Journal

KB ToDo – Single Page Application

Modul 294 – Frau Duc

Verfasser	Kasum Bajrami
Schule	Informatikmittelschule (IMS), Modul 294
Lehrperson	Frau Duc
Projektzeitraum	13.05.2026 – 19.06.2026
Abgabe	19.06.2026
Version	1.0

Inhalt

1. Analyse des Projekts

1.1 Der Auftrag

Im Modul 294 hatte ich die Aufgabe, eine Single Page Application zu bauen. Die App soll TODOs verwalten und folgende elf Felder beinhalten: Titel, Task/Event, Beschreibung, Autor, Kategorie, Wichtig, Dringend, Priorität, Startdatum, Enddatum und Prozentsatz der Erledigung. Die Priorität wird dabei nicht direkt eingegeben, sondern aus den Booleans Wichtig und Dringend nach der Eisenhower-Matrix berechnet.

Funktional muss die App listen, suchen, neu erstellen, ändern (alle Felder auf einmal) und löschen können. Die Daten müssen gespeichert sein, also nach einem Reload immer noch da. Erlaubt sind HTML, CSS und JavaScript – PHP oder ein Backend sind ausgeschlossen.

1.2 Zielgruppe

Die App ist für mich selbst gedacht und für Leute in meinem Alter, die zwischen Schule, Lehrbetrieb und Freizeit jonglieren. Es geht darum, schnell aufzuschreiben was zu tun ist, und dann nach der Eisenhower-Matrix zu sehen welche Aufgabe wichtig genug ist, um sofort gemacht zu werden.

Im weiteren Sinn ist die Lehrperson ebenfalls Zielgruppe, weil sie die App bewerten muss. Darum soll der Code lesbar sein, die App ohne Installation laufen und die Funktionen klar erkennbar sein.

1.3 Warum dieses Thema

Ich hatte mehrere Ideen: eine Watchlist für Filme, eine Finanz-App, eine ToDo-Liste. Ich habe mich für die ToDo-Liste entschieden, weil:

- Die Eisenhower-Matrix ist ein schönes Konzept, das man visuell umsetzen kann (Farbpunkte für Prioritätsstufen).
- Eine ToDo-App deckt automatisch alle Pflicht-Funktionen ab (CRUD, Suchen, Validierung).
- Ich nutze das Resultat selbst im Alltag und das motiviert mehr, als ein erfundenes Beispiel.
- Ich kann zeigen, dass ich mit Datum-Vergleichen, Boolean-Logik und localStorage umgehen kann.

1.4 Funktionale Anforderungen (Muss / Soll / Kann)

Stufe	Anforderung
Muss	Single Page Application ohne Reload
Muss	Alle elf Felder vorhanden und funktionsfähig
Muss	CRUD: Erstellen, Listen, Suchen, Ändern, Löschen
Muss	Priorität wird automatisch aus Wichtig + Dringend berechnet
Muss	Jedes Feld wird mit JavaScript validiert
Muss	Daten werden gespeichert und überleben einen Reload
Muss	HTML und CSS bestehen die W3C-Validierung
Soll	Dark- und Light-Mode mit Toggle
Soll	Responsives Layout (Handy + Desktop)
Soll	Bestätigungs-Modal beim Löschen, damit nichts versehentlich weg ist
Kann	Tastatur-Bedienbarkeit und Focus-Trap im Modal
Kann	Toast-Meldungen für Erfolg / Fehler
Kann	Live-Zeichenzähler beim Tippen

2. Detaillierte Planung

2.1 Zeitplan

Ich habe vom Startdatum bis zur Abgabe rund fünfeinhalb Wochen Zeit. Den Zeitplan habe ich in Etappen aufgeteilt, mit klarer Reihenfolge:

Datum	Etappe	Ergebnis
13. – 17.05.	Analyse + Konzept	Auftrag verstanden, Wireframes auf Papier, Tech-Stack festgelegt
18. – 24.05.	Pflichtenheft	Pflichtenheft als Word-Dokument geschrieben und abgegeben
25. – 29.05.	Masken + Planung	Wireframes digitalisiert, detaillierter Arbeitsplan im Journal
30.05. – 05.06.	HTML + CSS	Grundgerüst der index.html, alle Form-Felder, CSS-Variablen, Light-/Dark-Mode
06. – 12.06.	JavaScript CRUD	localStorage, Liste rendern, Erstellen, Ändern, Löschen, Suchen
13. – 17.06.	Validierung + Tests	JS-Validierung pro Feld, W3C-Check, manuelle Tests, Bug-Fixes
18. – 19.06.	Polishing + Abgabe	Journal fertig, letzte Checks, Abgabe

2.2 Aufteilung der Arbeit

Ich arbeite alleine, also musste ich nichts delegieren. Pro Werktag habe ich mir ungefähr 1 bis 2 Stunden vorgenommen, am Wochenende 3 bis 4 Stunden. Mein Vorsatz war: lieber jeden Tag ein bisschen, statt alles auf den letzten Tag schieben.

Wichtige Regel für mich: Nach jeder Etappe eine Sicherheitskopie der index.html mit Datum im Dateinamen ablegen. So konnte ich notfalls auf den Stand von gestern zurückgehen, falls ich etwas Wichtiges kaputt mache.

2.3 Tools, die ich benutzt habe

- **Visual Studio Code –** Editor für HTML, CSS und JavaScript.
- **Chrome DevTools –** zum Debuggen, vor allem für localStorage-Inhalte (Application-Tab) und Konsolen-Ausgaben.
- **Firefox –** zum Cross-Browser-Test.
- **W3C-Validator (validator.w3.org) –** für die HTML-Validierung.
- **Papier und Bleistift –** für die ersten Wireframes. Schneller als jedes Design-Tool.

3. Konzept

3.1 Architektur

Die ganze App liegt in einer einzigen Datei: index.html. HTML, CSS und JavaScript sind inline. Das hat folgende Vorteile:

- Kein Build-Tool nötig (kein Webpack, kein Vite, kein npm).
- Doppelklick auf die Datei reicht – sie öffnet sich im Standard-Browser.
- Die Lehrperson kann den ganzen Code in einem Tab anschauen, ohne zwischen Dateien zu springen.
- Kein Server, keine Installation, keine Abhängigkeiten.

Die Nachteile: Die Datei wird mit ~2300 Zeilen ziemlich lang. Ich habe das gelöst, indem ich den Code in klar beschriftete Sektionen unterteilt habe (Reset, Header, Form, Liste, Toast, Modal, Footer, Utilities, Responsive). Im JavaScript-Teil gibt es ebenfalls Sektions-Kommentare (Theme, Storage, Validierung, etc.).

3.2 Datenmodell

Ein TODO ist ein JavaScript-Objekt:

```
{ id, title, description, author, category, isEvent,   isImportant, isUrgent, startDate,
endDate, progress, createdAt }
```

Alle TODOs zusammen sind ein Array. Dieses Array wird mit JSON.stringify in einen String umgewandelt und unter dem Schlüssel “kbTodo_data_v1“ im localStorage gespeichert. Beim Laden mache ich JSON.parse mit try/catch, damit kaputte Daten nicht die ganze App abschiessen.

Die Priorität speichere ich bewusst nicht. Sie wird bei jedem Rendern aus isImportant und isUrgent neu berechnet. So gibt es keine inkonsistenten Zustände (z. B. ein TODO, das gespeichert wurde mit Priorität 1, aber wo dann jemand “Wichtig“ entfernt hat).

3.3 Eisenhower-Matrix

Die Logik ist einfach und steckt in einer Funktion:

```
function calculatePriority(isImportant, isUrgent) {   if (isImportant && isUrgent)  return
1; // Sofort   if (isImportant && !isUrgent) return 2; // Einplanen   if (!isImportant &&
isUrgent) return 3; // Delegieren   return 4;                                  // Weg damit }
```

Die Zahl 1 bis 4 bestimmt sowohl die Sortierung in der Liste (1 oben) als auch die Farbe des kleinen Punktes neben dem TODO. Die deutschen Texte stehen in einem Lookup-Objekt PRIORITY_LABELS.

3.4 Sichten und Navigation

Die App hat zwei Sichten, zwischen denen man mit Tabs umschaltet:

- Sicht 1 – Meine TODOs: Suchfeld plus Liste von TODO-Karten, sortiert nach Priorität.
- Sicht 2 – Erstellen / Bearbeiten: Formular mit allen Feldern, das je nach Modus seinen Titel ändert.

Die Sichten sind technisch zwei section-Elemente. Die jeweils sichtbare Sicht trägt die Klasse .active. In CSS gilt: .view { display: none; } .view.active { display: block; }. JavaScript schaltet diese Klasse um – kein Reload, keine URL-Änderung. Das ist die ganze SPA-Logik.

3.5 Design-Entscheidungen

- Light-Mode als Standard und Dark-Mode per Toggle.
- Farbsystem über CSS-Variablen (--bg, --text, --priority-1 ...). Dark-Mode überschreibt die gleichen Variablen – kein Element muss zweimal gestylt werden.
- Schlankes Layout: maximal 960 px breit, mit viel Weissraum. Kein überladenes Design.
- Prioritäts-Farben: Rot, Orange, Gelb, Grau. Standard-Symbolik – auch jemand ohne Brille erkennt sofort, was wichtig ist.
- Toast-Meldungen oben rechts: Erfolg, Fehler oder Info. Verschwinden nach drei Sekunden automatisch.

4. Phasen der Arbeit

Phase 1 – Analyse und Konzept

Ich habe den Auftrag dreimal gelesen und die elf Pflicht-Felder auf ein Blatt geschrieben. Danach habe ich auf Papier kleine Wireframes der zwei Sichten gezeichnet, um zu verstehen wohin welches Feld kommt. Erst dann habe ich angefangen, an Farben oder Schriften zu denken.

Wichtigste Erkenntnis aus dieser Phase: Erstellen und Bearbeiten müssen sich ein einziges Formular teilen. Sonst dupliziere ich Code und mache Fehler. Ein verstecktes Feld todoId entscheidet, ob es ein neuer Eintrag oder ein Update ist.

Phase 2 – HTML-Struktur

Ich habe zuerst den head, dann den Header, dann die zwei sections in main aufgebaut. Wichtig war mir, semantisch sauber zu sein: header, main, section, article, footer statt überall nur divs. Das hilft Screenreadern und macht den Code lesbarer.

Für das Formular habe ich jedes Feld mit einem dazugehörigen label verknüpft (label for="..."). Pflichtfelder bekommen ein required und ein Sternchen. Maxlength-Werte habe ich vom Auftrag übernommen (Titel 255, Autor 20).

Phase 3 – CSS-Design

Ich habe mit den CSS-Variablen angefangen, dann das Layout (Flex / Grid) und erst zum Schluss die Details (Border-Radius, Shadows). Mobile-Anpassungen über eine media-query bei 720 px. Dark-Mode über data-theme="dark" auf html – damit ist der Mode-Switch eine einzige Attribut-Änderung.

Phase 4 – JavaScript: CRUD und localStorage

Ich habe das JavaScript in Funktionen aufgeteilt: getTodos, saveTodos, findTodoById für den Storage; renderCard, renderList, refreshList fürs Anzeigen; handleSubmit, handleListClick, handleConfirmDelete für die Events. Ein einziger initApp am Ende verkabelt alles.

Pattern, das ich verwende: Event-Delegation. Statt jeden Lösch-Button mit einem eigenen Listener auszustatten, hängt ein Listener am ganzen Listen-Container. event.target.closest('button[data-action]') findet heraus, was geklickt wurde. Vorteil: Karten, die später hinzukommen, funktionieren automatisch mit.

Phase 5 – Validierung und XSS-Schutz

Ich habe pro Feld eine eigene validateXxx-Funktion gemacht, die einen leeren String zurückgibt wenn alles OK ist, oder eine deutsche Fehlermeldung sonst. validateForm ruft alle einzeln auf und sammelt die Fehler in einem Objekt. Anschliessend wird displayErrors aufgerufen, das die Fehler ans richtige Feld schreibt und einen roten Rahmen setzt.

Für den XSS-Schutz habe ich eine escapeHtml-Funktion eingebaut. Sie wird vor jedem User-Inhalt aufgerufen, der mit innerHTML ins DOM kommt. Wenn jemand <script>alert(1)</script> eintippt, wird der Text als Text gerendert und nicht als Code ausgeführt.

Phase 6 – Polishing und W3C-Validierung

Erst ganz am Schluss habe ich die App durch den W3C-Validator geschickt. Das war ein kleiner Schock, weil mehrere Fehler kamen, die ich vorher nicht erwartet hatte – zum Beispiel, dass meta charset zu spät stand. Diese Probleme habe ich behoben und die Datei jetzt ist sauber durch.

5. Verwendete Funktionen

Hier eine Übersicht der wichtigsten JavaScript-Funktionen aus meiner index.html, gruppiert nach Bereich:

5.1 Theme-Verwaltung

Funktion	Was sie macht
`setTheme(theme)`	Setzt data-theme auf html und speichert die Wahl im localStorage.
`toggleTheme()`	Wechselt zwischen 'light' und 'dark'.
`initTheme()`	Beim Start: gespeicherte Wahl benutzen, sonst System-Präferenz (matchMedia).

5.2 Navigation

Funktion	Was sie macht
`showView(name)`	Schaltet zwischen 'list' und 'create'. Aktualisiert auch aria-selected.

5.3 Speicherung

Funktion	Was sie macht
`getTodos()`	Liest aus localStorage, JSON.parse mit try/catch. Liefert leeres Array bei kaputten Daten.
`saveTodos(arr)`	Speichert das ganze Array mit JSON.stringify zurück.
`findTodoById(id)`	Hilfsfunktion: liefert ein einzelnes TODO oder undefined.

5.4 Priorität und Validierung

Funktion	Was sie macht
`calculatePriority(w, d)`	Eisenhower-Matrix: gibt 1-4 zurück.
`validateTitle / Author / Description / Category / StartDate / EndDate / Progress / Type`	Pro Feld eine eigene Funktion. Liefert leeren String oder Fehlertext.
`validateForm(data)`	Ruft alle Einzel-Validatoren auf und sammelt die Fehler in einem Objekt.
`displayErrors(errors)`	Schreibt die Fehlertexte in die error-msg-divs und setzt .invalid auf die Inputs.

5.5 Anzeige und Formular

Funktion	Was sie macht
`renderCard(todo)`	Erzeugt eine TODO-Karte (article) mit allen Inhalten und Buttons.
`renderList(todos)`	Leert die Liste, sortiert nach Priorität, rendert jede Karte.
`refreshList()`	Holt alle TODOs, filtert nach Suchtext, ruft renderList auf.
`readForm() / resetForm() / fillForm(todo)`	Daten aus dem Formular lesen / Formular leeren / Formular mit bestehendem TODO füllen.
`updatePriorityDisplay()`	Live-Update der Prioritäts-Anzeige im Formular, wenn Wichtig/Dringend geändert wird.
`updateCharCounters()`	Aktualisiert die Live-Zähler unter Titel, Autor, Beschreibung.

5.6 Events und Modal

Funktion	Was sie macht
`handleSubmit(event)`	Submit-Handler: liest, validiert, speichert (CREATE oder UPDATE).
`handleListClick(event)`	Event-Delegation: erkennt ob Bearbeiten oder Löschen geklickt wurde.
`openDeleteModal / closeModal / handleConfirmDelete`	Lösch-Modal mit Fokus-Rückgabe an den Trigger-Button.
`handleModalKeydown`	Escape schliesst das Modal, Tab fängt den Fokus innerhalb des Modals (Focus Trap).
`syncProgressInputs(src)`	Hält Range-Slider und Number-Input synchron.

Funktion	Was sie macht
`showToast(msg, type)`	Zeigt eine Toast-Meldung oben rechts, die nach 3 Sekunden verschwindet.
`escapeHtml(str)`	XSS-Schutz: wandelt < > & in HTML-Entities um.
`formatDate(iso)`	Wandelt 2026-05-21 in 21.05.2026 (deutsches Format).
`initApp()`	Wird bei DOMContentLoaded aufgerufen, hängt alle Event-Listener an.

6. Aufgetretene Probleme und ihre Lösung

Problem 1 – W3C-Validator: charset zu spät

Symptom: Fatal Error “Changing encoding at this point would need non-streamable behavior“ und “charset attribute on a meta element found after the first 1024 bytes“.

Ursache: Ich hatte oben in der Datei zwei lange Kommentar-Blöcke mit Erklärungen für die Lehrperson. Dadurch war das

```
<meta charset="UTF-8">
```

erst auf Zeile 24 und nach Byte ~1100 – zu spät. HTML5 verlangt, dass charset innerhalb der ersten 1024 Bytes steht.

Lösung: DOCTYPE und meta charset ganz nach oben gesetzt, die Erklärungs-Kommentare nach unten (nach dem title) verschoben. meta charset steht jetzt bei Byte 60. Fehler weg.

Problem 2 – W3C-Warning: zwei Bindestriche im Kommentar

Symptom: “The document is not mappable to XML 1.0 due to two consecutive hyphens in a comment.“ Acht solche Warnungen.

Ursache: In meinen Kommentaren hatte ich Trennlinien wie ---------- für Abschnitte. Das sind XML-technisch ungültige Sequenzen, weil -- der Anfang von -->

Lösung: Bindestriche durch Gleichheitszeichen (==========) ersetzt in den HTML-Kommentaren. In CSS- und JS-Kommentaren spielt es keine Rolle, weil sie nicht XML-geparst werden.

Problem 3 – W3C-Warning: aria-required redundant

Symptom: “Attribute aria-required is unnecessary for elements that have attribute required.“

Ursache: Ich hatte sowohl required als auch aria-required="true" gesetzt. HTML5 setzt aria-required automatisch, sobald required vorhanden ist.

Lösung: aria-required entfernt überall dort, wo required steht. Resultat: alle Pflichtfelder funktionieren weiter, Validator ist still.

Problem 4 – Enddatum kann vor Startdatum liegen

Symptom: Ein TODO konnte mit Startdatum 20.05. und Enddatum 15.05. gespeichert werden. Logisch falsch.

Lösung: Cross-Field-Validierung in validateEndDate. Beide Daten werden in Date-Objekte umgewandelt und verglichen:

```
if (startDate && new Date(endDate) < new Date(startDate)) {   return 'Enddatum muss nach dem Startdatum liegen.'; }
```

Problem 5 – localStorage mit kaputten Daten

Symptom: Beim Testen hatte ich manuell den localStorage-Eintrag in den DevTools verändert. Beim Laden warf JSON.parse einen Fehler und die App startete nicht mehr.

Lösung: getTodos in einen try/catch gewickelt. Bei einem Fehler logge ich in die Konsole und gebe ein leeres Array zurück. Die App startet trotzdem.

Problem 6 – Bearbeiten-Modus zeigt 'Speichern' statt 'Aktualisieren'

Symptom: Beim Klick auf 'Bearbeiten' füllte sich das Formular zwar, aber der Button-Text und der Formular-Titel blieben gleich – verwirrend.

Lösung: In fillForm setze ich formTitle.textContent = 'TODO bearbeiten' und submitBtn.textContent = 'Aktualisieren'. In resetForm setze ich beide zurück.

Problem 7 – Range-Slider und Number-Input nicht synchron

Symptom: Ich hatte zwei Eingaben für den Fortschritt: einen Slider und ein Number-Feld. Wenn ich am Slider zog, blieb das Number-Feld stehen.

Lösung: Eine syncProgressInputs(source)-Funktion. Beide Inputs haben einen input-Listener und rufen sie auf. Werte werden auf 0-100 begrenzt, falls jemand 999 eintippt.

Problem 8 – XSS durch User-Eingabe

Symptom (theoretisch): Wenn ich einen Titel wie

```
<script>alert('Hacked')</script>
```

eingebe und der Titel mit innerHTML ins DOM kommt, würde der Browser den Code ausführen.

Lösung: Eine escapeHtml-Funktion, die ich auf jeden User-Inhalt anwende, bevor er ins DOM kommt. Sie nutzt den Trick mit document.createElement('div') + textContent + innerHTML – der Browser escaped automatisch.

Problem 9 – Code-Einrückung verrutscht

Symptom: Nach einer Edit-Session war die Einrückung im JavaScript-Block ziemlich kaputt. Der Code lief, aber lesen war Stress.

Lösung: Den ganzen JS-Teil manuell durchgegangen und auf konsistente Einrückung gebracht (vier Spaces für Funktionen, sechs für Bodies). Seither ist auch das Code-Lesen wieder angenehm.

7. Durchgeführte Tests

Ich habe systematisch manuelle Tests gemacht. Pro Funktion eine Testreihe, mit Soll- und Ist-Wert. Hier die wichtigsten:

#	Was wurde getestet	Erwartetes Ergebnis	Ist
1	Neues TODO mit allen Feldern erstellen	Erscheint in der Liste	OK
2	Leeres Formular abschicken	Fehlermeldungen pro Pflichtfeld	OK
3	Titel mit 300 Zeichen eintippen	maxlength stoppt bei 255	OK
4	Enddatum vor Startdatum	Fehler beim Enddatum	OK
5	Fortschritt 150 im Number-Feld	Wird auf 100 begrenzt	OK
6	Wichtig + Dringend setzen, Priorität ansehen	Zeigt 'Sofort erledigen'	OK
7	TODO bearbeiten, Titel ändern, speichern	Update statt neuer Eintrag, gleicher Platz	OK
8	Löschen-Knopf klicken, Modal kommt	Modal offen, Fokus auf 'Löschen'	OK
9	Im Modal Escape drücken	Modal schliesst, Fokus zurück auf Trigger	OK
10	Browser neu laden	Alle TODOs noch da	OK
11	Suche nach Teil des Titels	Nur passende TODOs in Liste	OK
12	Suche nach Autor in Grossbuchstaben	Case-insensitive, findet trotzdem	OK
13	Dark-Mode-Toggle klicken	Theme wechselt sofort, bleibt nach Reload	OK
14	XSS-Test: Titel = <script>alert(1)</script>	Wird als Text angezeigt, kein Alert	OK
15	Layout im Handy-Format (375 px)	Formular einspaltig, keine Überläufe	OK
16	W3C-HTML-Validator	Keine Errors, keine Warnings	OK
17	W3C-CSS-Validator	Keine Errors	OK
18	Test in Chrome, Firefox, Edge	Funktioniert in allen	OK

Alle 18 Tests sind durchgegangen. Die Tests habe ich nicht nur einmal am Schluss gemacht, sondern jeweils nach grösseren Änderungen wiederholt, damit ich Regressionen früh sehe.

8. Zusammenfassung und was ich gelernt habe

8.1 Zusammenfassung des Arbeitsauftrags

Mein Auftrag war eine Single Page Application zur TODO-Verwaltung mit Eisenhower-Logik, elf Pflichtfeldern, CRUD-Funktionen, JS-Validierung, localStorage und W3C-konformem HTML/CSS. Plus ein Pflichtenheft und dieses Journal.

Resultat: eine Datei index.html mit ungefähr 2300 Zeilen HTML, CSS und JavaScript. Alle elf Felder funktionieren, alle Aktionen laufen ohne Reload, die Daten sind persistent, die App ist responsive, der Code ist auf Deutsch kommentiert. Die App besteht den W3C-Validator ohne Errors und ohne Warnings. Dark- und Light-Mode sind integriert, ein Bestätigungs-Modal verhindert versehentliches Löschen, und Toast-Meldungen geben dem Benutzer Feedback.

8.2 Was ich technisch gelernt habe

- **Vanilla JavaScript reicht weit.** Vorher dachte ich, man braucht für eine seriöse App ein Framework wie React. Aber für ein Projekt dieser Grösse ist Vanilla JS sogar lesbarer und einfacher zu debuggen.
- **Event-Delegation.** Statt jeden Button einzeln zu verkabeln, einen Listener am Container und mit closest() filtern. Sauberer Code, weniger Listener, funktioniert auch mit dynamisch hinzugefügten Elementen.
- **CSS Custom Properties.** Eine Variable an einer Stelle ändern, und der ganze Dark-Mode funktioniert. Das hätte ich nicht so einfach geglaubt.
- **Accessibility ist nicht optional.** Labels, aria-Attribute, Focus-Trap im Modal, Tastatur-Bedienung – das macht die App nicht nur barrierefrei, sondern auch für mich beim Testen einfacher.
- **XSS ist real.** Vor diesem Projekt habe ich XSS für ein theoretisches Problem gehalten. Jetzt weiss ich: jede App, die innerHTML mit User-Eingaben verbindet, ist potenziell verwundbar.
- **W3C-Validierung ist kein Selbstläufer.** Auch sauber aussehender Code kann subtile Fehler haben – charset-Reihenfolge, doppelte Bindestriche in Kommentaren, redundante aria-Attribute. Früh validieren ist Gold wert.

8.3 Was ich über Projekt-Arbeit gelernt habe

- **Planen zahlt sich aus.** Die Stunde, die ich auf Papier-Wireframes verbracht habe, hat mir später drei Stunden Refactoring erspart.
- **Inkrementelles Vorgehen.** Eine Funktion fertig, testen, dann die nächste. Wenn etwas kaputt geht, weiss ich genau, wo der Fehler ist.
- **Kommentieren ist für mich selber.** Wenn ich nach zwei Wochen zurückschaue, ist mein Kommentar oft die einzige Chance zu verstehen, was ich gedacht habe.
- **Sicherheitskopien.** Bei jedem grossen Schritt eine Kopie wegspeichern. Hat mich schon mehr als einmal gerettet.
- **Den Auftrag wirklich lesen.** Ich habe den Auftrag drei Mal gelesen, bevor ich angefangen habe – und trotzdem habe ich im Verlauf mehrfach nachgesehen, ob ich Punkte vergessen habe.

8.4 Was ich beim nächsten Mal anders machen würde

- Den W3C-Validator schon zur Hälfte des Projekts laufen lassen, nicht erst am Schluss.
- Vor dem Coden eine kleine Test-Checkliste anlegen, statt während des Codens Tests zu erfinden.
- Mit Git arbeiten statt mit datierten Kopien. Hatte ich nicht eingerichtet, würde ich beim nächsten Modul sicher tun.
- Das Journal nicht ganz am Schluss schreiben, sondern als laufendes Tagebuch. Sonst vergesse ich Details über aufgetretene Probleme.