um das Wetter an der aktuell in `_city` hinterlegten Stadt abzurufen und auszugeben.

2.5.7 Existierende HTML-Elemente erweitern

Auch wenn es verlockend erscheinen mag, seine eigenen Elemente von Grund auf zu definieren, so ist es fast immer besser, existierende Elemente zu erweitern. Der entscheidende Unterschied ist, dass existierende Elemente von Browsern auch im Fall von JavaScript-Fehlern oder fehlender Web-Components-Unterstützung korrekt funktionieren. Darüber hinaus stellen Browser für bekannte HTML-Elemente eine Reihe von Features für barrierefreie Anwendungen bereit, die in eigenen Elementen nachgebildet werden müssen.

Eine solche Erweiterung erfolgt in drei Schritten:

1. Der Prototyp unseres Custom-Elements erbt vom Prototypen des zu erweiternden Elements (z. B. dem `HTMLButtonElement.prototype`).
2. Bei der Registrierung wird mittels `extends` angegeben, welches Element erweitert wird.
3. Bei der Nutzung der Komponente wird das Basiselement (z. B. `<button>`) eingesetzt und das `is`-Attribut gesetzt.

Als Beispiel schauen wir uns an, wie wir einen »Image Button«, also einen Button mit einem Bild, als Erweiterung eines `<button>`-Elements realisieren können und wie dieser dann zu verwenden ist.

```
<!doctype html>
 <html>
 <body>
   <template>
     <style>
       img {
         float: right;
         height: 2em;
       }
     </style>
     <button>
       <img>
       <content></content>
     </button>
   </template>
   <button is="image-button" src="happy.png">Hallo!</button> ❶
   <script>
     var ImgButtonProto = Object.create(HTMLButtonElement.prototype); ❷
     ImgButtonProto.createdCallback = function() {
       this._root = this.createShadowRoot()
       var tplContent = document.querySelector('template').content,
           node       = document.importNode(tplContent, true)
       this._root.appendChild(node)
     }
     ImgButton = document.registerElement('img-button', { prototype:
   ImgButtonProto, extends: 'button' }) ❸
```

```
  </script>
</body>
</html>
```

❶ Die Verwendung unserer Komponente ist, bis auf das `src`-Attribut, identisch mit einem gewöhnlichen `<button>`-Element.

❷ Unser Prototyp erbt direkt vom `HTMLButtonElement.prototype`.

❸ Beim Registrieren geben wir mittels `extends` an, dass dieses Element das `<button>`-Element erweitert.

2.6 Komponenten wiederverwenden

Bei allen unseren bisherigen Beispielen haben wir ein Problem noch nicht gelöst: die Wiederverwendung und das Einbinden in eine Anwendung. Bislang haben wir einfach den Code unserer Elemente mit dem Anwendungscode vermischt. Um dies zu ändern, können wir den vierten und letzten Web-Components-Standard benutzen: HTML-Imports.

Der eigentliche Vorgang des Importierens einer Komponente ist simpel:

```
<link rel="import" href="meine-komponente.html">
```

Damit ist es uns möglich, die eigentliche Komponente in der Datei `meine-komponente.html` zu definieren und durch eine einzige Zeile in jede beliebige Anwendung einzubinden.

2.6.1 Anwendungsdokument und Komponentendokument

Bei der Entwicklung von Komponenten gibt es eine Besonderheit, auf die wir achten müssen. In normalen Webanwendungen werden DOM-Elemente einfach über das `document` angesprochen, zum Beispiel mit `document.querySelector`.

Bei Anwendungen, die Komponenten importieren, haben wir aber mehrere Dokumente:

1. das Anwendungsdokument, also das DOM der eigentlichen Anwendung (z. B. in `index.html`)
2. das Komponentendokument, also das DOM der Komponente (z. B. in `meine-komponente.html`)

Wir benötigen also einen Weg, diese Dokumente getrennt anzusprechen. Betrachten wir die folgende Komponentendatei:

```
<html>
   <template>
     <h1>Hallo Komponente</h1>
   </template>
```

```
    <script>
      var Proto = Object.create(HTMLDivElement.prototype);
      Proto.createdCallback = function() {
        alert(document.querySelector('h1').textContent)
      }
      MeineKomponente = document.registerElement('meine-komponente', { prototype:
  Proto })
    </script>
  </html>
```

Was wird die Komponente ausgeben, wenn sie in das folgende Dokument importiert wird:

```
<html>
   <head>
     <link rel="import" href="meine-komponente.html">
   </head>
   <body>
     <h1>Hallo Anwendung!</h1>
     <meine-komponente></meine-komponente>
   </body>
 </html>
```

Die Anwendung wird »Hallo Anwendung!« ausgeben, statt wie gewünscht »Hallo Komponente«.

Achtung

`document` bezieht sich **immer** (also auch im Komponentendokument) auf das Anwendungsdokument!

Wenn wir also das Komponentendokument ansprechen wollen, müssen wir einen Umweg machen, für den uns `document.currentScript.ownerDocument` zur Verfügung steht. Mit `document.currentScript` erhalten wir das `<script>`-Element, das gerade ausgeführt wird, und sein `ownerDocument` gibt das Dokument an, in dem es sich befindet. Im Fall eines `<script>`-Elements innerhalb des Komponentendokuments liefert dieser Code also das Komponentendokument.

```
var componentDoc = document.currentScript.ownerDocument
```

Hinweis

Der Polyfill für HTML-Imports definiert nicht `document.currentScript`, sondern `document._currentScript`. Hier ist also ein Fallback nötig.

2.7 Ein Grundgerüst für eigene Komponenten

Mit den Erkenntnissen aus den letzten Abschnitten können wir uns ein einfaches Grundgerüst zusammenstellen, das für alle unsere Komponenten als Grundlage zu verwenden ist.

```
<template>
   <!-- Komponenten-Markup hier einfügen -->
 </template>
 <script>
   var Element = null ❶
   (function(currentScript) {
     var prototype = Object.create(HTMLDivElement.prototype) ❷
     // Private Attribute / Methoden
     // Öffentliche Attribute / Methoden
     // Lifecycle Callbacks
     prototype.createdCallback = function() {
       this._root = this.createShadowRoot()
       var tplContent = currentScript.ownerDocument.querySelector('template').
 content, ❸
           node = document.importNode(tplContent, true) ❹
       this._root.appendChild(node)
     }
     // Registrierung
     Element = document.registerElement('my-element', {prototype: prototype})
   })(document.currentScript || document._currentScript) ❺
 </script>
```

❶ Diese Variable wird später unser registriertes Element beinhalten, damit es auch imperativ verwendet werden kann. Der Name sollte an die eigene Komponente angepasst werden.

❷ Der Prototyp unserer Komponente wird erzeugt. Es sollte ein möglichst ähnliches natives HTML-Element als Prototyp verwendet werden.

❸ Hier nutzen wir den Umweg über `document.currentScript.ownerDocument` (oder den Fallback des Polyfills), um auf das Komponentendokument zuzugreifen.

❹ Zum Importieren können wir weiterhin `document` verwenden.

❺ Hier rufen wir den Wrapper auf und übergeben entweder `document.currentScript` oder den Polyfill-Fallback.