Chris A. Mair

JavaScript

Entspannt zur Single-Page-Application

Ein Webmasters Press Lernbuch

Version 1.1.1 vom 3.11.2016

Autorisiertes Curriculum für das Webmasters Europe Ausbildungs- und Zertifizierungsprogramm.

www.webmasters-europe.org

Inhaltsverzeichnis

Vorwort

JavaScript! JavaScript! JavaScript?

Wenn Sie diese Zeilen lesen, haben Sie den Einstieg in die Programmierung mit *JavaScript* hinter sich.

Sie manipulieren HTML wie ein Magier und der Webbrowser macht genau das, was Sie ihm sagen! Vielleicht sind Sie geradezu euphorisch und stellen sich schon vor, wie Sie den nächsten Hit im Internet landen?

Leider gibt es eine Reihe von Problemen, von denen Sie noch gar nicht wissen, dass Sie sie haben (werden) ;-)

Wie vermeidet man komplizierte HTML-Manipulations-Klimmzüge? Was war jetzt eigentlich dieses HTTP? Wie kommen denn die Daten überhaupt vom Server in den Webbrowser (und wieder zurück)? Server? Wieso Server?

Die *JavaScript*-Küche brodelt über vor Techniken und Trends: AJAX, Websockets, JSON, *Node.js*, *AngularJS*…

Dieses Buch möchte Sie an der Hand nehmen und die Stufen vom frisch gekürten *JavaScript*-Programmierer zum Frontend-Entwickler so flach wie möglich halten und dafür sorgen, dass Sie die Lösung der oben genannten Probleme schon kennen, bevor sie auftauchen!

Dieses Buch will auf keinen Fall ein Nachschlagwerk zu den verwendeten Technologien sein, sondern vielmehr ein Rundgang. Es soll das Bild von *JavaScript* abrunden, um Sie als Frontend-Programmierer bereit für die große Welt zu machen.

Viel Spaß dabei!

Chris Mair

3 Die erste Anwendung: eine To-do-Liste

In dieser Lektion lernen Sie

- wie man Webseiten per HTTP-Protokoll aufrufen kann.
- wie man eine einfache Anwendung unter Berücksichtigung des MVC-Entwurfsmusters in *JavaScript* programmiert.

3.1 Der eigene Webserver

Als Leser dieses Buches haben Sie sicher bereits eigene HTML-Dateien erstellt und direkt im Webbrowser geöffnet. Wenn ich eine Datei *hi.html* auf meinem Desktop öffne, sieht das so aus:

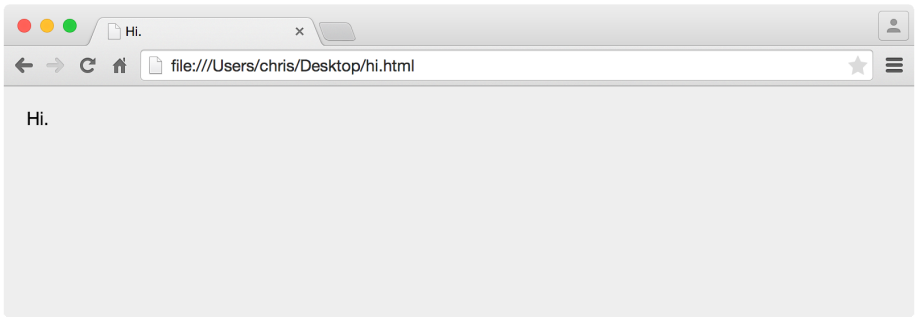

Abb. 3.1 *Adressleiste mit* ***file://****-URL*

Die Zugriffsmethode ist hier `file://`, was einfach nur bedeutet, dass der Webbrowser die Datei direkt öffnet, also auf das lokale Dateisystem zugreift. Dies eignet sich prima für einfache Experimente mit HTML und CSS, wird aber schnell zur Hürde, wenn man etwas aufwendigere Dinge machen möchte.

Die meisten Webentwickler benutzen daher ihren eigenen, lokal installierten Webserver, um Dateien im Webbrowser mittels HTTP zu öffnen. Das klingt zunächst etwas kompliziert, ist aber tatsächlich sehr einfach!

An dieser Stelle benötigen Sie ***Node.js***. Falls Sie die Software noch nicht installiert haben, sollten Sie das nun tun (siehe Lektion 8).

Die Installation von *Node.js* erlaubt Zugriff auf eine umfangreiche Sammlung von *JavaScript*-Paketen und -Programmen, zum Beispiel — Sie erraten es schon — einen kleinen, aber feinen Webserver! Geben Sie bitte in der Eingabeaufforderung (siehe Abschnitt 8.2) folgenden Befehl ein:

```
npm install http-server -g
```

Dieser Befehl installiert mittels NPM das Paket ***http-server***: einen Open Source Webserver, der ganz ohne Konfiguration auskommt.

NPM

NPM stand ursprünglich für Node Package Manager, wird aber mittlerweile auch von anderen Projekten außer *Node.js* verwendet. Die NPM-Webseite[2] hat viele Vorschläge, was die Abkürzung sonst noch bedeuten könnte: Ich glaube, man darf sich das selbst zusammenreimen…

Der `npm`-Befehl, aufgerufen mit dem Argument `install`, kümmert sich selbstständig um den Download und die Installation des gewünschten Paketes von der *NPM*-Webseite, in diesem Fall des Paketes *http-server*.

Das Argument `-g` steht für *global*: Es weist `npm` an, *http-server* betriebssystemweit zu installieren. Somit muss Ihr Benutzerkonto auch entsprechende Rechte aufweisen! Unter *OS X* sollten Sie daher dem `npm`-Befehl noch ein `sudo` voranstellen, unter *GNU/Linux* eine Root-Shell verwenden und unter *Microsoft Windows* den entsprechenden Dialog bestätigen.

Überlegen Sie nun, welches Verzeichnis Sie vom Webserver per HTTP exportieren lassen möchten, und legen Sie darin eine Beispieldatei ab, etwa eine HTML-Datei namens *hi.html*. Starten Sie danach den Webserver in der Eingabeaufforderung:

```
http-server /Users/chris/Desktop/Projekt
```

Hier ist */Users/chris/Desktop/Projekt* der Pfad auf meinem System, den Sie entsprechend anpassen sollten. Benutzen Sie *Microsoft Windows*? Dann denken Sie bitte daran, dass dieses System Backslashes als Pfadtrenner verwendet:

```
http-server c:\Users\chris\Desktop\Projekt
```

Enthält der Pfad Leerzeichen? Dann muss er mit Anführungszeichen geschützt werden:

```
http-server "c:\Users\chris\Desktop\Mein Projekt"
```

Sie können nun per HTTP auf Ihre Dateien zugreifen! Geben Sie in der Adressleiste Ihres Webbrowsers die URL `http://localhost:8080/hi.html` ein. Dabei ist ***localhost*** immer der eigene Rechner und `8080` der sogenannte ***Port***, auf dem der Webserver läuft. Während Webserver in Produktionsumgebungen den Standardport 80 verwenden, ist der Port 8080 für Testwebserver vorgesehen. Das Ganze sollte nun so aussehen:

2. http://www.npmjs.com

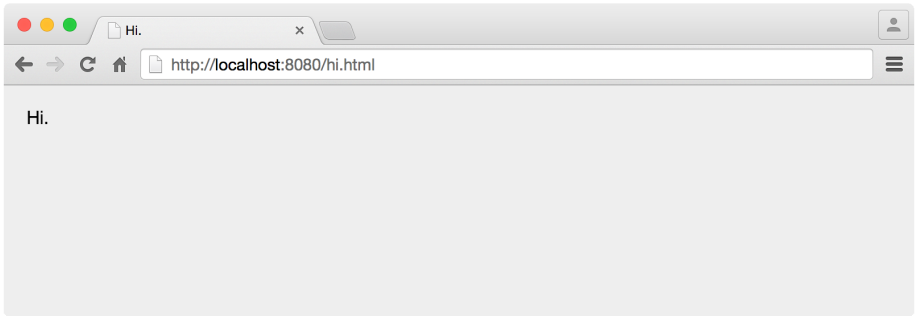

Abb. 3.2 *Adressleiste mit* **http://***-URL*

Glückwunsch! Auf Ihrem Rechner läuft nun Ihr eigener Webserver.

3.2 Die To-do-Liste

Eine ***To-do-Liste*** ist sicherlich eine überschaubare Anwendung, trotzdem ist es notwendig, zunächst einige Begriffe zu definieren und die Anforderungen festzuhalten.

- Ein To-do ist ein kurzer Text mit einem Status (»erledigt« oder »nicht erledigt«).
- Alle To-dos werden in einer Liste gesammelt, zu der jederzeit neue To-dos hinzugefügt werden können. Der Anfangsstatus eines To-do ist »nicht erledigt«.
- Der Status jedes einzelnen To-dos kann jederzeit beliebig verändert werden, d. h. ein To-do kann als »erledigt« markiert werden, bereits erledigte To-dos aber auch wieder als »nicht erledigt«.
- Außerdem gibt es die Möglichkeit, die Liste »aufzuräumen«, d. h. alle To-dos mit Status »erledigt« aus der Liste zu entfernen.

Wir brauchen uns vorerst keine Gedanken über die ***Persistenz*** der Daten zu machen. Unerledigte To-dos werden einfach verworfen, wenn der Webbrowser geschlossen wird (wäre es im Arbeitsalltag nur auch so einfach).

Wenn Sie jetzt schon zum HTML-Editor greifen: Stopp, immer langsam mit den jungen Pferden! Da war noch was: MVC!

3.2.1 Das Modell (todolist.js)

Sie sollten sich zuerst um das Datenmodell kümmern.

Es bietet sich an, eine *JavaScript*-Klasse zu verwenden. Die Klasse soll die Eigenschaften einer To-do-Liste enthalten sowie die Methoden, die sich aus den oben genannten Anforderungen ergeben. Zum Glück ist unsere Anwendung recht einfach — entsprechend übersichtlich ist die Klasse:

```
 1 "use strict";
 2
 3 class TodoList {
 4
 5   constructor() {
 6     this.list = [];
 7   }
 8
 9   getList() {
10     return this.list;
11   }
12
```

```
13   add(t) {
14     this.list.push( { text: String(t), done: false } );
15   }
16
17   set(ix, done) {
18     this.list[ix].done = done;
19   }
20
21   clean() {
22     let cleanedList = [];
23     this.list.forEach( el => {
24       if (!el.done) {
25         cleanedList.push(el);
26       }
27     });
28     this.list = cleanedList;
29   }
30
31   dump() {
32     this.list.forEach(
33       (t, i) => console.log(`${i}: ${t.text}: ${t.done}`)
34     );
35     console.log("---"); // Trenner
36   }
37
38 }
```

Codebeispiel 3.1 *ch03/todolist.js*

Jedes To-do ist ein Objekt mit den Eigenschaften `text` (ein String) und `done` (ein Boolean). Ein erledigtes To-do erhält `done = true`. Im Konstruktor wird ein leeres Array namens `list` angelegt, auf das mittels `this.list` zugegriffen wird.

Außerdem sind noch fünf Methoden vorhanden:

- `getList()` gibt die Liste selbst (also das Array `this.list`) zurück.
- `add(t)` fügt der Liste ein neues (noch nicht erledigtes) To-do mit dem Text `t` hinzu.
- `set(ix, done)` setzt die Eigenschaft `done` des To-dos an der Position `ix` in der Liste.
- `clean()` entfernt alle erledigten To-dos aus der Liste. Dazu wird ein neues Array erstellt, in das alle Einträge mit `done == false` kopiert werden. Dieses neue Array wird anschließend der Variablen `this.list` zugewiesen.
- `dump()` loggt die ganze To-do-Liste übersichtlich in die Konsole.

Übung 4: Die Klasse TodoList ausprobieren

Probieren Sie die TodoList-Klasse aus! Übertragen Sie den Code aus Codebeispiel 3.1 ins »Scratchpad« von *Firefox* oder eine vergleichbare Umgebung um *JavaScript*-Code auszuprobieren. Fügen Sie nach der Klassendefinition noch folgende Zeilen zum Testen an:

```
let myList = new TodoList();

myList.add('Kaffee kaufen');
myList.add('Code testen');
myList.dump();

myList.set(0, true);
```

```
myList.dump();

myList.clean();
myList.dump();
```

Wenn Sie den Code ausführen, sollte das Ergebnis so aussehen:

```
0: Kaffee kaufen: false
1: Code testen: false
---
0: Kaffee kaufen: true
1: Code testen: false
---
0: Code testen: false
---
```

So weit, so gut. Das To-do-Listen-Modell ist unter Dach und Fach: Jetzt kommen die Präsentation und die Steuerung ins Spiel!

3.2.2 Die Präsentation (app.html)

Ein Listing sagt mehr als tausend Worte. Das folgende Codebeispiel enthält einen Vorschlag, wie die Präsentation der Anwendung aussehen könnte:

```
<!doctype html>
<html>

  <head>
    <meta charset="utf-8" />
    <title>To-do-Liste (Kapitel 3)</title>
    <style type="text/css">
      body  {
        font-family: sans-serif;
        margin: 20px;
      }
      table {
        border-collapse: collapse;
      }
      td {
        padding: 8px;
      }
    </style>
    <script src="todolist.js"></script>
    <script src="app.js" defer></script>
  </head>

  <body>
    <h2>To-dos</h2>
    <table>
      <tbody id="todos" />
      <tbody>
        <tr>
          <td><input id="newTodo" type="text" /></td>
          <td><button id="add">hinzufügen</button></td>
        </tr>
      </tbody>
    </table>
    <br /><br />
    <button id="dump">in der Konsole anzeigen</button>
```

```
36      <button id="clean">erledigte To-dos löschen</button>
37    </body>
38
39  </html>
```

Codebeispiel 3.2 *ch03/app.html*

Der HTML-Code gestaltet sich simpel: In den Zeilen 24-33 gibt es eine Überschrift und eine Tabelle, die in zwei `tbody`-Bereiche aufgeteilt ist. Der erste Bereich (mit der ID »todos«) wird die To-dos aufnehmen, ist aber zunächst leer. Der zweite Bereich enthält das Eingabefeld für den Text eines neuen To-dos und einen Button zum Bestätigen der Eingabe. Die zwei weiteren Buttons in den Zeilen 35-26 erlauben es, die aktuelle To-do-Liste in der Konsole anzuzeigen, sowie die erledigten To-dos zu löschen.

Das Interessante sind natürlich die beiden Skripte, die in den Zeilen 19-20 geladen werden: das Modell in der bereits getesteten TodoList-Klasse, sowie die Steuerung in der Datei *app.js*.

Wir sehen uns *app.js* gleich genauer an! Vorerst sollten Sie die Anwendung aber einfach einmal ausprobieren. Kopieren Sie dazu alle drei Dateien (*todolist.js*, *app.js* und *app.html* in Ihr Webserver-Verzeichnis und rufen sie die URL `http://localhost:8080/app.html` auf. Denken Sie daran, dass `http-server` laufen muss (siehe Abschnitt 3.1), sonst kann Ihr Webbrowser keine Verbindung zu `localhost:8080` herstellen.

Sie können nun probeweise ein paar To-dos eingeben. Ich muss Kaffee kaufen und Code testen; das sieht dann so aus:

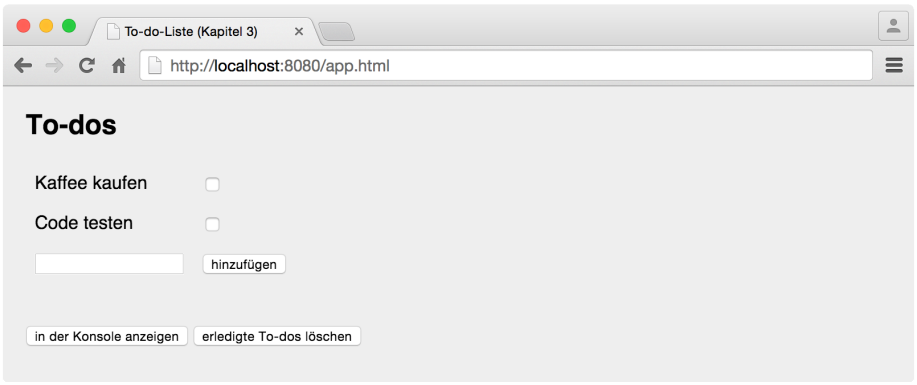

Abb. 3.3 *zwei wichtige To-dos…*

(Wenn der Kaffee alle ist, sollte ich keinen Code testen: Die Prioritäten sind hier schon mal klar.)

Was passiert genau, wenn ein To-do hinzugefügt wird? Die meisten Webbrowser liefern Werkzeuge, um die ***DOM***-Manipulationen sichtbar zu machen — *Firefox* z. B. den »***Inspector***«. Der folgende Screenshot zeigt, wie die beiden To-dos im ersten `tbody`-Bereich hinzugefügt wurden. Der »Inspector« bietet hierbei eine praktische Live-Ansicht.

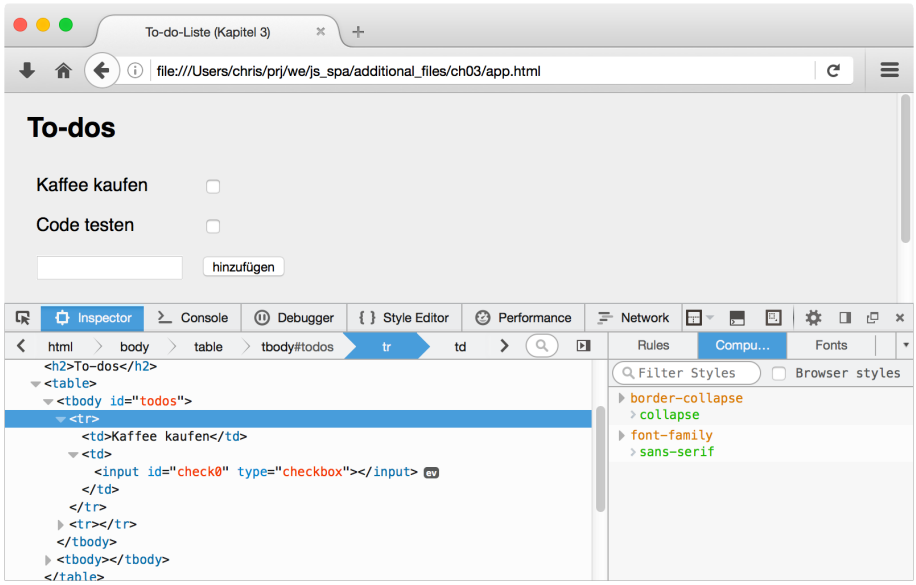

Abb. 3.4 *der Firefox »Inspector«*

3.2.3 Die Steuerung (app.js)

Welche Aufgaben muss nun die Steuerung übernehmen?

- Ein Klick auf den »hinzufügen«-Button soll der Liste ein neues (nicht erledigtes) To-do hinzufügen — der Text dazu kommt aus dem Textfeld. Die entsprechende `add()`-Methode im Modell ist bereits vorhanden. Anschließend muss noch die Darstellung der Liste aufgefrischt werden, damit das neue To-do auch angezeigt wird.
- Ein Klick auf »erledigte To-dos löschen« soll die `clean()`-Methode im Modell aufrufen und wiederum die Darstellung der Liste auffrischen (die hoffentlich kürzer geworden ist ;).
- Ein Klick auf »in der Konsole anzeigen« soll einfach die `dump()`-Methode im Modell aufrufen. Weitere Aktionen sind in diesem Fall nicht nötig.
- Schließlich muss auch noch das Auswählen (oder Nicht-Auswählen) der einzelnen Checkboxen berücksichtigt und die `set()`-Methode im Modell mit den entsprechenden Parametern aufgerufen werden.

Da gibt es tatsächlich einiges zu tun! Das folgende Codebeispiel setzt das alles um:

```
"use strict";

const myList = new TodoList();

const refreshView = () => {

  let content = "";
  myList.getList().forEach( (el, ix) => {
    let checked = "";
    if (el.done) { checked = "checked"; }
    content += `
      <tr>
        <td>${escape(el.text)}</td>
        <td>
          <input type="checkbox" id="check${ix}" ${checked} />
        </td>
      </tr>
    `;
  });
  $("#todos").innerHTML = content;

```

```
22   myList.getList().forEach( (el, ix) => {
23     $("#check" + ix).addEventListener("click",
24       () => { myList.set(ix, $$("#todos input")[ix].checked); }
25     );
26   });
27
28 };
29
30 const $ = document.querySelector.bind(document);
31 const $$ = document.querySelectorAll.bind(document);
32
33 const escape = str => {
34   return str.replace(/\&/g, "&amp;")
35             .replace(/\</g, "&lt;")
36             .replace(/\>/g, "&gt;");
37 }
38
39 $("#add").addEventListener("click", () => {
40   if ($("#newTodo").value.trim() !== "") {
41     myList.add($("#newTodo").value);
42     refreshView();
43   }
44   $("#newTodo").value = "";
45 });
46
47 $("#clean").addEventListener("click", () => {
48   myList.clean();
49   refreshView();
50 });
51
52 $("#dump").addEventListener("click", () => {
53   myList.dump();
54 });
```

Codebeispiel 3.3 *ch03/app.js*

In Zeile 3 wird eine (leere) To-do-List instanziiert: `myList`.

Die Funktion `refreshView()` in den Zeilen 5-28 kümmert sich um die Darstellung der To-do-Liste. Dabei baut eine Schleife über die Listenelemente die Tabellenzeilen Zeile für Zeile auf (8-19) und überschreibt damit den ersten `tbody`-Bereich, der mittels Selektor `id="todos"` gefunden wird.

Machen Sie ein interessantes Experiment: Geben Sie in der Webbrowser-Console ein:

```
myList.add("JavaScript lernen");
refreshView();
```

Die erste Anweisung verändert das Modell, aber erst der Aufruf von `refreshView()` macht die Änderung sichtbar!

Die Funktion `refreshView()` hat noch eine weitere Aufgabe. In den Zeilen 22-26 gibt es eine weitere Schleife über die Listenelemente. Darin wird jeder Checkbox noch ein ***Eventhandler*** hinzugefügt. Ein Klick auf die `ix`-te Checkbox soll also die `set()`-Methode im Modell aufrufen (Zeile 24):

```
myList.set(ix, $$("#todos input")[ix].checked);
```

Das erste Argument (`ix`) ist die Position des To-do in der Liste. Das zweite Argument (`$$("#todos input")[ix].checked`) ist das `checked`-Attribut der entsprechenden Checkbox (ein Boolean).

Vielleicht fragen Sie sich, warum das Ganze noch in einen Funktionsausdruck »eingeschlossen« wird?

```
() => { myList.set(ix, $$("#todos input")[ix].checked); }
```

Ganz einfach: `addEventListener()` erwartet als zweites Argument ja eine Funktion. Allerdings kann nicht einfach `myList.set` übergeben werden, da die `set()`-Methode ja ihrerseits Argumente benötigt, in diesem Fall sogar variable Argumente.

Eine auf diese Art »eingeschlossene« Funktion kann auf lokale Variablen zugreifen, die im Moment ihrer Entstehung gültig waren. Und zwar auch dann, wenn die Funktion erst viel später aufgerufen wird, wie hier, als Eventhandler. Man nennt eine solche Funktion übrigens eine »***Closure***«.

Sie können sich das einfach so vorstellen, dass beim Durchlaufen der `forEach`-Schleife jede Checkbox ihren eigenen Eventhandler bekommt:

```
() => { myList.set(0, $$("#todos input")[0].checked); }
() => { myList.set(1, $$("#todos input")[1].checked); }
() => { myList.set(2, $$("#todos input")[2].checked); }
...
```

Das Gröbste ist geschafft!

Die Zeilen 30-37 enthalten Utility-Funktionen. Die kompakten Definitionen der Selektoren `$` und `$$` kennen Sie bereits aus dem Buch *JavaScript. HTML mühelos manipuliert* aus der *Webmasters Europe*-Reihe (M. EMRICH, C. MARIT (2015-2). Die Funktion `escape()` ersetzt die HTML-relevanten Zeichen (&, < und >) durch die entsprechenden Entities (&, < und >). Ohne den Aufruf von `escape()` in Zeile 13 könnten To-dos nicht korrekt dargestellt werden, wenn sie diese Zeichen enthalten.

Schließlich werden noch die drei Eventhandler für die drei Buttons definiert (Zeile 39-54). Diese greifen auf die passenden Methoden im Modell und auf `refreshView` zurück und sind entsprechend überschaubar.

Übung 5: Anzahl der unerledigten To-dos

Erweitern Sie die Anwendung. Unter der Überschrift (Zeile 24 im Codebeispiel 3.2) soll jederzeit die Anzahl der noch nicht erledigten To-dos angezeigt werden!

Übung 6: Vorsicht Fehler!

Ändern Sie die Zeile 15 in *app.js* (Codebeispiel 3.3). Aus:

```
<input type="checkbox" id="check${ix}" ${checked} />
```

wird

```
<input type="checkbox" id="check${ix}" />
```

Auf den ersten Blick funktioniert die Anwendung noch. Wenn Sie etwas genauer testen, werden Sie aber einen Fehler finden. Beschreiben Sie den Fehler aus der Sicht eines Anwenders!

Übung 7: Konfliktvermeidung

Der Code in *app.js* (Codebeispiel 3.3) definiert einige globale Konstanten, wie `myList`, `refreshView()` u. a. Das ist zum Testen in der Webbrowser-Console bequem, »verschmutzt« allerdings den globalen ***Namensraum*** mit Bezeichnern. Gerade bei Allerweltsbezeichnern wie `escape()` sind Konflikte vorprogrammiert, wenn verschiedene Codeteile in einem Projekt zusammen geführt werden.

Ein bekannter Trick besteht darin, den Code in eine sogenannte ***IIFE*** zu packen. *IIFE* steht für *Immediatly Invoked Function Expression* — ein Funktionsausdruck, der sofort ausgeführt wird:

```
(() => {
    // Code
})();
```

Packen Sie also den gesamten Code in *app.js* in eine *IIFE*. Testen Sie, ob die Anwendung noch korrekt funktioniert.

3.3 Ausblick

Werfen wir einen abschließenden Blick auf den Code der Steuerung in *app.js* (Codebeispiel 3.3)! Der Code funktioniert. Ideal ist er aber nicht. Die Funktion `refreshView()` zum Aktualisieren der Präsentation fällt als besonders umständlich auf.

Die Tabellenzeilen werden »von Hand« aufgebaut. Das funktioniert zwar, ist aber alles andere als robust. Ein Teil der Präsentation ist sozusagen fest in die Steuerung verdrahtet. Was passiert z. B., wenn die Tabelle irgendwann noch andere Spalten enthalten soll? Wie wartbar wird das Ganze, wenn die Anwendung komplizierter wird?

Außerdem ist der Aufwand, die Präsentation mit dem Datenmodell konsistent zu halten, beträchtlich, und es schleichen sich schnell Fehler ein (siehe Übung 6).

Hier gibt es Raum für Verbesserungen, wie wir in der folgenden Lektion sehen werden!

3.4 Wissensfragen

1. Welcher Satz beschreibt *Node.js* am besten?

   Bitte ankreuzen:

   a. *Node.js* ist ein Plug-In für *Google Chrome*.

   b. *Node.js* ist eine *JavaScript*-Bibliothek, die gerne auf Servern verwendet wird.

   c. *Node.js* ist eine Plattform, auf der *JavaScript*-Programme laufen können.

   d. *Node.js* ist ein Webserver.