1 Grundprinzip CSR

Wir haben bei den Template Engines über die Vorteile von Serverside Rendering gesprochen. Nun wollen wir über die Vorteile von CSR sprechen – bzw. korrekterweise über Single Page Applications (SPA), welche Clientside Rendering voraussetzen. Die Idee hinter Clientside Rendering ist, dass die HTML-Struktur nicht auf dem Server erzeugt wird, sondern per JavaScript auf dem Client. Sehen wir uns hierzu folgende HTML-Seite im File `staticPage.html` an:

```
<!DOCTYPE html>
<html lang="de">
<head>
   <meta charset="utf-8">
   <title>Beispielseite</title>
   <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
  <h1>Wichtige Funktionen für CSR</h1>
  <ul>
    <li>getElementById</li>
    <li>getElementsByTagName</li>
    <li>querySelector</li>
    <li>createElement</li>
    <li>append</li>
  </ul>
</body>
</html>
```

Die komplette Seite wurde durch das HTML-File vorgegeben. Der Browser zeigt exakt die HTML-Struktur an, welche auf dem Server liegt. Durch die hierarchische Struktur des HTML-Codes erzeugt der Browser jedoch intern ein Document Object Model (DOM), welches er für die Anzeigt nutzt. Dieses DOM können wir nun auch per JavaScript verarbeiten, oder gar komplett selbst erzeugen. Wir kopieren nun die dynamische Seite in ein neues File `dynamicPage.html` und reduzieren unseren HTML-Body nun auf das folgende Konstrukt:

```
…
</head>
<body onload="buildMyPage()">
</body>
</html>
```

Weiterhin erzeugen wir ein JavaScript File `dynamicPage.js` und referenzieren es im HTML-Header:

```
…
   <title>Beispielseite</title>
   <script src="dynamicPage.js"></script>
   <meta name="viewport" content="width=device-width, initial-scale=1">
…
```

Dieses Skript soll nun dynamisch das gleiche HTML-Konstrukt erzeugen, wie unser `staticPage.html`. Hier fügen wir folgenden Code ein, wobei wir die einzelnen Elemente im Anschluss besprechen werden:

```
function buildMyPage() {
  const body = document.getElementsByTagName("body")[0];
  const head = document.createElement("h1");
  head.innerHTML = "Wichtige Funktionen für CSR";
  body.append(head);
  const list = document.createElement("ul");
  const listContent = ["getElementById",
                "getElementsByTagName",
                "querySelector",
                "createElement",
                "append"];
  listContent.forEach(entryText => {
    let entry = document.createElement("li");
    entry.innerHTML = entryText;
    list.append(entry);
  });
  body.append(list);
}
```

Rufen wir nun `dynamicPage.html` auf, sehen wir wieder das gleiche Ergebnis. Mehr noch – wenn wir im Browser das development Fenster öffnen (bspw. mit der Taste `F12`) und dort den „Element“ Tab öffnen, sehen wir das gleiche Ergebnis, als würden wir dies in der `staticPage.html` machen:

Wir haben nun die Seite dynamisch mittels DOM-Manipulationen erzeugt.

2 Funktionen für die DOM-Manipulation

Das Wichtigste ist erstmal, dass wir existierende Elemente des DOM identifizieren und referenzieren können. Hierzu bietet uns das global existierende `document` Objekt mehrere Funktionen an:

Funktion im document Obj.	**Erläuterung**
`getElementById(`*`id`*`)`	Sucht den DOM nach einem Element ab, welches im id-Attribut den Wert des Strings *`id`* hat und gibt eine Referenz zurück.
`querySelector(`*`sel`*`)`	Sucht den DOM nach einem Element nach dem in *`sel`* spezifizierten Element ab und gibt eine Referenz zurück. Wobei *`sel`* ein CSS-Queryselector[1] darstellen muss. Würden mehrere der Forderung entsprechen, wird der erste zurückgegeben.
`querySelectorAll(`*`sel`*`)`	Wie querySelector(), jedoch wird eine `NodeList` zurückgegeben. Dadurch können bei mehreren „Treffern“ alle qualifizierenden Elemente zurückgegeben werden.
`getElementsByTagName(`*`name`*`)`	Sucht den DOM nach Elementen mit dem in *`name`* eingetragenen Namen ab und gibt eine `NodeList` mit den entsprechenden Referenzen zurück.

[1] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors

Beim Erzeugen von neuen Elementen müssen wir prinzipiell drei Dinge tun:

- Erzeugen des Elements
- Festlegen der Eigenschaften
- Hinzufügen des neuen Elements zum Elternelement

Hierzu bietet uns JavaScript folgende Funktionen an:

Funktion/Eigenschaft.	**Erläuterung**
`createElement(`*`tagName`*`)`	Die Methode befindet sich im `document` Objekt und erzeugt ein neues DOM-Objekt des Typs *`tagName`*.
`elm.innerHTML`	`innerHTML` ist die Stringrepräsentation des Inhaltes zwischen dem öffnenden und schließenden Tag des `elm`.DOM Objektes. Alle weiteren Eigenschaften, wie bspw. `elm.id` finden sich online, bspw. in der mozilla Dokumentation[2]. Achtung – bei dem „`class`“ Attribut muss über `elm.classList.add("`*`myClass`*`")`, verwendet werden.
`elm.addEventListener(`*`type, listener`*`)`	Fügt an ein Element Objekt einen Eventlistener zu vom Typ *`type`* zu[3]. *`listener`* muss eine Funktion mit einem Argument (dem `Event`[4]) sein.
`elm.append(`*`child`*`)`	Mit append fügt man *`child`* als Kindelement zu `elm` hinzu.
`elm.remove()`	Entfernt das Element aus der ChildList des Elternelements.

3 Aufrufreihenfolge

Eine HMTL-Seite wird ohne weiteres Zutun erstmal sequenziell abgearbeitet und interpretiert. Dies bedeutet aber auch, dass eventueller JavaScript Code dann ausgeführt wird, wenn er geladen wird. Gehen wir von folgendem JavaScript File testScript.js aus:

```
let myDiv = document.createElement("div");
myDiv.innerHTML = "hello, world";
document.getElementById("myBody").append(myDiv);
```

Es erzeugt also direkt beim Laden des Skriptes ein DIV Element und fügt es an das Element mit der ID myBody hinzu. Dieses wird in der HTML Seite wie folgt geladen:

```
<head>
  <title>Beispielseite</title>
  <script src="testScript.js"></script>
</head>
<body id="myBody">
</body>
</html>
```

Öffnen wir die HTML-Datei mit einem Browser, erhalten wir eine Fehlermeldung, dass „append“ nicht auf null ausgeführt werden kann – sprich der Browser hat das Body-Element nicht gefunden. Dies liegt daran, dass es erst nach dem Laden – und ausführen – des JavaScript Files erzeugt wird. Das Element existiert also noch nicht im DOM, wenn der JavaScript Code läuft! Um diesem Dilemma zu entkommen, findet man im Regelfall eine der drei möglichen Lösungsvarianten. Entweder wir ergänzen im `script` Tag das Wort „`defer`“, was den Browser dazu bewegt, das JavaScript File am Schluss zu laden:

```
…
  <script src="testScript.js" defer></script>
…
```

[2] https://developer.mozilla.org/en-US/docs/Web/API/Element
[3] https://developer.mozilla.org/en-US/docs/Web/Events
[4] https://developer.mozilla.org/en-US/docs/Web/API/Event

Die zweite Möglichkeit ist es, das `script` Tag am Ende einzubetten:

```
<head>
  <title>Beispielseite</title>
</head>
<body id="myBody">
  <script src="testScript.js"></script>
</body>
</html>
```

Die dritte Möglichkeit ist es, das Skript in eine Funktion einzubetten und diese im `body` Tag mit `onload` aufzurufen, so wie es ganz oben mit `buildMyPage()` gezeigt wurde.

4 Aufgabenstellung

Nehmen Sie das File `tickTackToe.html` und analysieren Sie es. Löschen Sie nun den folgenden Teil dieser Datei:

```
<div class="table">
  <div id="0/0" onclick="gridClicked(event)"> </div>
  <div id="1/0" onclick="gridClicked(event)"> </div>
  <div id="2/0" onclick="gridClicked(event)"> </div>

  <div id="0/1" onclick="gridClicked(event)"> </div>
  <div id="1/1" onclick="gridClicked(event)"> </div>
  <div id="2/1" onclick="gridClicked(event)"> </div>

  <div id="0/2" onclick="gridClicked(event)"> </div>
  <div id="1/2" onclick="gridClicked(event)"> </div>
  <div id="2/2" onclick="gridClicked(event)"> </div>
</div>
</br>
<span class="myButton" id="resetGame" onclick="resetGame()">reset</span>
```

und erzeugen Sie ihn dynamisch per JavaScript. Erweitern Sie hierzu im File `tickTackToe.js` die Funktion `buildMyPage`.