

Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.
©2018 Beuth Hochschule für Technik Berlin

JSL - Interaktionssteuerung mit Javascript



Überblick und Lernziele

In der vorangegangenen Lerneinheit haben Sie sich damit beschäftigt, wie die Ansichten einer Webanwendung und die Struktur der darin darzustellenden Inhalte mit HTML modelliert werden können und wie Ansichten und Inhalte mittels CSS in eine visuelle Darstellung überführt werden können.

Die vorliegende Lerneinheit wird Ihnen nun mit JavaScript eine weitere Sprache vorstellen, die es Ihnen darüber hinaus erlaubt, die Interaktion eines Nutzers mit den in einer Ansicht dargestellten Inhalten und den Bedienelementen einer Ansicht zu steuern, d. h. Ihre Anwendung darauf reagieren zu lassen, dass ein Nutzer mit ihr interagiert. Wir werden außerdem zeigen, wie JavaScript eingesetzt werden kann, um die Ansichten einer Anwendung dynamisch aufzubauen und dafür Daten zu verwenden, die der im Browser ausgeführten Webanwendung von einer server-seitigen Anwendungskomponente geliefert werden.

Auch in dieser Lerneinheit setzen wir voraus, dass Sie bereits über Kenntnisse in der Anwendung von JavaScript verfügen und dass Ihnen insbesondere die Syntax sowie die wichtigsten Operatoren von JavaScript vertraut sind. In dieser Lerneinheit ist daher nur eine sehr kurze Übersicht über die Ausdrucksmittel von JavaScript enthalten – falls erforderlich, verweisen wir dafür auf existierendes frei verfügbares [Material im Netz](#) bzw. einschlägige weiterführende Publikationen, z. B. [\[Osm12\]](#).

Unsere eigene Darstellung basiert auf einer Auslegung des Model View Controller Architekturmusters für Anwendungen mit Nutzerschnittstelle, anhand dessen sich die Einsatzgebiete von JavaScript zur Interaktionssteuerung sehr deutlich illustrieren lassen. Dabei werden wir auch darauf eingehen, wie mit den aktuell verfügbaren Ausdrucksmitteln standardisierter Webtechnologien eine sinnvolle Arbeitsteilung zwischen HTML, CSS und JavaScript umgesetzt werden kann.

Zunächst werden wir in Kürze einige Merkmale von JavaScript darstellen, die für die Sprache – auch in Abgrenzung zu den beiden anderen bisher betrachteten Sprachen HTML und CSS – charakteristisch sind.



Lernziele

Nachdem Sie die Lerneinheit durchgearbeitet haben, sollten Sie in der Lage sein:

- Die Merkmale von JavaScript zu nennen und gegenüber anderen Programmiersprachen abzugrenzen.
- Zu erläutern warum JavaScript eine wichtige Rolle bei der Entwicklung von mobilen Webanwendungen spielt und auf welche Weise sie mit HTML5 und CSS zusammenwirkt.
- Zu erklären was eine Fat Client Architektur beinhaltet
- Ausdrucksmittel von JavaScript zu kennen, sie verfügbar zu machen und zu verwenden
- Mit JavaScript Interaktion eines Nutzers mit einer *graphischen Nutzerschnittstelle* (GUI) zu steuern. Das heißt, auf Nutzereingaben zu *reagieren*, die Daten zu *verarbeiten*, und dem Nutzer *Feedback* bezüglich seiner Eingabe zu geben.
- Die Phasen der Ereignisbehandlung zu kennen und praktisch anzuwenden.
- DOM-Manipulationen durchzuführen
- Zu erläutern wie mit JavaScript HTTP-Requests aufgebaut werden können.



Gliederung

Gliederung

- Merkmale von JavaScript
- Interaktionssteuerung
- Abstimmung von HTML, CSS und JavaScript
- Zusammenfassung
- Wissensüberprüfung
- Übungen



Zeitbedarf

Zeitbedarf und Umfang

Für die Bearbeitung der Lerneinheit benötigen Sie etwa 4 Stunden. Für die Bearbeitung der Übungen JSL und JSR für die Beispielanwendung etwa 6 Stunden und für die Wissensfragen ca. 2 Stunden.

1 Merkmale von JavaScript

Entwickelt wurde JavaScript ursprünglich in den 90er Jahren durch das Unternehmen Netscape, das mit dem Netscape Navigator so etwas wie den „Urvater des Firefox Browsers“ entwickelt hat. Die Namensgebung „JavaScript“ war dabei jedoch von Anfang an irreführend, da die Sprache lediglich einige syntaktische Merkmale mit Java gemeinsam hat, die allerdings auch in anderen Sprachen, z. B. C, verwendet werden. Dies betrifft insbesondere die Syntax für die elementaren Anweisungen, die in JavaScript für Wertzuweisungen, Verzweigungen, Schleifen und Funktionsaufrufe verwendet werden können.

Motiv der Namensgebung dürfte denn auch – zumindest unter anderem – die beginnende Java Euphorie der 90er Jahre gewesen sein, im Hinblick auf welche die Namenswahl eine hohe Öffentlichkeitswirksamkeit garantierte.

Seit 1996/1997 wird die Arbeit an den Ausdrucksmitteln von JavaScript von der Organisation ECMA („European association for standardizing information and communication systems“) koordiniert. Daher kommt die Bezeichnung „ECMAScript“, welche den voll standardisierten Anteil von JavaScript umfasst. Diesbezüglich findet sich bei Mozilla die folgende Aussage: „*The JavaScript standard is ECMAScript*“

 <https://developer.mozilla.org/de/docs/JavaScript>

Die Bezeichnung „JavaScript“ ist hingegen zwar weitaus häufiger gebraucht, aber weniger aussagekräftig als „ECMAScript“, da darin browserspezifische Abweichungen vom Standard enthalten sein können. Im folgenden werden wir dennoch mit „JavaScript“ die unschärfere, aber vertrautere Bezeichnung verwenden.

1.1 JavaScript als imperative Programmiersprache?

JavaScript war zunächst vor allem als „punktuelle Ergänzung“ von HTML gedacht und sah die Implementierung einfacher Skripte vor, um auf die Interaktion eines Nutzers mit einer Webanwendung ohne Neuladen des dargestellten Dokuments reagieren zu können. Diese „Steuerung“ kam jedoch eher der Charakter einer „optionalen Dekoration“ zu, die für die Bedienbarkeit und das Funktionieren einer Webanwendung im Idealfall nicht notwendigerweise erforderlich sein durfte – so konnte der Nutzer bereits in der Frühphase der öffentlichen Verbreitung von Webtechnologien die Ausführung von JavaScript durch den Browser deaktivieren.

AJAX

Mit zunehmender Leistungsfähigkeit der Endgerätehardware, optimierten Browserimplementierungen und Erweiterungen des Funktionsumfangs von JavaScript wurde von den Ausdrucksmitteln der Sprache jedoch zunehmend Gebrauch gemacht. Auf den Punkt gebracht wird diese Entwicklung durch den gegen die Mitte der Nuller Jahre geprägten Begriffs AJAX. Wie wir unten sehen werden, verbindet AJAX (*Asynchronous JavaScript and XML*) insbesondere den Zugriff auf serverseitige Daten und Inhalte, die clientseitig zum Zweck einer Manipulation bzw. Aktualisierung der dargestellten Ansichten verwendet werden, ohne dass dafür ein Neuladen des im Browser dargestellten HTML- Dokuments und damit der Neuaufbau der Ansicht erforderlich ist.

Durch die zunehmende Verbreitung von AJAX Implementierungsmustern und der damit ermöglichten „Dynamisierung“ der im Browser dargestellten Ansichten wurde JavaScript zunehmend in den Stand einer umfassenden Programmiersprache für die Steuerung clientseitig ausgeführter Anwendungen mit User Interface erhoben. Wie wir einleitend dargelegt haben, liegt eine solche „Fat Client“ Architektur auch der in dieser Lehrveranstaltung vermittelten Anwendung von JavaScript zugrunde. Verstärkt wird die Tendenz, JavaScript als vollgültige Programmiersprache zu behandeln, auch durch die Verfügbarkeit serverseitiger Ausführungsumgebungen für JavaScript wie NodeJS, mit der wir uns in der kommenden Lerneinheit beschäftigen werden.

Gewöhnliche
Programmierersprache

Im Gegensatz zu HTML und CSS dürfte JavaScript vermutlich am ehesten dem entsprechen, was Sie sich unter einer „gewöhnlichen Programmiersprache“ vorstellen, und tatsächlich haben wir es bei JavaScript mit einem dritten Typ von Programmiersprachen zu tun. Diente HTML als Auszeichnungssprache der Zuschreibung von darstellungsbezogenen Strukturmerkmalen an textuelle Inhalte zum Zweck des Aufbaus einer visuellen Ansicht im Browser, so konnten wir mit CSS als Regelsprache Bedingungen über diesen Merkmalen formulieren und abhängig davon einem HTML Dokument konkrete Gestaltungseigenschaften zuweisen.

JavaScript hingegen ist zunächst eine imperative Sprache, die uns eine Menge von Anweisungen – u. a. die bereits oben erwähnten Anweisungen für Wertzuweisungen, Verzweigungen, Schleifen und Funktionsaufrufe – bereitstellt. Diese Anweisungen können verwendet werden, um das Verhalten einer Anwendung im Anschluss an eine Nutzerinteraktion zu beschreiben. Dies schließt den Aufbau und die Modifikation des einer Ansicht zugrunde liegenden HTML Dokuments beim Starten bzw. im Laufe der Bedienung durch den Nutzer ausdrücklich mit ein.

Wie wir sehen werden, kann die konkrete Verwendung der Sprache durchaus aber auch Aspekte regelbasierter Programmierung beinhalten. Dies gilt insbesondere für die Deklaration von *Event Handlern*, mittels derer „Reaktionsregeln“ für das Verhalten einer Anwendung im Fall von Nutzerinteraktion beschrieben werden können.

1.2 Ausdrucksmittel

Typisierung

Hinsichtlich seiner Ausdrucksmittel zeichnet sich JavaScript gegenüber anderen Sprachen wie Java oder Objective-C dadurch aus, dass es keine Typzuweisungen an Variablen inklusive an die einer Funktion übergebenen Argumente erlaubt. Dies bedeutet jedoch keineswegs, dass JavaScript „typenlos“ ist o. ä. Vielmehr besagt es nur, dass die betreffenden Bezeichner nicht eingeschränkt sind bezüglich des Typs der Werte, auf die sie verweisen. So können z. B. einer Variablen hintereinander Werte verschiedener in JavaScript vorgesehener Typen zugewiesen werden, ohne dass dies in Fehlern beim Laden oder bei der Ausführung des betreffenden Skripts resultiert. Fehler können aber zur Laufzeit durchaus auftreten, wenn der Wert einer Variable in einer Weise behandelt wird, die nicht seinem Typ entspricht. So könnten z. B. die beiden Literale `0.5` und `„0.5“` bei einem vollständigen Verzicht auf Typen als gleichwertig betrachtet werden; in JavaScript werden sie jedoch nicht notwendigerweise in diesem Sinne behandelt.

Wertvergleich

Für den Vergleich von Werten erleichtert Ihnen JavaScript die Handhabung verschiedener Typen allerdings dadurch, dass bei Anwendung der meisten Vergleichsoperatoren wie `==`, `<`, `>`, etc. eine automatische Typumwandlung der Operanden vorgenommen wird. Beispielsweise wird bei Vorliegen eines numerischen Operanden ein etwaiger String-wertiger Operand versuchsweise in einen numerischen Wert umgewandelt und ggf. damit die Überprüfung durchgeführt. Soll der Typ hingegen bei einer Überprüfung auf Wertgleichheit berücksichtigt werden, können Sie dafür den „strikten“ Vergleichsoperator `===` verwenden.

Vergleichsoperatoren in JavaScript

Eine detailliertere Darstellung der Vergleichsoperatoren in JavaScript finden Sie z. B. auf [www.https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators).

Und falls Sie selbst z. B. den durch einen String repräsentierten numerischen Wert ermitteln möchten, stehen Ihnen dafür Funktionen wie `parseFloat()` zur Verfügung, die Sie vermutlich auch aus Java kennen. Damit...

- ... evaluiert der Ausdruck `(0.5 == „0.5“)` zu `true`
- ... evaluiert der Ausdruck `(0.5 === „0.5“)` zu `false`
- ... evaluiert der Ausdruck `(0.5 === parseFloat(„0.5“))` zu `true`

Neben den primitiven Datentypen `Number`, `String` und `Boolean` sieht JavaScript die beiden Referenzdatentypen `Array` und `Object` sowie die beiden „speziellen“ Datentypen `Undefined` und `Null` vor. Letztere drücken die Tatsache aus, dass bezüglich einer Variable noch keine Wertzuweisung erfolgt ist, bzw. dass eine Referenzvariable nicht auf einen Wert referenziert. Im letzteren Fall resultiert der Aufruf einer vermeintlich vorhandenen Funktion auf dem Variablenwert bzw. der Zugriff auf ein darauf verfügbares Attribut in einer `NullPointerException`, wie Sie sie aus Java kennen.

Objekte

Der Datentyp `Object` wird in JavaScript verwendet, um Instanzen komplexer Datentypen zu repräsentieren, wie Sie sie aus der objektorientierten Programmierung kennen. Zur Notation von Objekten steht uns die sogenannte *JavaScript Object Notation* zur Verfügung, die unter dem Akronym JSON wohlbekannt ist. Kommaseparierte Mengen von „:“-separierten Key-Value-Pairs werden hier innerhalb von geschweiften Klammern zu Objekten gruppiert, z. B.:

```
001 // instantiiere ein Objekt
002 var obj = {attr1: "val1", attr2: val2ref, attr3:["e1", "e2"], attr4: {
003   attr41: "val41"}}
```

JSON

Eine Besonderheit von JSON besteht darin, dass uns JavaScript die beiden Funktionen `JSON.parse()` und `JSON.stringify()` zur Verfügung stellt, mittels derer aus einem String in JSON Notation ein entsprechendes Objekt aufgebaut bzw. ein Objekt in einen String überführt werden kann. Wie wir unten sehen werden, werden diese beiden Funktionen u. a. beim Datenaustausch zwischen einer im Browser ausgeführten Webanwendung und etwaigen serverseitigen Anwendungskomponenten verwendet.

Funktionsobjekte

Objekte können in JavaScript sowohl Attribute, als auch „Methoden“ zur Verfügung stellen und eignen sich damit grundsätzlich für die Kapselung von Daten und Verhalten, wie Sie sie ebenfalls in der Beschäftigung z. B. mit Java kennengelernt haben. Grundlage für die Implementierung von „Methoden“ ist dabei die Tatsache, dass in JavaScript Funktionen als *Funktionsobjekte* repräsentiert werden, die damit auch an jeder Stelle verwendet werden können, wo Objekte auftreten können, z. B. als Wert des Attributs eines Objekts. Und da die Implementierung von Funktionen innerhalb von Objekten Zugriff auf Werte von Attributen und andere Variablen des enthaltenden Objekts hat, lässt sich damit die Funktionsweise von Instanzmethoden, die Verhalten über *Instanzattributen* implementieren, auch in JavaScript umsetzen.

Das nachfolgende Beispiel zeigt, wie eine durch ein Funktionsobjekt namens `funcobj` repräsentierte Funktion aufgerufen werden kann. Verwendet wird hierfür das Postfix „`()`“ innerhalb dessen ggf. die Werte von Argumenten referenziert werden können, die der Funktion übergeben werden sollen:



Quellcode

Funktionsobjekt funcobj

```
001 function func2(funcobj) {
002     /*...*/
003     // rufe die an die Funktion übergebene Funktion auf und nimm ihren
004     // Rückgabewert entgegen:
005     var returnval= funcobj(Argumente);
006     /*...*/
007 }
```

Die Verfügbarkeit einer Funktion auf einem Objekt kann dabei analog zum Wertzugriff auf Attribute überprüft werden – beachten Sie hier, dass die Evaluierung einer Variable in JavaScript eine gültige boolesche Expression ist und mithin die Überprüfung der Verfügbarkeit wie folgt durchgeführt werden kann.

```
001 // überprüfe, ob auf obj die Funktion namens func gesetzt ist und rufe
002 // diese ggf. auf:
003 if (obj.func) {
004     obj.func();
005 }
```

Genau genommen überprüft dieses Beispiel allerdings nur, ob ein Attribut namens `func` auf `obj` gesetzt ist und nicht, ob es sich bei diesem Attribut tatsächlich auch um eine Funktion handelt. Falls das Attribut auf einen Wert eines anderen Typs gesetzt ist, würde der vermeintliche Funktionsaufruf `func()` daher einen Laufzeitfehler hervorrufen. Soll dies vermieden werden, kann zusätzlich zur Überprüfung des Vorliegens des Attributs die folgende Typüberprüfung angewendet werden, für die uns JavaScript den `typeof` Operator zur Verfügung stellt:

```
001 if (obj.func && typeof obj.func == "function") {
002     obj.func();
003 }
```

Die Vermeidung von Laufzeitfehlern durch eine solche strikte Überprüfung inklusive Typüberprüfung ist jedoch nicht immer praktikabel. So weist ein Laufzeitfehler in den meisten Fällen unübersehbar auf einen Programmierfehler hin und kann Ihnen damit gerade in der Entwicklungsphase einer Anwendung ggf. das Debugging erleichtern.

Für Details zur Verwendung von JavaScript als objektorientierter Sprache, die es auch erlaubt, durch Verwendung von *Prototypenfunktionen* Vererbungsbeziehungen zu implementieren, verweisen wir auf die einschlägige Literatur die wir bereits eingangs zitiert haben. Wozu Funktionsobjekte in JavaScript unabhängig von objektorientierten Programmierprinzipien verwendet werden können, werden Sie im Verlauf dieser Lerneinheit, aber vor allem auch in den folgenden Lerneinheiten erfahren.

Eine Übersicht über einige Ausdrucksmittel von JavaScript finden Sie in folgender Tabelle.

Wertzuweisung	<code>currenttime = Literalwert oder Auswertung;</code>
Verzweigung	<pre> if (Bedingung) { Folge von Anweisungen } else { Folge von Anweisungen } switch (Auswertung) { case Auswertung 1: Anweisungen 1 case Auswertung 2: Anweisungen 2 ... default: Default-Anweisungen } </pre>
Iteration	<pre>for (Laufvariable;Bedingung;Inkrement) { Folge von Anweisungen }</pre>
Schleife	<pre> while (Bedingung) { Folge von Anweisungen } bzw. do { Folge von Anweisungen } while (Bedingung) </pre>
Aufruf	<code>Funktionsname(Argumente)</code>
Rückgabe	<code>return Rückgabewert;</code>
Sequenz	<pre> Anweisung 1; Anweisung 2; ... </pre>
Funktionsdeklaration	<pre>function Funktionsname(Argumente) { Anweisungen }</pre>

Tab.: Notation von Anweisungen in JavaScript

Im `switch` Statement enthalten die Anweisungsblöcke `Anweisungen 1`, `Anweisungen 2`, etc. üblicherweise ein finales `break`; Default-Anweisungen sind optional.

1.3 Verfügbarmachung und Verwendung von JavaScript

Ähnlich wie CSS-Regeln können auch Anweisungen und Funktionen in JavaScript wahlweise innerhalb eines HTML-Dokuments als textueller Inhalt eines `<script>` Elements deklariert werden oder in Form externer JavaScript Dateien in ein HTML-Dokument „importiert“ werden, z. B.



Quellcode

Import von CSS und JavaScript

```
001 <head>
002   <title>Interaktionssteuerung mit JavaScript</title>
003   <!-- importiere ein Stylesheet -->
004   <link rel="stylesheet" href="css/demo.css"/>
005   <!-- importiere ein JavaScript Skript -->
006   <script type="text/javascript" src="js/demo.js"></script>
007 </head>
```

Ausführung

Die Inhalte eines `<script>` Elements werden normalerweise beim Laden des HTML-Dokuments unmittelbar im Zuge der Verarbeitung des Elements „an Ort und Stelle“ durch den Browser ausgeführt – unabhängig davon, ob alle anderen Elemente bereits verarbeitet worden sind. In diesem Fall ist es möglich, dass ein Skript auf einer noch nicht vollständigen Repräsentation des HTML-Dokuments operiert und daraus ggf. Fehler auftreten. Soll das vollständige Laden des Dokuments als Voraussetzung für die Skriptausführung abgewartet werden, muss dafür das boolesche Attribut `defer` auf dem betreffenden `<script>` Tag gesetzt werden.

Zugriffsbeschränkungen

Hinsichtlich des Orts, von dem Skripte durch Angabe eines `src` Attributs geladen werden können, sieht JavaScript keine Einschränkungen vor. Der nicht durch den Nutzer einer Anwendung kontrollierte bzw. kontrollierbare Aufruf server-seitiger Anwendungsfunktionen mittels des unten ausführlicher dargestellten Ausdrucksmittels `XMLHttpRequest` obliegt jedoch aus Sicherheitsgründen der Same Origin Policy, welche den Zugriff auf URLs innerhalb der Ursprungsdomain des im Browser geladenen Dokuments einschränkt.

Logging

Ein praktischer Aspekt von JavaScript besteht in der Verfügbarkeit eines einfachen Mechanismus zur Ausgabe von Logmeldungen, den wir durch das global verfügbare console Objekt und die darauf definierte Funktion `log()` nutzen können. Falls der von Ihnen verwendete Browser über eine *Logging Console* verfügt, wird diese zur Darstellung der Log-Meldungen zur Laufzeit genutzt.

LogLebensdauer

In hohem Maße beachtenswert bei der Verwendung von JavaScript ist die eingeschränkte Lebensdauer aller zur Laufzeit ggf. instantiierten Variablenwerte, inklusive die der Werte globaler Variablen. Diese Lebensdauer ist gebunden an die Dauer der Darstellung des HTML-Dokuments, das zum Zeitpunkt der Wertzuweisung dargestellt wurde und aus dem heraus die Skripte referenziert werden. Unter „Darstellung“ wird hier tatsächlich die Darstellung im *Vordergrund*, d. h. sichtbar in einem Browserfenster verstanden. Wird das HTML-Dokument neu geladen oder durch ein anderes Dokument ersetzt, dann gehen die betreffenden Werte verloren und werden auch bei Rückkehr über die Browser-History nicht wieder hergestellt. Als Entwickler müssen Sie daher ggf. manuelle Vorkehrung für die Zwischenspeicherung von Variablenwerten treffen, die Sie über eine Folge von verschiedenen Dokumenten hinweg verwenden wollen.

Neben den hier in Kürze vorgestellten grundlegenden Ausdrucksmitteln von JavaScript als Programmiersprache können unter dem Begriff „JavaScript“ aber auch im weiteren Sinne die Programmierschnittstellen (APIs) verstanden werden, die uns durch eine client- oder server-seitige Ausführungsumgebung für JavaScript bereitgestellt werden und die sich der genannten Ausdrucksmittel bedienen bzw. zu deren Verwendung wir diese Ausdrucksmittel verwenden müssen. Solche *JavaScript APIs*, die Sie zur Realisierung von Webanwendungen auf mobilen Geräten nutzen können, bilden einen wesentlichen Bestandteil des Lernstoffs, den Sie sich in den kommenden Lerneinheiten aneignen werden. Eingebunden wird deren Nutzung jeweils in die Maßnahmen zur Interaktionssteuerung, die wir verallgemeinert und noch ohne Bezug auf konkrete Anwendungsfunktionalität im folgenden Abschnitt vorstellen werden.

2 Interaktionssteuerung

Wie eingangs erwähnt, werden wir JavaScript verwenden, um die Interaktion eines Nutzers mit einer *graphischen Nutzerschnittstelle* (GUI) zu steuern. Das bedeutet insbesondere, dass JavaScript es uns ermöglichen soll

- auf Nutzereingaben zu *reagieren*,
- die Nutzereingabe und ggf. vom Nutzer eingegebene Daten zu *verarbeiten*,
- nach Abschluss oder während der Verarbeitung dem Nutzer *Feedback* bezüglich seiner Eingabe zu geben.

Feedback in GUIs kann dabei auf unterschiedliche Weise erfolgen, beispielsweise kann dies eine Modifikation der aktuell dargestellten Ansicht umfassen, z. B. durch Farbänderungen, Hervorhebungen oder das Ein- und/oder Ausblenden von Teilen der Ansicht. Möglich sind aber auch Überlagerungen der bestehenden Ansicht, z. B. durch modale Dialoge, deren Entfernung eine Nutzerinteraktion erfordert, oder durch selbstausblendende Überlagerungen wie Wartezeichen oder Fortschrittsbalken. Eine besonders in Android gerne genutzte Variante von Überlagerungen sind die sogenannten „Toasts“, bei denen das Ein- und Ausblenden einer Textmeldung unter Aufrechterhaltung der Bedienbarkeit der überlagerten Ansicht erfolgt. Möglich ist schließlich auch ein Ansichtswechsel durch Übergang in eine Folgeansicht.

Die Umsetzung der oben beschriebenen Verarbeitungsschritte, die von der Detektion einer Nutzereingabe bis zur Entscheidung bezüglich der als Erwiderung darzustellenden Ansicht oder Ansichtsmanipulation reichen, erfolgt üblicherweise im Rahmen einer Softwarearchitektur, die sich an den Ideen des Model View Controller (MVC) Architekturmusters orientiert. Dieses sieht insbesondere vor, dass die Verantwortung bezüglich der einzelnen auszuführenden Schritte an unterschiedliche Komponenten einer Softwareanwendung delegiert wird, welche sich den drei im MVC-Muster angenommenen Schichten von Model, View und Controller zuordnen lassen. Die Annahme dieser Schichten geht darauf zurück, dass für die Umsetzung einer Softwareanwendung mit Nutzerschnittstelle die Differenzierung der folgenden Aspekte nahe liegend erscheint:

- Die **Daten und Operationen**, die eine Nutzerschnittstelle verwendet bzw. auf die sie einem Nutzer den Zugriff ermöglicht. Diese werden im Begriff des *Model* zusammengefasst.
- Die **Ansichten** einer Nutzerschnittstelle, die die Daten des Models zur *Darstellung* bringen und dem Nutzer die Eingabe von Daten und die Ausführung von Operationen auf den dargestellten oder eingegebenen Daten erlauben. Diese werden im Begriff des *View* bezeichnet.
- Die **Vermittlung** zwischen Daten und Operationen, zum einen, und Ansichten, zum anderen. Dies umfasst insbesondere die Verarbeitung von Nutzereingaben unter Aufruf lesender und schreibender Operationen auf dem Datenbestand, den eine Anwendung verwendet, die Auswahl der zu verwendenden Ansichten und ggf. die Übergabe von Daten an die Ansichten. Der Begriff des *Controllers* bezeichnet diese Schicht einer Nutzerschnittstelle.

Auch wenn im folgenden bisweilen von „dem Model“, „der View“ oder „dem Controller“ die Rede sein wird, werden damit nicht notwendigerweise einzelne Komponenten einer Anwendung bezeichnet. Dies bringt auch der von uns verwendete Begriff der „Schicht“ zum Ausdruck. Dieser weist darauf hin, dass es sich bei „Model“, „View“ und „Controller“ eher um Zuständigkeitsbereiche handelt, mit denen die konkreten Komponenten einer Anwendung assoziiert werden können.

2.1 Model View Controller

Unter Bezugnahme auf eine MVC Architektur lassen sich die einzelnen Schritte, die im Zuge des Verarbeitungszyklus einer Nutzereingabe durchgeführt werden müssen, wie in folgender Abbildung dargestellt weiter präzisieren.

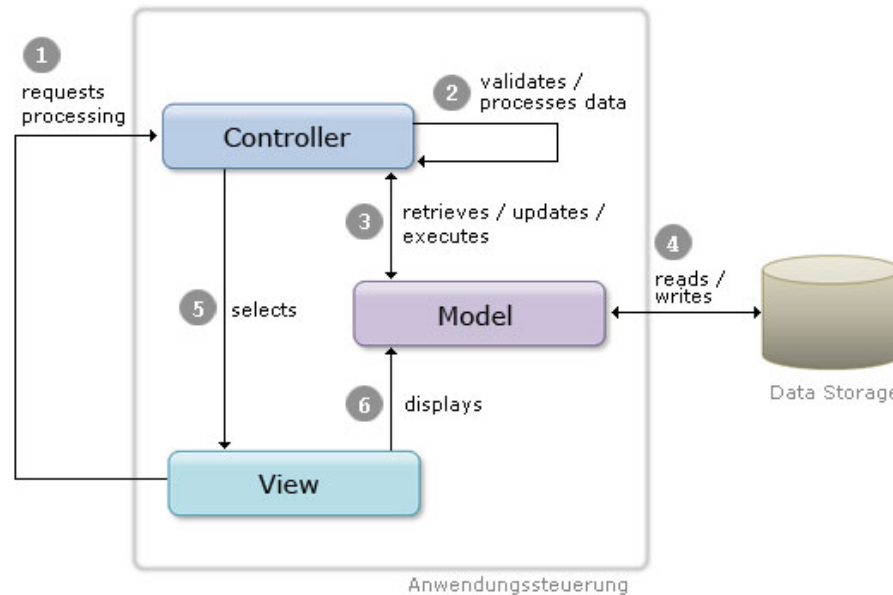


Abb.: Aufgaben von Model, View und Controller im Verarbeitungszyklus einer Nutzereingabe

- Wir gehen hier davon aus, dass die Verarbeitung als Reaktion der View auf eine Nutzereingabe initiiert wird (1).
- Im Zuge der Verarbeitung nimmt der Controller eine Verarbeitung und ggf. eine Validierung der eingegebenen Daten vor (2),
- zu deren Zweck er sich des Models und der darin bereitgestellten Daten und Operationen bedienen kann (3).
- Unsere Darstellung zeigt diesbezüglich auch, dass das Model seinerseits auf Komponenten außerhalb der Nutzerschnittstelle zugreifen kann (4), wobei dieser Zugriff vor den anderen Komponenten der MVC-Architektur verborgen ist. Das Model übernimmt entsprechend dieser Darstellung die Rolle einer Schnittstelle zu externen Anwendungssystemen, deren Funktionalität und Datenbestände durch das Model für die Nutzerschnittstelle zur Verfügung gestellt werden.

Seeheim-Modell

Diese Sichtweise auf die Rolle des Model als „Anwendungsinterface“, das Zugriff auf die Fachlogik-Schicht einer Anwendung bietet, entspricht der als Seeheim-Modell bekannten Auslegung des ursprünglichen MVC Architekturmodells. Die Bezeichnung geht zurück auf einen Workshop, der 1983 in der gleichnamigen Ortschaft am Bodensee stattfand. 📖 [Pfa85].

Im Zuge der Eingabeverarbeitung kann auch ein schreibender Zugriff des Controllers auf das Model erfolgen.

- Als letzter Schritt der Eingabeverarbeitung entscheidet der Controller, welche Ansicht dem Nutzer als Feedback dargestellt werden soll (5). Dies beinhaltet insbesondere die Entscheidung darüber, ob ein Übergang zu einer Folgeansicht vorgenommen werden soll, oder ob die aktuell dargestellte Ansicht modifiziert beibehalten werden soll.
- Ist diese Entscheidung getroffen, baut sich die Ansicht durch Zugriff auf das Model auf oder wird durch den Controller unter Übergabe der zu verwendenden Daten aus dem Model aufgebaut (6).

Anhand der hier identifizierten Schritte des Verarbeitungszyklus lassen sich die Ausdrucksmittel, die unterschiedliche Programmiersprachen bzw. Frameworks für die Entwicklung von Nutzerschnittstellen bereitstellen, sehr gut erschließen. Diese Aussage stützt sich auf die Analyse und Darstellung der Ausdrucksmittel des Play Frameworks, der APIs zur Entwicklung von Apps für Android und iOS sowie von Java Server Faces (JSF) durch den Autor dieser Lerneinheit. Nicht zuletzt gilt sie auch für die Ausdrucksmittel der browserseitig verfügbaren Java Script APIs, die wir nachfolgend betrachten werden. Es erscheint uns auch denkbar, dass verschiedene Spielarten des MVC-Architekturmusters – z. B. die Varianten *Model View Presenter* und *Model View Viewmodel* – sich anhand unterschiedlicher Realisierungen der nachfolgend genannten verallgemeinerbaren Anforderungen charakterisieren lassen. Diese Annahme wurde bisher aber nicht weiter verifiziert und soll daher an dieser Stelle nur als Hypothese „in den Raum gestellt“ werden.

Für die Umsetzung einer MVC-Architektur und die Implementierung des in der vorherigen Abbildung „Aufgaben von Model, View und Controller“ dargestellten Verarbeitungszyklus ist es insbesondere erforderlich, zu klären, wie die folgenden Teilfunktionen in den einzelnen Schritten des Zyklus realisiert werden:

- **Schritt 1:** Wie erfolgt die Anbindung der View an den Controller und die Übergabe der vom Nutzer eingegebenen Daten?
- **Schritt 2-4:** Wie und an welcher Stelle werden die vom Nutzer übergebenen Daten validiert – erfolgt bei Zugriff auf eine externe serverseitige Anwendung die Validierung nur serverseitig, oder sollen Daten auch durch die Nutzerschnittstelle selbst validiert werden?
- **Schritt 2-4:** Wie greift der Controller auf das Model zu, und wie können externe Anwendungen angebunden werden?
- **Schritt 5:** Wie wird die Abfolge von Ansichten bzw. die Modifikation einer bestehenden Ansicht umgesetzt?
- **Schritt 6:** Wie werden im Model enthaltene Daten für die Darstellung im View verfügbar gemacht?

Auf diese Fragen werden wir nachfolgend unter Betrachtung der von uns verwendbaren JavaScript APIs eingehen – bzw. wir werden mit ihrer Klärung beginnen. Andere Aspekte wie z. B. die Ausdrucksmittel, die uns Formulare zur Verfügung stellen, um Daten zu validieren, die Initiierung verschiedener Datenzugriffsoperationen via XMLHttpRequest oder auch die Verwendung von IndexedDB zum Zweck der clientseitigen Datenspeicherung sind vorab in folgender Abbildung zusammengefasst und werden im weiteren Verlauf der Veranstaltung behandelt.

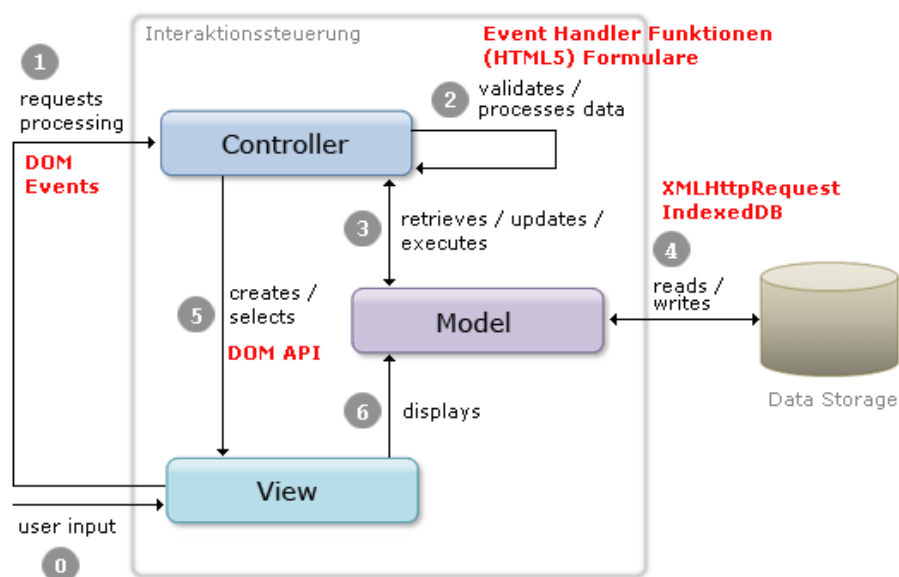



Abb.: Clientseitige JavaScript APIs


Clientseitige JavaScript APIs, die zur Umsetzung des Verarbeitungszyklus von Nutzereingaben zur Verfügung stehen.

In der vorliegenden Lerneinheit werden wir nun zunächst auf die Behandlung von DOM Events und auf die Verwendung der DOM API zur Realisierung von Ansichtsmanipulationen eingehen und damit die Eckpunkte des Verarbeitungszyklus behandeln. Wir werden außerdem zeigen, wie mittels `XMLHttpRequest` lesende Zugriffe auf einen Datenspeicher durchgeführt werden können und wie die ausgelesenen Daten ihrerseits für den Aufbau von Ansichten verwendet werden können.

2.2 Document Object Model

Das  Document Object Model (DOM) ist eine Schnittstellenspezifikation, die eine plattformunabhängige API für den lesenden und schreibenden Zugriff auf XML-Dokumente beschreibt. Dazu gehören u. a. die folgenden Zugriffe.

- Suchen eines Elements / einer Menge von Elementen in einem Dokument
- Lesen / Schreiben der Attribute von Elementen
- Einfügen / Löschen von Elementen in ein / aus einem Dokument


Diese API wird nachfolgend auch mit dem Begriff DOM API bezeichnet. Neben diesen Zugriffsfunktionen beschreibt die DOM-Spezifikation auch, wie Ereignisse bezüglich der Elemente eines XML-Dokuments als  DOM Events zu modellieren sind.

DOM Object

Grundlage für die Verwendung der Zugriffsfunktionen auf Dokumente und die Verarbeitung von Ereignissen bezüglich derer Elemente ist jeweils die Repräsentation des Dokuments als ein Objekt der jeweils verwendeten Programmiersprache bzw. die Bereitstellung eines Objekts, das die DOM API implementiert und deren Funktionen als Zugriffe auf eine geeignete Repräsentation des Dokuments umsetzt. Dies gilt z. B. für Java, wo im Paket `org.w3c.dom` verschiedene Interfacetypen deklariert werden, die der DOM Spezifikation entsprechen. Die DOM Spezifikation wird auch durch die JavaScript Ausführungsumgebung in Browsern genutzt und dient hier dazu, zur Laufzeit den Zugriff auf das von einem Browser geladene und dargestellte HTML-Dokument inklusive der ggf. via JavaScript vorgenommenen Manipulationen zu ermöglichen.

Auch wenn HTML eine Syntax verwendet, die verschiedene in XML nicht vorgesehene Konstrukte erlaubt, werden die von einem Browser zu ladenden HTML-Dokumente in eine Objektrepräsentation überführt, die der eines wohlgeformten XML-Dokuments entspricht. Der Zustand des HTML-Dokuments, das durch einen Browser ggf. unter Anwendung von CSS-Regeln dargestellt wird, kann also zur Laufzeit mittels der DOM API ausgelesen und manipuliert werden.

Zugriff auf das DOM Objekt

Wenn hier oder an anderer Stelle Begriffe wie „das DOM-Objekt“ oder „der DOM“ verwendet werden, dann ist damit die Objektrepräsentation dieses Dokuments gemeint. Zugreifbar ist diese aus JavaScript heraus durch Verwendung der globalen Variable `document`, welche das DOM-Objekt als Wert des gleichnamigen Attributs des `window` Objekts referenziert – dieses  window Objekts können Sie sich als Repräsentation eines Browserfensters bzw. eines Browsertabs vorstellen, in welchem das genannte Dokument zur Darstellung gebracht wird.

Nachfolgend werden wir mit DOM Events sowie anhand der DOM API verschiedene Verwendungsweisen der DOM-Spezifikation im Rahmen des Verarbeitungszyklus für Nutzereingaben illustrieren.

2.3 Eingabeverarbeitung

Typen von Events

Die bereits oben erwähnte [W3C DOM Events Spezifikation des W3C](#) beschreibt verschiedene Typen von Ereignissen, die bezüglich eines HTML-Dokuments auftreten können. Dazu gehören z. B. die sogenannten *User Interface Events*, die sich allerdings – anders, als der Name es suggeriert – nur auf die „Eckdaten“ der Darstellung eines HTML-Dokuments beziehen. Beispielsweise signalisiert `load` den Abschluss des Ladens des Dokuments und aller daraus referenzierten Dokumente, Bilder, CSS-Regeln und JavaScript Dateien, während `resize` nicht wirklich überraschend eine Größenänderung des Browserfensters anzeigt.

Die eigentlichen „Bedienereignisse“, die bezüglich der einzelnen Elemente eines als DOM-Objekt repräsentierten HTML-Dokuments auftreten können, werden dann als davon getrennte Ereignistypen spezifiziert, z. B. beziehen sich die *Focus Events* `focus` und `blur` auf das Fokussiertwerden bzw. den Verlusts des Fokus aus der Sicht eines Elements des DOM. Darüber hinaus werden *Mouse Events* wie `click`, *Keyboard Events* wie `keydown` und `keypress` sowie *Wheel Events* spezifiziert, mittels derer die Bedienung des Mausekkrads verfolgt werden kann, falls erwünscht.

Strukturelle Änderungen des DOM-Objekts werden schließlich durch *Mutation Events* angezeigt, und für alternative Eingabeverfahren wie Handschrift kann der Ereignistyp von *Composition Events* verwendet werden. Wie Sie weiter unten sehen werden, existieren außerdem für spezifische Elemente, die zur aktuellen Fassung von HTML gehören, elementenspezifische DOM-Ereignisse, die genau wie die hier vorgestellten Typen von Ereignissen behandelt werden können.

Touch Events

Um der spezifischen Eingabemodalität aktueller mobiler Endgeräte mit berührungsempfindlichem Display Rechnung zu tragen, führt [W3C](#) eine aktuelle *W3C Spezifikation* außerdem den neuen Ereignistyp *Touch Events* ein. Dieser sieht zum einen die vier elementaren Touch Ereignisse `touchstart`, `touchend`, `touchmove` und `touchcancel` vor und erlaubt es zum anderen, diese Ereignisse zur Behandlung von *Multi-Touch-Eingaben* zu gruppieren, bei denen mehr als ein Berührungsbereich vorliegt. Um aber die Entwicklung mobiler Webanwendungen im Rahmen unserer Veranstaltung auch auf Grundlage eines Desktop- oder Laptop-Browsers zu ermöglichen, verwenden wir in den Implementierungsbeispielen keine Touch Events, sondern verlassen uns darauf, dass die Berührung eines Bedienelements zum Zweck der Ausführung einer Aktion auch in mobilen Browsern durch das DOM-Ereignis `click` signalisiert wird. Für die Verwendung von Touch Events in Firefox sei [W3C](#) auf die *Dokumentation* verwiesen.

Event Handler

DOM-Events dienen dazu, eine feingranulare Reaktion auf den Vollzug einer Nutzereingabe zu geben, falls erforderlich. Dies schließt die Möglichkeit ein, bereits während des Vollzugs Feedback durch eine Modifikation der bestehenden Ansicht zu realisieren, beispielsweise durch Änderung von Hintergrundfarbe oder Durchsichtigkeit von Schaltflächen. Insbesondere für letztere Ereignisse kann auch mittels *Pseudoselektoren* wie `:active` eine alternative Stilzuweisung erfolgen, ohne dass dafür die Implementierung von JavaScript-Code erforderlich wäre.

Zur Ereignisbehandlung in JavaScript werden JavaScript-Funktionen verwendet, denen bei Auftreten eines Ereignisses eine Objektrepräsentation dieses Ereignisses übergeben wird. Diese enthält in einem `target` Attribut eine Repräsentation des DOM-Elements, bezüglich dessen das Ereignis aufgetreten ist, sowie ein `type` Attribut, das einen String-wertigen Bezeichner für den Ereignistyp als Wert enthält.



Quellcode

Event Handler Funktion

```
001 // Event Handler Funktion
002 function handleEvent(event) {
003     console.log("got event of type " + event.type + " on element " +
004         event.target + " with id: " + event.target.id);
005     // behandle das Ereignis
006     /* (...) */
007 }
```

Setzen von Event Handlern

Die Übergabe des `event` Objekts ermöglicht es zum einen, einen einzigen Event Handler für die Behandlung von Ereignissen bezüglich mehrerer DOM-Elemente zu deklarieren, welche ggf. anhand ihres Typs oder ihrer `id` voneinander unterschieden werden können. Zum anderen erleichtert sie die Ereignisbehandlung insofern, als das betreffende Element nicht mehr „manuell“ aus dem HTML-Dokument ausgelesen werden muss, sondern *automatisch* bereit steht. Eine vollständige [www](#) Referenz des `Event` Objekts finden Sie in der Mozilla Dokumentation.

Für die Assoziation eines Event Handlers mit einem DOM-Element können zwei grundsätzlich verschiedene Ansätze sowie eine Alternative bezüglich eines dieser Ansätze angewendet werden. So ist es zum einen möglich, Event Handler direkt auf Ebene des HTML-Markup zu deklarieren:

```
<button id="eventbutton" onclick=handleEvent(event)"/>
```

Assoziation mit Funktionen und Methoden

Stilistisch gilt diese Lösung im allgemeinen als grenzwertig, da hier eine Abhängigkeit zu einer konkreten JavaScript-Funktion auf Ebene des HTML-Markups eingeführt wird. Damit wird die Bedienbarkeit des Buttons an die Ausführbarkeit von JavaScript auf dem betreffenden Endgerät gebunden, was wir jedoch in unserer Veranstaltung als gegeben erachten. Wir möchten außerdem darauf hinweisen, dass sowohl iOS als auch ansatzweise Android eine solche Möglichkeit vorsehen, auf Ebene eines View – wie sie durch unser HTML-Dokument gegeben ist – eine Assoziation zu ereignisbehandelnden Funktionen bzw. Methoden deklarieren. Die betreffenden Funktionen müssen dort dann durch den verwendeten `ViewController` bzw. die steuernde `Activity` bereit gestellt. Vor diesem Hintergrund erscheint uns durchaus ein Ermessensspielraum bezüglich der expliziten Assoziation von DOM-Elementen und Event Handlern zu existieren.

Eine Alternative zur Deklaration von Event Handlern in HTML-Dokumenten ist die Zuweisung des Handlers zu den betreffenden Elementen in JavaScript. Hierfür wird üblicherweise unter Verwendung der (nachfolgend beschriebenen) DOM API ein Element aus dem DOM ausgelesen und auf diesem Element die von uns gewünschten Event Handler gesetzt. Dafür kann die Event Handler Funktion einem entsprechend dem Ereignistyp benannten Attribut des Elements als Wert übergeben werden.

```
001 // lies das Element aus dem DOM aus
002 var showtoast = document.getElementById("eventbutton");
003 // setze einen Event Handler auf dem Element
004 showtoast.onclick = handleEvent;
```

Wiederum eine Alternative hierzu ist die Verwendung der Funktion `addEventListener()`.

Dieser wird zum einen der Ereignistyp zum anderen der Event Handler übergeben. Wir zeigen nachfolgend einen Fall, in dem letzteres nicht durch Referenzierung eines existierenden Funktionsobjekts – `handleEvent` aus den vorgenannten Beispielen – erfolgt, sondern durch Deklaration einer anonymen Funktion beim Aufruf von `addEventListener()`:

```
001 // Setzen eines anonymen Event Handlers
002 showtoast.addEventListener("click", function(event) {
003     // behandle das Ereignis
004     /* (...) */
005 });
```

Ein wesentlicher Unterschied zwischen den beiden Deklarationen ist der, dass etwaige Event Handler, die für das `click` Ereignis bereits existieren, im ersten Fall überschrieben werden. Wird jedoch `addEventListener()` genutzt, wird der angegebene Handler etwaigen anderen zum gegebenen Zeitpunkt existierenden Handlern hinzugefügt. Mag dies auch manchmal erwünscht sein, so scheint die Situation zu überwiegen, dass diese Hinzufügung versehentlich erfolgt und daher zu Irritationen führt, falls der Handler tatsächlich zweimal ausgeführt wird.

Sie sollten also ggf. dafür Sorge tragen, dass `addEventListener` genau einmal ausgeführt wird. Soll andererseits ein bereits existierender Handler entfernt werden kann dies durch Aufruf von `removeEventListener()` erfolgen. Dieser Funktion muss dann aber das Funktionsobjekt des Event Listener übergeben werden, d. h. in diesem Fall darf der Listener nicht als anonyme Funktion deklariert werden.

Phasen der Ereignisverarbeitung

Ein wichtiger Aspekt von DOM Events ist die Umsetzung eines durch die Spezifikation beschriebenen Verhaltens, welches als „Event Bubbling“ bezeichnet wird. Dies trägt dem Umstand Rechnung, dass die Elemente einer Ansicht entsprechend der Repräsentation der Ansicht als Baumstruktur ineinander eingebettet sind. Damit kann z. B. ein `click` Ereignis, das bezüglich eines `<button>` Elements auftritt, immer auch als Ereignis bezüglich des Elternelements von `button`, des Großelternelements, etc. angesehen und vor allem behandelt werden. Im Hinblick auf eine generalisierte Behandlung von Ereignissen werden dafür drei folgenden Phasen unterschieden.

1. **Capture Phase:** das Ereignis „wandert“ von der Wurzel des Dokuments zum angezielten Element und kann hier von dessen „Vorfahren“-Elementen bearbeitet werden.
2. **Target Phase:** das Ereignis wird durch das angezielte Element selbst bearbeitet.
3. **Bubbling Phase:** das Ereignis „wandert“ vom angezielten Element zurück zur Wurzel und kann hier wiederum von den Vorfahren-Elementen bearbeitet werden.

Im Rahmen eines DOM braucht die Behandlung von Ereignissen also nicht erst bei dem Element zu beginnen, für das ein Ereignis eigentlich aufgetreten ist. Vielmehr „sinken“ Ereignisse zu diesem Element ausgehend von der Wurzel des Dokuments „hinab“, bevor sie dann wieder von dort aus zur Wurzel „hochblubbern“, und in allen Phasen kann eine Ereignisbehandlung erfolgen. Die folgende Abbildung illustriert diesen Vorgang.

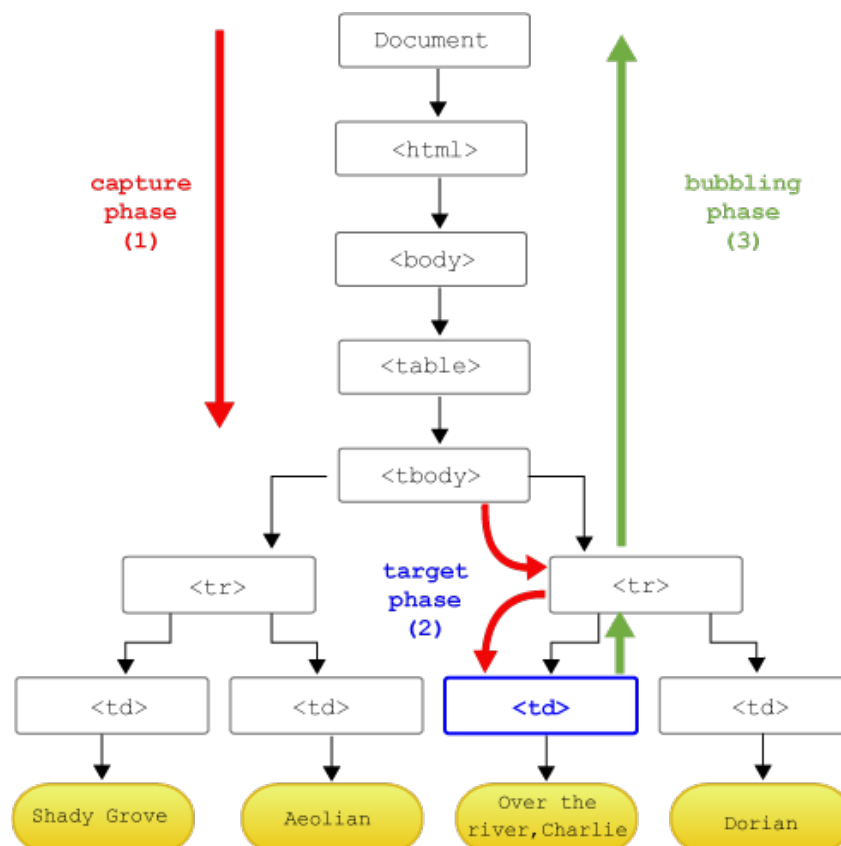


Abb.: Phasen der Ereignisbehandlung

Für die Ereignisbehandlung ist diesbezüglich zweierlei von Bedeutung. Zum einen kann durch einen zusätzlichen Parameter bei Aufrufen von `addEventListener()` und `removeEventListener()` angezeigt werden, für welche Phase ein Event Handler hinzugefügt oder entfernt werden soll. Soll die Capture Phase behandelt werden, wird der Wert `true` übergeben.

Ohne Angabe des Parameters wird der Handler für die Target Phase bzw. die *Bubbling Phase* deklariert. Auf der Objektrepräsentation des Ereignisses im event Objekt kann die aktuelle Phase, für die der Handler aufgerufen wird, mittels eines Attributs `phase` ermittelt werden. Das `<target>` Attribut des Objekts verweist dabei immer auf das Element, bezüglich dessen das Ereignis ursprünglich detektiert wurde. Ein anderes Attribut, `currentTarget`, zeigt hingegen an, für welches Element im DOM-Baum der Event Handler zum gegebenen Zeitpunkt aufgerufen wird. Soll schließlich der in der vorherigen Abbildung „Phasen der Ereignisbehandlung“ gezeigte Verarbeitungsprozess abgebrochen werden, empfiehlt sich der Aufruf der Funktion `stopPropagation()` auf dem event Objekt.

2.4 Visuelles Feedback

Wir werden nun verschiedene Möglichkeiten vorstellen, für die eine Anwendung Event Handler nutzen kann, um dem Nutzer im Verlauf der Verarbeitung einer Nutzereingabe Feedback zu geben. Hinsichtlich der Inhalte einer solchen Erwiderung lassen sich die folgenden Dimensionen unterscheiden:

1. Ist die Eingabe *als Eingabe* durch die Anwendung *zur Kenntnis* genommen worden, bzw. wird *ihr Vollzug* zur Kenntnis genommen?
2. Findet nach erfolgter Eingabe eine Verarbeitung *statt oder nicht*?
3. *Welche Inhalte* werden durch die Anwendung für die Verarbeitung verwendet?
4. *Können* die übermittelten Inhalte für die Verarbeitung *verwendet werden*?
5. *Was ist das Resultat der Verarbeitung* der Nutzereingabe und der bereitgestellten Inhalte?

Durch Rückmeldungen bezüglich dieser Aspekte stellt eine Anwendung Transparenz bezüglich ihres Verhaltens her und ermöglicht damit auf Nutzerseite Verständnis auch in Situationen, in denen die Anwendung nicht das durch den Nutzer eigentlich erwartete Verhalten zeigt. Zu Frage 3 sei angemerkt, dass es in Anwendungen mit graphischer Nutzerschnittstelle für den Nutzer meistens offensichtlich ist, welche Werte für die Verarbeitung verwendet werden, da die Eingabeelemente einer Anwendung normalerweise die durch den Nutzer eingegebenen Inhalten anzeigen, seien es Checkboxes, Auswahl Elemente oder Freitexteingabefelder.

Gerade für mobile Anwendungen ist die Eingabe von Freitext aber durch die dafür zumeist verwendete virtuelle Tastatur einer höheren Fehleranfälligkeit ausgesetzt, weshalb die Eingabe üblicherweise durch Funktionen wie Autovervollständigung oder Autokorrektur unterstützt wird. Wie Sie bestimmt aus eigener Erfahrung wissen, können gerade diese Funktionen aber auch zu Verfälschungen der eigentlichen Eingabe führen, und trotz des vorhandenen visuellen Feedbacks haben Sie mangels Aufmerksamkeit bestimmt auch schon Nachrichten versendet, deren Sinnhaftigkeit durch die genannten Eingabehilfen zumindest verschleiert wurde – wir verweisen diesbezüglich auf einen unterhaltsamen Artikel von Elena Senft.

 [Tücken und Tricks bei Mails und SMS - von Elena Senft.](#)

Die in Punkt 1 genannte Zurkenntnisnahme einer Eingabe als Eingabe kann insbesondere durch die bereits erwähnte Möglichkeit einer alternativen Stilzuweisung für `:active` Eingabeelemente gewährleistet werden, und das Feedback bezüglich der Verwendbarkeit der eingegebenen Werte als Bestandteil der im Verarbeitungszyklus oben genannten Eingabevalidierung wird teilweise durch aktuell verfügbare Ausdrucksmittel für HTML-Formulare ermöglicht, mit denen wir uns in der Lerneinheit „CDF - CRUD Datenzugriff mit Formularen“ beschäftigen.

Abgesehen davon erfordern die genannten Feedbackdimensionen jeweils die Manipulation, Überlagerung oder den Austausch der aktuellen Ansicht aus der Eingabeverarbeitung heraus, d. h. durch die Implementierung von Event Handlern. Nachfolgend werden wir verschiedene Alternativen vorstellen, die uns für die Durchführung dieser Maßnahmen zur Verfügung stehen. Besonderes Augenmerk richten wir dabei auf die Manipulation des DOM Objekts mittels der DOM API.

2.4.1 Standard-Dialoge und Neuladen von Dokumenten

Modale Dialoge

Zur Überlagerung der durch einen Browser dargestellten Ansicht stehen zwei standardisierte Bedienelemente zur Verfügung, die jeweils als modaler Dialog realisiert werden, d. h. als eine blockierende Überlagerung, deren Aufhebung eine Nutzereingabe auf dem betreffenden Element erfordert. Ausgelöst wird ihre Darstellung durch die JavaScript Funktionen `alert()` bzw. `confirm()`, denen jeweils der anzuzeigende Text übergeben werden kann. Die Funktionen unterscheiden sich darin, dass `alert()` nur die „aktive Zurkenntnisnahme des Nutzers“ erfordert, während `confirm()` dem Nutzer zwei Handlungsalternativen anbietet. Dem entsprechend verfügt der durch `alert()` ausgelöste Dialog nur über ein einziges Bedienelement, während `confirm()` wahlweise eine „Ja“ oder eine „Nein“ Antwort ermöglicht. Die Darstellung der beiden Dialoge ist abhängig vom verwendeten Browser sowie dem Endgerät, auf dem dieser ausgeführt wird. Firefox auf Android bedient sich beispielsweise der nativen User Interface Komponenten, die Android für Dialoge bereitstellt:

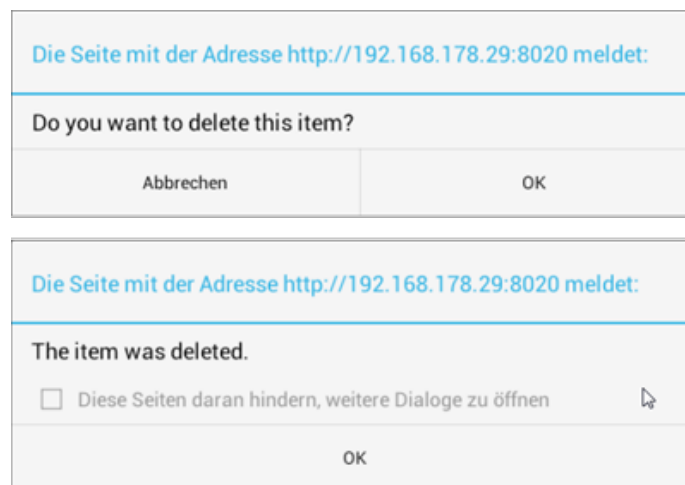


Abb.: Android Dialoge

Wie Sie sehen können, handelt es sich hier um den typischen Fall einer Löschaktion, deren Durchführung eine explizite Rückbestätigung sinnvoll erscheinen lässt. Dafür verwenden wir die Funktion `confirm()`, die für den Fall der Auswahl von „Ja“/„OK“ den Wert `true` zurückgibt und damit als Bedingung einer `if` Anweisung verwendet werden kann. Das Feedback bezüglich der erfolgreichen Aktionsausführung realisieren wir hier ebenfalls mittels eines modalen Dialogs, ausgelöst durch `alert()`. Die Motivation hierfür liegt aber eher darin, dass wir beide Funktionen anhand eines einzigen Beispiels illustrieren möchten... so wäre für die Rückbestätigung einer erfolgreichen Aktionsausführung die Verwendung eines „Toasts“ als selbstabblender nicht blockierender Überlagerung oder auch nur die Anzeige des Effekts der Aktion sinnvoller. Falls z. B. ein Element aus einer Liste entfernt wird, reicht dafür ggf. die Anzeige der aktualisierten, d. h. um das betreffende Element reduzierten, Liste aus.

Das nachfolgende Codebeispiel setzt die hier angenommene Löschaktion als Event Handler um, der einem Bedienelement für das `click` Ereignis übergeben wird:



Quellcode

Codebeispiel Event Handler

```
001 // deklariere eine Event Handler Funktion, die eine Löschaktion
002 //rückbestätigt und ausführt
003 function deleteObject(event) {
004     if (confirm("Do you want to delete this item?")) {
005         // führe die Löschaktion aus
006         /* (...) */
007         // gib nach erfolgreicher Ausführung Feedback
008         alert("The item was deleted.");
009     }
010 }
011
012 // setze die Funktion als Event Handler auf einem Bedienelement
013 deleteButton.onclick = deleteObject;
```

Alternativ zur Verwendung von `alert()` und `confirm()` lassen sich modale Dialoge auch auf Grundlage einer DOM-Manipulation realisieren, wie wir sie weiter unten beschreiben werden. Auf mobilen Geräten werde diese Dialoge dann zwar nicht, wie oben gezeigt, mittels nativer Bedienelemente realisiert, dafür bestehen bezüglich der Gestaltung der Dialoge und der darin angebotenen Handlungsalternativen aber auch keine Einschränkungen.

Einen Ansichtswechsel durch Neuladen eines HTML-Dokuments können Sie aus JavaScript heraus sehr einfach initiieren, indem Sie den Wert des Attributs `window.location` auf die URL des zu ladenden Dokuments setzen – wir nehmen im folgenden Beispiel an, dass diese als Wert der Variable `nextViewUrl` gegeben ist.

```
// lade ein neues Dokument  
window.location = nextViewUrl;
```

Solange die Sicherheitsanforderungen der Same Origin Policy gewahrt sind, d. h. die angegebene URL die selbe Ursprungsdomain verwendet wie das geladene Dokument, kann hier als Wert von `location` jegliche gültige URL verwendet werden. Dies schließt die Möglichkeit ein, mittels eines „#“ Suffixes auf ein Fragment, d. h. einen anderen Seitenbereich, des aktuellen Dokuments zu verweisen. Wird jedoch tatsächlich ein anderes Dokument geladen, so ist zu beachten, dass dies zu einer Neuinstantiierung des Ausführungskontexts für JavaScript führt, d. h. etwaige zuvor gesetzte Variablenwerte sind nach Laden des referenzieren Dokuments nicht mehr verfügbar.

2.4.2 DOM Zugriff und Manipulation

Für den lesenden Zugriff auf das DOM-Objekt stehen uns in JavaScript verschiedene Funktionen zur Verfügung. Diese tragen den verschiedenen Identifizierungsmöglichkeiten für Elemente eines HTML-Dokuments Rechnung, so können Elemente z. B. mittels der folgenden Funktionen anhand ihres Elementnamens, eines möglicherweise gesetzten `id` Attributs oder eines Werts des `class` Attributs identifiziert werden:

- `getElementsByTagName (name)`
- `getElementById (id)`
- `getElementsByClass (klass)`

Die beiden Funktionen, die die Pluralform „Elements“ im Namen tragen, liefern hier einen Array von Elementen zurück, über den Sie mit den in JavaScript verfügbaren Mitteln iterieren können. Die Funktion `getElementById()` gibt Ihnen hingegen das betreffende Element, falls es im DOM gefunden werden kann, und andernfalls `null`. Diese Funktion ist nur auf `document` selbst definiert, während die anderen beiden Funktionen auch [www](#) auf Elementen selbst aufgerufen werden können, um ggf. darin eingebettete Elemente auszulesen. Letztere Zugriffsalternativen via `document` oder über ein Element eines Dokuments bestehen auch für die beiden folgenden Funktionen, die es Ihnen ermöglichen, Elemente anhand eines ggf. zusammengesetzten CSS-Selektors zu identifizieren:

- `querySelector (selector)`
- `querySelectorAll (selector)`

Unterschiede bestehen hier darin, dass `querySelector()` im Gegensatz zu `querySelectorAll` als Rückgabewert nur das erste Element im DOM liefert, auf das der Selektor zutrifft. Durch diese beiden Methoden wird so ein zentrales Ausdrucksmittel von CSS auch für die Ebene von JavaScript nutzbar gemacht. Dies sollte nicht als eine Verwirrung der Zuständigkeiten beider Sprachen gesehen werden, da die Identifikation von Elementen via Selektoren in CSS sich ja auf den Zustand des DOM-Objekts bezieht, der ggf. durch DOM-Manipulationen in JavaScript herbeigeführt wurde. Da erscheint es nur folgerichtig, dass sich auch JavaScript der Möglichkeit bedienen kann, Elemente mittels Selektoren zu identifizieren.

Mittels der genannten Methoden kann also ein gezielter Zugriff auf Elemente im DOM-Objekt erfolgen. Auf solchen Elementen können wir dann, wie wir es in den obigen Beispielen bereits getan haben, einerseits Event Handler setzen, um Nutzereingaben zu verarbeiten. Die ausgelesenen Elemente können aber auch hinsichtlich aller Attribute bzw. durch Hinzufügung von Tochterelementen modifiziert werden. Für jede Modifikation des DOM-Objektes aktualisiert der Browser dessen Darstellung inklusive der Anwendung etwaiger CSS-Regeln, d. h. DOM-Manipulationen resultieren unmittelbar in einer Modifikation der dargestellten Ansicht.

Zur Durchführung von DOM-Manipulationen stehen uns nun verschiedene Möglichkeiten zur Verfügung. Diese illustrieren wir nachfolgend am Beispiel einer Listenansicht, wie sie von zahlreichen mobilen Anwendungen verwendet wird. Ein Beispiel dafür sehen Sie in der folgenden Abbildung. Grundlage dafür ist ein `` Element, dem eine Menge von `` Elementen als Tochterelementen zugewiesen werden, für welche wir die folgende Struktur nutzen – es handelt sich hier um eine Vereinfachung der Struktur, die in den späteren Implementierungsbeispielen verwendet wird und die der in der folgenden Abbildung (rechts) gezeigten Ansicht zugrunde liegt.

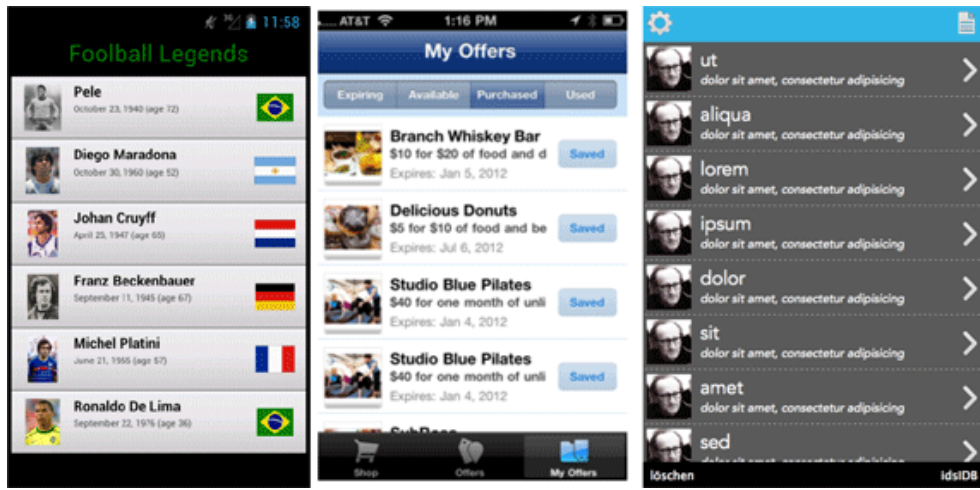


Abb.: Beispiele für die Verwendung von Listenansichten in mobilen Anwendungen (links, mitte) und Nachbau einer Listenansicht mit HTML und CSS.



Quellcode

Listenansicht

```
001 <li class="listitem-img-h2-h3">
002   
003   <div class="titleblock">
004     <h2>Lorem Ipsum</h2><h3>dolor sit amet, consectetur adipiscing</h3>
005   </div>
006 </li>
```

innerHTML

Die augenscheinlich einfachste Lösung zur DOM-Manipulation besteht darin, das Attribut `innerHTML` zu setzen, das auf jedem DOM-Element existiert. Als dessen Wert kann ein beliebiger String mit (hoffentlich) wohlgeformtem HTML-Markup angegeben werden. So könnte z. B. das oben gezeigte Element auf die folgende Weise einem `` Element hinzugefügt werden. Wir nehmen dabei an, dass die durch `<h2>` und `<h3>` markierten Titel und Untertitel des Elements als Werte der Variablen `title` und `subtitle` vorliegen:



Quellcode

innerHTML

```
001 // lies das Wurzelement der Liste aus
002 var listroot = document.getElementsByTagName("ul")[0];
003
004 /* baue ein Listenelement unter Verwendung der Werte von
005    title und subtitle auf */
006 var listelement = "<li class='listitem-img-h2-h3'><img src='img/hm2.jpg'>
007   <div class='titleblock'><h2>" + title + "</h2><h3>" + subtitle
008   + "</h3></div></li>";
009 // setze das Listenelement als Inhalt der Liste
010 listroot.innerHTML = listelement;
```

Problematik der Nutzung

Dieses Beispiel zeigt sehr deutlich bereits die Problematik, die die Nutzung von `innerHTML` mit sich bringt, wenn damit nicht nur String-wertige Inhalte, sondern Elemente mit ggf. eingebetteten Elementen als Tochterelemente gesetzt werden sollen. So ist nicht nur die Lesbarkeit des im Quellcodes gesetzten Markups beeinträchtigt.

Wie in allen Fällen, in denen Markup durch manuelle String-Konkatenierung erzeugt wird, ist es hier zudem nicht möglich, die Wohlgeformtheit des Markups zu verifizieren. Vollends ungeeignet erscheint das Verfahren aber im vorliegenden Fall aus einem anderen Grund. Die oben gezeigte Anweisung fügt nämlich nicht etwa ein neues Element zu einer bestehenden Listenansicht hinzu, sondern ersetzt den gesamten Inhalt des `` Elements. Zwar könnte grundsätzlich auch die folgende Anweisung durchgeführt werden, bei der das neue Element zum bestehenden Inhalt hinzugefügt wird:

```
listroot.innerHTML = listroot.innerHTML + listelement;
```

In diesem Fall müsste der Browser jedoch von neuem den gesamten textuellen Inhalt, der als Wert von `innerHTML` gesetzt werden soll, parsen und damit die Listenelemente innerhalb des DOM-Objekts aufbauen. Angesichts der Tatsache, dass es alternative Verfahren gibt, bei denen tatsächlich nur eine Hinzufügung des neuen Elements vorgenommen wird, erscheint diese Lösung mit Blick auf die dafür erforderliche Rechenleistung jedoch nicht vertretbar. Sinnvoll ist die Verwendung von `innerHTML` im Zusammenhang mit Listenansichten aber durchaus für den Fall, dass eine bestehende Liste komplett neu aufgebaut werden soll. So können die bestehenden Inhalte wie folgt entfernt werden.

```
// lösche alle Listeninhalte
listroot.innerHTML = "";
```

Aufbau von Elementen

Das Gegenstück zur Verwendung von `innerHTML` ist die Verwendung von spezifischen Funktionen der DOM API, die es uns u. a. ermöglichen neue Elemente zu erstellen, Attribute auf diesen Elementen zu setzen sowie Elementen Kind-Elemente gezielt hinzuzufügen. Diese wenden wir im folgenden Codebeispiel an.

Erklärungsbedürftig ist hier insbesondere der Einsatz der beiden Funktionen `createElement()` und `createTextNode()` auf dem DOM-Objekt. Bei diesen handelt es sich um „Factory-Methoden“, die uns die durch die gegebene DOM-Implementierung erforderliche Objektrepräsentation von Elementen, Textinhalten etc. liefern. Diese Objekte können dann mittels `appendChild()` als Tochterelemente zu einem Element hinzugefügt werden. Für die Setzung von Attributen bestehen verschiedene Möglichkeiten. So kann diese grundsätzlich als JavaScript Wertzuweisung an Objektattribute erfolgen, wie unten bezüglich `img.src`. Die DOM API stellt uns aber für Elemente eines Dokuments auch die Methode `setAttribute()` zur Verfügung, welcher wir Name und Wert des betreffenden Attributs übergeben können.

Für die Handhabung des `class` Attributs existiert außerdem eine [www](#) spezielle API, die es uns erlaubt, dessen Attributwert nicht als String, sondern als eine Liste von Strings zu behandeln. Dementsprechend können wir auch mittels `add()` einen neuen Wert für `class` setzen, ohne uns darum kümmern zu müssen, welche anderen Werte möglicherweise bereits zugewiesen worden sind:



Quellcode

Neue Elemente erstellen

```
001 // erzeuge ein neues li Element
002 var li = document.createElement("li");
003 // setze das erforderliche Attribut
004 li.classList.add("listitem-img-h2-h3");
005 // erzeuge das img Element
006 var img = document.createElement("img");
007 // setze das src Attribut
008 img.src = "img/hm2.jpg";
009 // erzeuge das div Element
010 var div = document.createElement("div");
011 // setze das class Attribut (alternativer Zugriff)
012 div.setAttribute("class", "titleblock");
013 // erzeuge das h2 Element
014 var h2 = document.createElement("h2");
015 // füge dem Element den Text aus title hinzu
016 h2.appendChild(document.createTextNode(title));
017 // erzeuge das h3 Element und füge den Text aus subtitle hinzu
018 var h3 = document.createElement("h3");
019 h3.appendChild(document.createTextNode(subtitle));
020
021 // baue die Struktur des li Elements auf
022 li.appendChild(img);
023 li.appendChild(div);
024 div.appendChild(h2);
025 div.appendChild(h3);
026
027 // füge das li Element der Liste hinzu
028 listroot.appendChild(li);
```

Vermutlich wird Ihnen dieser Beispielcode gemessen an seinem Effekt – der Hinzufügung eines einzigen Elements zu einer bestehenden Liste – nicht nur überdimensioniert erscheinen sondern ähnlich intransparent hinsichtlich der tatsächlich erzeugten Struktur wie der oben konkatenierte (und nicht zur Lesbarkeit mit Zeilenumbrüchen formatierte) String-Wert für `innerHTML`. Ein essentieller Unterschied besteht jedoch zwischen beiden Verfahren insofern, als die Anwendung der DOM API die Wohlgeformtheit des Dokuments *garantiert*, das über eine Folge von Funktionsaufrufen aufgebaut wurde.

Templates

Als weitere Alternative zur Umsetzung einer DOM Manipulation wollen wir eine Lösung vorschlagen, die in besonderer Weise für die *wiederholte* Hinzufügung von Elementen gleicher Struktur und ggf. unterschiedlichen Inhalts geeignet ist, mit der wir es im Fall unserer Liste zu tun haben. Auch hier werden wir durch die DOM API durch spezifische Funktionen unterstützt, die wir bisher noch nicht verwendet haben. Unsere Lösung nimmt an, dass Sie in Ihrem HTML Dokument eine dynamisch aufzubauende Liste z. B. wie folgt mit einem einzigen Listenelements als Inhalt darstellen. Um das Verfahren explizit zu machen, markieren wir dieses Element mit einem zusätzlichen Wert für `class` als `listitem-template`:



Quellcode

Liste

```
001 <ul>
002   <li class="listitem-img-h2-h3 listitem-template">
003     <!-- interner Aufbau wie oben -->
004   </li>
005 </ul>
```

Dieses `` Element werden die Nutzer unserer Anwendung aber nicht zu Gesicht bekommen, da wir es beim Laden des Dokuments z. B. in einem `onload` Event Handler bezüglich `<body>` aus dem DOM-Objekt mittels der Funktion `removeChild()` „ausschneiden“ und als Wert einer Variable „beiseite legen“:



Quellcode

Liste als Variable

```
001 // lies die Liste aus
002 var listroot = document.getElementsByTagName("ul")[0];
003
004 // lies das Element aus und entferne es aus der Liste
005 var listitemTemplate = document.getElementsByClassName("listitem-template")[0];
006 listroot.removeChild(listitemTemplate);
```

Auf den Wert der Variable `listitemTemplate` greifen wir zu, wenn wir dann z. B. als Reaktion auf eine Nutzerinteraktion oder im Zuge der initialen Darstellung der Liste ein neues Listenelement hinzufügen wollen. Zur Erstellung dieses neuen Elements rufen wir auf dem Objekt `listitemTemplate` eine Funktion namens `cloneNode()` auf, die uns bei Übergabe des Arguments `true` eine echte Kopie von `listitemTemplate` liefert:

```
// erzeuge eine echte Kopie des Template Elements
var newListitem = listitemTemplate.cloneNode(true);
```

Auf dieser Kopie können wir dann die von uns für `title` und `subtitle` gewünschten Werte setzen und bedienen uns dafür z. B. der oben genannten Zugriffsfunktion `getElementsByTagName()`, die uns auf jedem DOM Element zur Verfügung steht sowie des Attributs `textContent`, das als Alternative zu `innerHTML` für die Setzung von nicht mit Markup versehenen textuellen Inhalten eines Elements verwendet werden sollte:

```
001 // übertrage die dynamischen Inhalte auf das neue Element
002 newListitem.getElementsByTagName("h2").textContent = title;
003 newListitem.getElementsByTagName("h3").textContent = subtitle;
```

Das neue Element fügen wir schließlich zur bestehenden Liste hinzu, ohne dass wir uns darum zu kümmern brauchen, wie viele Elemente bereits in der Liste existieren:

```
listroot.appendChild(newListitem);
```


Vorteilhaft bei diesem Verfahren ist nicht nur der überschaubare Codeumfang sowie die Wohlgeformtheitsgarantie, die durch die Verwendung der DOM API gegeben ist. Insbesondere ermöglicht das Verfahren auch die Konzeption von Dokumentenstruktur und CSS-Regeln für die dynamischen Elemente einer Anwendung, ohne dass dafür die Implementierung von JavaScript erforderlich wäre – denn das Template ist ja zunächst Bestandteil eines statischen HTML-Dokuments.

<template>

Als fragwürdig könnte unser Verfahren aber angesehen werden, weil wir durch Verwendung von Ansichtselementen, die wir evtl. gar nicht von Anfang an benötigen, sondern die wir beim Start unserer Anwendung erst einmal aus dem DOM entfernen, dem Browser unnötige Arbeit aufbürden: so werden die betreffenden Elemente vom Browser ja zunächst tatsächlich als gewöhnliche Bestandteile des HTML Dokuments behandelt und verarbeitet.

Letztere Problematik motiviert denn auch die Bereitstellung eines neuen Ausdrucksmittels für HTML in Form eines Elements `<template>`. Dieses soll zwar über die DOM API verfügbar sein, jedoch nicht bereits beim initialen Laden des DOM-Objekts verarbeitet werden. Vielmehr soll seine interne Struktur erst beim Zugriff aus der betreffenden Anwendung heraus aufgebaut werden und damit den initialen Overhead ersparen, der von uns vorgeschlagenen Lösung zweifelsohne attestiert werden kann. Die Standardisierung von `<template>` ist zum gegenwärtigen Zeitpunkt allerdings noch nicht abgeschlossen – siehe dafür z. B. die [Mozilla Dokumentation](#).

2.5 Client Server Interaktion

Alle bisher von uns betrachteten Ausdrucksmittel bezogen sich auf die Steuerung einer in einem Browser ausgeführten Webanwendung bzw. auf die Darstellung von deren Ansichten. Einen Server hatten wir bisher nur verwendet, um die statischen Ressourcen bereitzustellen, die für die Ausführung und Darstellung der Anwendung im Browser erforderlich waren – dazu gehören insbesondere die zu ladenden HTML-Dokumente, JavaScript-Dateien sowie CSS-Stylesheets, aber auch etwaige Bildressourcen, die aus diesen heraus referenziert werden.

Im vorherigen Abschnitt 2.4 hatten wir auch gezeigt, wie das Neuladen eines HTML-Dokuments aus browserseitig ausgeführtem JavaScript heraus veranlasst werden kann. In diesem Fall wird jedoch das DOM-Objekt, das aus dem zuvor geladenen HTML-Dokument aufgebaut und ggf. durch JavaScript manipuliert wurde, durch ein neues Objekt ersetzt, und auch der Ausführungskontext von JavaScript wird unter Verlust zuvor von uns gesetzter Variablenwerte von neuem initialisiert.

Der vorliegende Abschnitt wird nun ein Ausdrucksmittel einführen, das wir in JavaScript verwenden können, um Inhalte von einem Server zu laden und gleichzeitig das DOM-Objekt beizubehalten, das der aktuellen Darstellung zugrunde liegt. In Abhängigkeit von den Inhalten, die uns der Server geliefert hat, können wir auf diesem Objekt dann unter Verwendung der DOM API Manipulationen vornehmen, z. B. die Inhalte in eine visuelle Darstellung überführen. Dabei stehen uns alle Daten, die wir vor dem Zugriff auf den Server im Zuge der JavaScript-Ausführung aufgebaut haben, durchgängig zur Verfügung.

Anstelle einer durch Neuladen von HTML Dokumenten immer wieder veranlassten Neuinitialisierung unserer Anwendung, wird diese also über eine Folge von Serverzugriffen hinweg auf einen gemeinsamen Ausführungskontext zugreifen können. Die Architektur dieser Anwendung wird sich damit an Architekturen annähern können, wie Sie sie evtl. im Zuge der Beschäftigung mit GUIs auf Basis von Java Swing oder anderen Programmiersprachen kennengelernt haben, die lokal auf einem Endgerät ausgeführt werden.

Unsere Anwendung wird jedoch nach wie vor wesentliche Charakteristika einer Webanwendung aufweisen, d. h. sie wird durch einen auf dem Endgerät installierten Browser und unter Verwendung von Ressourcen ausgeführt, die durch einen Server bereitgestellt werden und die unabhängig vom Betriebssystem des Endgeräts sind. Zum Ende dieser Veranstaltung werden Sie darüber hinaus Ausdrucksmittel kennenlernen, um diese Anwendungen unabhängig von der Verfügbarkeit eines Servers auszuführen.

Das Ausdrucksmittel, das uns wesentlich die Umsetzung einer solchen – auch als „Fat Client“ bezeichneten – Architektur ermöglicht, ist eine JavaScript API namens `XMLHttpRequest`. Deren Verwendung werden wir nachfolgend illustrieren. Wie der Name der API andeutet, erlaubt sie uns die Übermittlung von HTTP-Requests an einen Server und die Entgegennahme von Daten aus den vom Server an unsere Anwendung übermittelten HTTP-Responses unter Beibehaltung des aktuell dargestellten DOM-Objekts. Bei jenen Daten kann es sich um strukturierte Inhalte in einem XML konformen Markup, inklusive XHTML, handeln – darauf geht der Name der API ursprünglich zurück. Die API stellt zwar eine gewisse Unterstützung für XML zur Verfügung, erlaubt aber darüber hinaus die Übermittlung von Daten in beliebigen textuellen und binären Formaten. Wir werden sie daher verwenden, um Daten in unsere Anwendung zu laden, die uns der Server im JSON Format bereit stellt, d. h. in der *Java Script Object Notation*, die Sie in dieser Lerneinheit bereits kennengelernt haben.

Ihre Aneignung bzw. Vertiefung der Kenntnisse zur Verwendung von `XMLHttpRequest` wird sich in zwei Schritten vollziehen. Zunächst werden wir im folgenden nach einer allgemeinen Darstellung der Ausdrucksmittel der API deren Verwendung für den lesenden Zugriff auf serverseitige Datenbestände zeigen. Verwenden werden wir dafür zunächst die Inhalte statischer Dateien, die im Dateisystem des Servers vorliegen und die wir mittels URLs referenzieren. Die folgende Lerneinheit wird diese Verwendung auf die Erstellung und Manipulation dynamischer Datenbestände, die serverseitig vorgehalten werden, ausweiten.

Die Vermittlung der Ausdrucksmittel des HTTP-Protokolls, die für die Handhabung von `XMLHttpRequest` relevant sind, wird im Rahmen der Darstellung der API erfolgen. Für eine detaillierte Darstellung verweisen wir auf die Lehrveranstaltung zu *Internet Server Programmierung*, auf die [www.w3.org](#) Spezifikation des W3C und auf einschlägige Publikationen, z. B. [\[Til11\]](#).

2.6 XMLHttpRequest

Wie bereits erwähnt ist [XMLHttpRequest](#) eine – ursprünglich als `ActiveXObject` von Microsoft entwickelte – JavaScript API zur Handhabung von HTTP-Requests und Responses. Sie erlaubt den Aufbau von HTTP-Requests, deren Übermittlung an einen Server und das Auslesen des vom Server erwiderten HTTP-Responses. Als Zugriffsbeschränkung gilt auch hier die *Same Origin Policy*, die verhindert, dass ohne Kontrolle durch den Nutzer Datenzugriffe bezüglich anderer Ursprungsdomains als derjenigen des aktuell geladenen Dokuments durchgeführt werden.

Die API ermöglicht in diesem Rahmen die Handhabung aller auf Client-Seite relevanten Ausdrucksmittel des HTTP-Protokolls, d. h. insbesondere die Angabe einer *URL* und einer *HTTP Methode* wie `GET` oder `POST`, das Setzen von *Request Header* und die Übermittlung eines *Request Body* bzw. einer *Request Entity*. Nach Übermittlung des Responses durch den Server können aus dem für die Übermittlung verwendeten `XMLHttpRequest` Objekt außerdem alle wichtigen Bestandteile des Responses ausgelesen werden, d. h. dessen *Status Code* und *Header* sowie die im *Response Body* übermittelten Daten.

Eine Übersicht der Verarbeitungsschritte, die für eine Client-Server Interaktion auf Client- Seite erforderlich sind, finden Sie in folgender Abbildung. Wir werden diese nun im folgenden anhand der Handhabung von `XMLHttpRequest` illustrieren und kommen unten in der Abbildung „Clientseitige Verarbeitungsschritte einer Client-Server Interaktion mit `XMLHttpRequest`“ wieder darauf zurück.

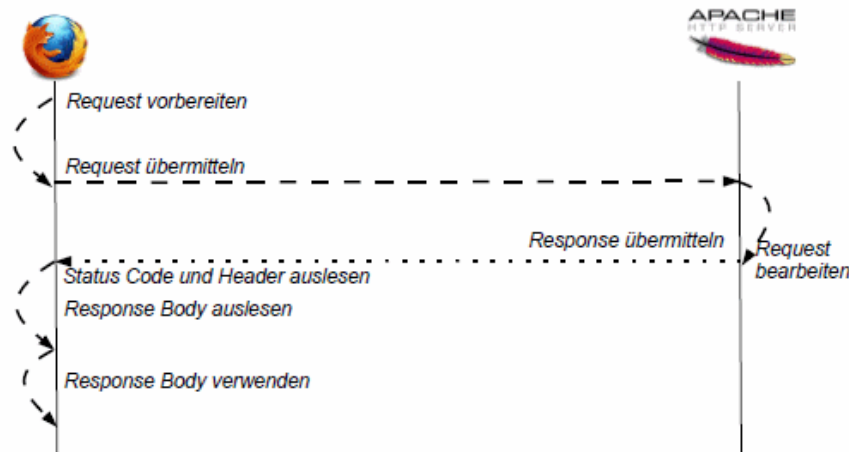


Abb.: Übersicht über die clientseitigen Verarbeitungsschritte einer Client-Server Interaktion auf Basis von HTTP

Die Nutzung von `XMLHttpRequest` beginnt immer mit der Instantiierung eines gleichnamigen Objekts:

```
var xhr = new XMLHttpRequest();
```

Dieses Objekt repräsentiert die gesamte Client-Server Interaktion, die wir im folgenden durchführen werden. Durch die Objektrepräsentation ist es uns insbesondere auch möglich, mehrere Client-Server Interaktionen gleichzeitig durchzuführen und die Responses, die wir dabei erhalten, jeweils dem „richtigen“ Request zuzuordnen.

Request vorbereiten

Für einen einzelnen Request rufen wir zunächst auf dem besagten Objekt die `open()` Funktion auf und benennen dabei die zuzugreifende URL sowie die HTTP-Methode, die wir für den Zugriff verwenden wollen. Wir zeigen dies im folgenden anhand der aktuellen Implementierungsbeispiele, bei denen wir mittels eines `GET`-Requests auf die durch die angegebene URL repräsentierten Daten zugreifen. Aus Sicht des Clients ist es dabei unerheblich, ob dieser URL serverseitig eine konkrete Datei entspricht oder – wie wir später sehen werden – Inhalte, die aus einer Datenbank ausgelesen werden:

```
xhr.open("GET", "/data/die_umsiedlerin.json");
```

Falls erforderlich, können Sie der `open()` Funktion außerdem in zusätzlichen Argumenten Authentifizierungsdaten für Nutzernamen und Passwort übergeben. Nach Ausführung von `open()` können auf dem Request Objekt Header gesetzt werden, die Metadaten bezüglich des Requests beinhalten. So können wir für den vorliegenden Request beispielsweise angeben, dass wir als Erwiderung Daten im JSON Format erwarten:

```
xhr.setRequestHeader("Accept", "application/json");
```

Ob und wie diese Präferenzen allerdings tatsächlich durch den Server berücksichtigt werden, obliegt der jeweiligen Server-Implementierung – zur Illustration der `setRequestHeader()` Funktion möge dieses Beispiel genügen.

Request übermitteln

In unserem Beispiel ist der Request nun so weit vorbereitet, dass wir ihn an den Server übermitteln können. Dafür rufen wir die Funktion `send()` auf:

```
xhr.send();
```

Falls mit dem Request ein *Request Body* übermittelt werden soll – z. B. zur Übertragung von Daten, die serverseitig in eine Datenbank geschrieben werden sollen – wird dieser dem Request Body als Argument übergeben. Beispiele dafür finden Sie in der folgenden Lerneinheit.

Für `GET` Requests wie den hier verwendeten sieht das HTTP-Protokoll jedoch keinen Request Body vor – im Gegensatz z. B. zu Requests mit `POST` Methode. So schreibt denn auch die [www Spezifikation von XMLHttpRequest](#) vor, dass u. a. für `GET` Request kein Body verwendet werden darf.

Mit der Ausführung von `send()` hat der Client zunächst einmal seine Schuldigkeit getan – es bleibt ihm an dieser Stelle erst einmal nichts übrig, als auf die Erwiderung des Servers zu warten. Wenn Sie jedoch in einer Anwendung die gerade beschriebenen Anweisungen ausführen und schließlich die `send()` Methode aufrufen, dann werden Sie feststellen, dass Ihre Anwendung keineswegs auf die Erwiderung wartet, sondern die Ausführung Ihres JavaScript Codes fortsetzt – ungeachtet dessen, dass noch keine Erwiderung des Servers vorliegt. Ursache dieses Verhaltens ist, dass der Browser die `send()` Methode *asynchron* ausführt, d. h. „parallel“ zu allen weiteren Verarbeitungsschritten, die Sie nach Ausführung von `send()` vorgesehen haben. Dies gewährleistet insbesondere, dass ungeachtet etwaiger Verzögerungen bei der serverseitigen Verarbeitung des Requests und / oder bei der Übertragung der Daten zwischen Client und Server die durch den Browser dargestellte Ansicht bedienbar bleibt und nicht „einfriert“.

onreadystatechange

Wie aber können Sie dann darauf reagieren, dass Ihnen der zugriffene Server früher oder später einen Response übermittelt? Voraussetzung dafür ist das Setzen einer von Ihnen implementierten Funktion auf dem `XMLHttpRequest` Objekt, das Sie für die vorliegende Client-Server Interaktion verwenden. Die API sieht dafür ein Attribut namens `onreadystatechange` vor, das insbesondere dann aufgerufen wird, wenn ein Response des Servers eingetroffen ist.

Wenn Sie eine Funktion als Wert dieses Attributs setzen, wird die Implementierung von `XMLHttpRequest` diese Funktion immer dann aufrufen, wenn eine für die Client-Server Interaktion relevante Zustandsänderung vorliegt – dem entsprechend wird eine solche Funktion auch als *Callback-Funktion* bezeichnet. Der erste Aufruf der Funktion erfolgt in unserem Fall bereits nach Aufruf von `open()`, deren erfolgreiche Ausführung eine Änderung des *ready state* von `UNSENT` auf `OPENED` zur Folge hat. Übergeben wird der Funktion dann jeweils ein event Argument, dessen `target` Attribut das `XMLHttpRequest` Objekt ist, für welches die Funktion aufgerufen wird – die `onreadystatechange` Funktion wird durch den Browser also genauso verwendet wie die Event Handler, mit denen wir oben auf Interaktionsereignisse des Nutzers reagieren konnten. Auch diesen wird als `event.target` das Objekt – dort das Bedienelement – übergeben, bezüglich dessen ein Ereignis auftritt.

Die allgemeine Form einer `onreadystatechange` Funktion ist nachfolgend dargestellt. Beachten Sie, dass die Variable `xhr`, sowie die innerhalb der Funktion verwendbaren Ausdrücke `this` und `event.target` jeweils auf dasselbe XMLHttpRequest Objekt verweisen und daher im vorliegenden Fall ausgetauscht werden könnten.

Für den Zugriff auf das Objekt verwenden wir hier und im weiteren den Bezeichner `this`, da dieser am deutlichsten ausdrückt, dass die Funktion eine „Instanzmethode“ des Request Objekts ist und damit unabhängig von etwaigen außerhalb vorgenommenen Setzungen bezüglich `xhr` auf dieses Objekt verweist. Wie Sie sehen, werden die genannten Zustände `UNSENT` und `OPENED` durch die numerischen Konstanten 0 bzw. 1 identifiziert:



Quellcode

Form von onreadystatechange

```
001 // setze eine Callback-Funktion auf dem Request Objekt
002 xhr.onreadystatechange = function(event) {
003
004 // zeige die Äquivalenz der Bezeichner (true wird als Meldung ausgegeben)
005 console.log(event.target == this && this == xhr);
006
007 // überprüfe den readyState
008 switch (xhr.readyState) {
009     case 0:
010         console.log("onreadystatechange: request not initialised yet.");
011         break;
012     case 1:
013         console.log("onreadystatechange: request initialised.");
014         /* (...) */
015     }
016 }
```

Nachdem `send()` ausgeführt wurde, erfolgt der nächste Aufruf an `onreadystatechange` erst dann, wenn vom Server ein Response übermittelt wird. Dessen Auslesen kann jedoch in mehreren Schritten erfolgen, die ebenfalls durch verschiedene Statuswerte für den *ready state* ausgedrückt werden.

Zunächst ist für uns auf Client-Seite von Interesse, ob der Request überhaupt erfolgreich durch den Server bearbeitet wurde oder ob Fehler irgendwelcher Art aufgetreten sind. Diese Information wird uns vom Server durch den auf dem Response gesetzten *Status Code* übermittelt, und auf diesen können wir zugreifen, sobald der ready state den Wert 2 – `HEADERS_RECEIVED` – eingenommen hat. Wie der Name des Status besagt, stehen uns hier dann auch bereits die Response Header zur Verfügung. In vielen Fällen – und auch im hier verwendeten Beispiel – sind wir aber an den Daten interessiert, die uns im Body des Response übergeben werden. Diese werden durch den Browser automatisch ausgelesen, wobei bei Beginn des Auslesens der ready state auf 3 – `LOADING` – gesetzt wird.

Nach erfolgreichem Auslesen der Daten erfolgt abermals eine Änderung des ready state auf den Wert 4 für `DONE` und ein entsprechender Aufruf an die `onreadystatechange` Funktion.

Spätestens jetzt liegen uns clientseitig also alle Daten vor, an denen wir im Erfolgsfall interessiert sind. Eine Minimallösung für `onreadystatechange` sieht damit wie folgt aus:



Quellcode

Minimallösung für onreadystatechange

```
001 // setze eine Callback-Funktion auf dem Request Objekt
002 xhr.onreadystatechange = function(event) {
003     // überprüfe den readyState
004     if (this.readyState == 4) {
005         // überprüfe den Status Code des Response
006         if (this.status == 200) {
007             /* bearbeite den Response */
008         }
009         else {
010             alert("Something went wrong! Got status code: " + this.status);
011         }
012     }
013     else {
014         console.log("readyState is not interesting yet.");
015     }
016 }
```

Auslesen des Response

Sie sehen also, dass über das Attribut `status` auf dem `XMLHttpRequest` Objekt der HTTP Status Code zugreifbar ist, von dessen Wert wir es abhängig machen, ob eine weitere Verarbeitung der übermittelten Daten vorgenommen wird oder ob eine Fehlermeldung ausgegeben wird. Im vorliegenden Fall überprüfen wir dafür nur, ob der Status Code den Wert 200 für OK hat. Die weitere Bearbeitung kann dann z. B. die Berücksichtigung von Response Headern vorsehen, so könnte wie folgt überprüft werden, ob uns vom Server tatsächlich Daten im gewünschten Format geliefert wurden und davon wiederum die weitere Verarbeitung abhängig gemacht werden:



Quellcode

Response Header

```
001 // überprüfe den Content-Type der übermittelten Daten
002 var contentType = this.getResponseHeader("Content-Type");
003 if (contentType == "application/json") {
004     // lies den Response Body aus
005     /* (...) */
006 }
007 else {
008     alert("Cannot handle Content-Type! Got: " + contentType);
009 }
```

Response Body auslesen

Für den Zugriff auf den Response Body stehen uns verschiedene Möglichkeiten zur Verfügung, deren Nutzung davon abhängt, welches Datenformat wir an dieser Stelle erwarten bzw. welches Format uns im Content-Type Header angezeigt wird. So wird uns der „rohe“ Response als Wert des Attributs `response` auf dem `XMLHttpRequest` Objekt bereitgestellt. Erwarten wir aber textuelle Daten oder Daten in XML-Markup, dann erhalten wir diese Daten durch Auslesen der Attribute `responseText` bzw. `responseXML`, wobei letzteres ggf. ein Objekt des Typs `Document` enthält, das wir mittels der DOM API weiter verarbeiten können. Dafür muss allerdings der Content-Type Header des Response auf `text/xml` gesetzt sein, die Setzung von `text/html` ist hier nicht ausreichend. Es besteht aber die Möglichkeit, vor Auslesen des Response durch den Browser, d. h. spätestens im Callback bezüglich des ready state `HEADERS_RECEIVED`, das Attribut `responseType` auf `XMLHttpRequest` auf den gewünschten Typen zu setzen. Die hierfür erlaubten Werte sind [laut Spezifikation](#) „arraybuffer“, „blob“, „document“, „json“ und „text“. Falls der Response entsprechend dem angegebenen Typ erfolgreich ausgelesen werden kann, steht uns dieser danach im gewünschten Format als Wert des Attributs `response` zur Verfügung. Ohne diese Setzung können Daten, die als Text im JSON Format übermittelt werden, aber auch wie folgt ausgelesen und weiterverarbeitet werden:

```
001 // lies den Response Body als Text aus und baue daraus ein JSON Objekt auf.
002 var jsonObj = JSON.parse(this.responseText);
003
004 // verarbeite das Objekt weiter
005 processJsonResponse(jsonObj);
```

Die hier angenommene Funktion `processJsonResponse()` führt dann die von der jeweiligen Anwendung gewünschte Weiterverarbeitung der vom Server übermittelten Daten durch. An dieser Stelle kehren wir also von der Ebene der Client-Server Interaktion zurück in unsere Anwendung und können z. B. auf Grundlage der Daten, die uns übergeben worden sind, dynamische Ansichtselemente aufbauen, wie es die Implementierungsbeispiele zeigen.

Eine Zusammenfassung der durch `XMLHttpRequest` ermöglichten Schritte der Client- Server Interaktion finden Sie in folgender Abbildung. Dort nehmen wir Bezug auf die verallgemeinerte Darstellung in vorherigen Abbildung „Übersicht über die client-seitigen Verarbeitungsschritte einer Client-Server Interaktion auf Basis von HTTP“ und illustrieren anhand dessen noch einmal die Verwendung der API Funktionen von `XMLHttpRequest` und die in den einzelnen Schritten jeweils gesetzten Werte für ready state.

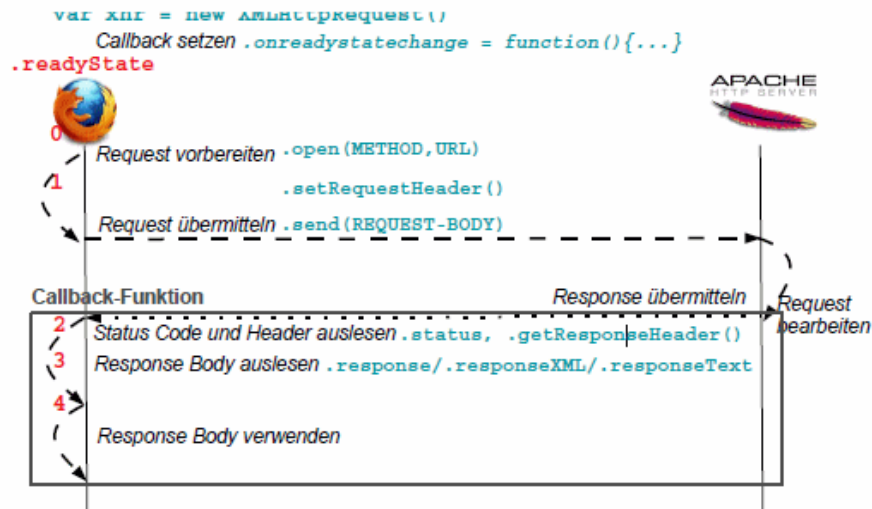


Abb.: Client-seitige Verarbeitungsschritte einer Client-Server Interaktion mit XMLHttpRequest

xhr()

Wenn Sie sich die einzelnen Schritte bei der Verwendung von XMLHttpRequest noch einmal vergegenwärtigen, wirft sich für Sie evtl. die Frage auf, „ob das ganze nicht einfacher gemacht werden kann“. Tatsächlich ist die API nicht unbedingt leichtgewichtig in dem Sinne, dass Sie als Softwareentwickler für jeden einzelnen Fall, in dem Sie aus JavaScript heraus asynchron mit einem Server kommunizieren wollen, alle hierfür erforderlichen Schritte von neuem implementieren wollten. Vielmehr werden Sie sich fragen, ob und inwieweit diese Schritte so verallgemeinert werden können, dass Sie idealerweise mit lediglich einer Zeile Programmcode einen XMLHttpRequest ausführen können. In diesem Sinne stellen Ihnen auch die Implementierungsbeispiele eine generische Funktion namens `xhr()` bereit, der Sie die folgenden Argumente übergeben können:

- Die zu verwendende HTTP Methode
- Die zu verwendende URL
- Optional ein via `send()` zu übermittelndes JSON-Objekt
- Eine Callback-Funktion, die im Erfolgsfall aufgerufen wird, d. h. für `readyState DONE` und Status Code 200
- Eine Callback-Funktion, die für andere Status Codes aufgerufen wird

Was die Implementierung Ihnen nicht abnimmt, ist das Auslesen der Daten aus dem HTTP-Response im Erfolgsfall – umgekehrt sind Sie damit auf Seiten Ihrer Anwendung flexibel bezüglich des Typs dieser Daten und können `xhr()` sowohl für den Zugriff auf JSON-Daten verwenden, als auch für das Laden von HTML-Fragmenten, die in die Ansichten Ihrer Anwendung eingebunden werden sollen. Mit dieser Funktion könnte das oben zugrunde gelegte Beispiel wie folgt umgesetzt werden – wir verzichten hier auf die Angabe einer Callback-Funktion für den Fehlerfall:



Quellcode

xhr() Element

```

001 xhr("GET", "/data/die_umsiedlerin.json", null, function(xmlhttp) {
002   // lies den Response Body als Text aus und baue daraus ein JSON Objekt auf.
003   var jsonObj = JSON.parse(this.responseText);
004   // verarbeite das Objekt weiter
005   processJsonReponse(jsonObj);
006 })
  
```

Auch die einschlägigen JavaScript-Frameworks wie jQuery stellen Ihnen vergleichbare – und funktional wie implementatorisch bei weitem ausgereifere – Komfortfunktionen für die Ausführung von XMLHttpRequest zur Verfügung. Sollte deren Ausdrucksmittel jedoch nicht ausreichen, steht es Ihnen jederzeit frei, direkt auf die XMLHttpRequest API zuzugreifen.

In diesem Sinne war es die Absicht der vorstehenden Ausführungen, einen ersten „Blick unter die Haube“ und auf die API selbst zu werfen, um Ihnen anhand der hier beschriebenen Grundfunktionen die Einarbeitung in fortgeschrittene Verwendungen von `XMLHttpRequest` zu erleichtern. Insbesondere die Verwendung von Callback- Funktionen, die auch bei Nutzung unserer eigenen `xhr()` „Komfortfunktion“ erforderlich sind, wird Ihnen in den folgenden Lerneinheiten aber auch als Ausdrucksmittel anderer JavaScript APIs begegnen.

2.7 AJAX

Die in den vorstehenden Abschnitten mit DOM Events, DOM API und `XMLHttpRequest` eingeführten Ausdrucksmittel stellen die Grundlage von Anwendungsfunktionen dar, die üblicherweise mit dem Akronym AJAX bezeichnet werden, das aufgelöst als *Asynchronous JavaScript and XML* gelesen werden kann. Was damit gemeint ist, lässt sich mit Blick auf die Implementierungsbeispiele umschreiben als die Zusammenführung der beiden folgenden Aspekte:

- Verarbeitung von Interaktionsereignissen bzw. initialer Aufbau von Ansichten mittels asynchroner HTTP Requests.
- Verwendung der HTTP Response Daten zur partiellen Aktualisierung der dargestellten Ansicht mittels Nutzung der DOM API.

Während der im Begriff enthaltene Verweis auf „*asynchrones JavaScript*“ sich also eindeutig auf die `XMLHttpRequest` API bezieht, bezeichnet das „X“ im Akronym nicht nur die Tatsache, dass als Response-Daten XML Dokumente oder auch (X)HTML Fragmente verwendet werden können.

So ist denn AJAX unserer Ansicht nach unabhängig vom konkreten Repräsentationsformat der Response Daten, als welches JSON mittlerweile weiter verbreitet zu sein scheint als XML. Weitaus essentieller für AJAX ist vielmehr die DOM API und die Tatsache, dass das DOM-Objekt ein XML-Dokument ist und dass Manipulationen dieses Objekts unmittelbar in einer Aktualisierung der dargestellten Ansicht durch den Browser resultieren.

Wie angesichts dessen das Zusammenspiel von JavaScript und CSS realisiert werden kann, möchten wir abschließend unter Betrachtung von Implementierungsbeispielen sowie von Anforderungen des Übungsprogramms exemplarisch darlegen.

3 Abstimmung von HTML, CSS und JavaScript

Wenn wir auf die behandelten Ausdrucksmittel der Webtechnologien HTML, CSS und JavaScript zurückblicken, wirft sich die Frage auf, wie diese für die Umsetzung konkreter Anforderungen in geeigneter Weise abgestimmt werden können. Dies betrifft insbesondere Aspekte dynamischen Verhaltens einer Anwendung als Reaktion auf Nutzerinteraktion, z. B. die Realisierung verschiedener Formen von Feedback, die wir ohne Neuladen eines HTML Dokuments realisieren können.

Diesbezüglich wird dieser Abschnitt eine Anforderung zum Ansichtsübergang aus den Gestaltungsvorlagen sowie „Toasts“ als ein für mobile Anwendungen geeignetes und in Android weit verbreitetes Feedbackelement betrachten. Anhand von deren Realisierung werden wir aufzeigen, wie ein Zusammenspiel der besagten Ausdrucksmittel umgesetzt werden kann. Wie bei allen Softwareentwicklungsmaßnahmen, die über die Nutzung einer einzigen API hinausgehen und die Verknüpfung verschiedener Teilfunktionen beinhalten, gibt es auch in den betrachteten Fällen nicht „die eine einzig richtige Lösung“.

Die nachfolgenden Vorschläge sind daher nur als exemplarische Empfehlung zu betrachten.

3.1 Ansichtsübergang

Wie in folgender Abbildung gezeigt, sieht das Designkonzept für die hier betrachtete Anwendung vor, dass von der Übersichtsansicht zu einem Thema – hier dem Werk „Die Umsiedlerin“ – Übergänge zu Detailansichten möglich sind. Je nach ausgewähltem Gestaltungselement stellen diese Ansichten aber keine neuen Inhalte dar, sondern zeigen Inhalte, die bereits in der Übersicht zu sehen waren, in anderer Form an. Beispielsweise werden in der Detailansicht für „Bilder“, die in der Übersicht bereits sichtbaren Bilder im Großformat dargestellt und mit einer Bildunterschrift versehen.

Auch bei der Detailansicht für das Element „Einführungstext“ handelt es sich um eine Expansion des bereits in der Übersicht als Ausschnitt gezeigten Textes. In beiden Fällen wird zudem das Gesamtlayout der Darstellung modifiziert. So ist beispielsweise die Überschrift „Die Umsiedlerin“ nach wie vor zu sehen, wird aber mit abweichender Größe und Textfarbe dargestellt. Auch die Hintergrundfarbe wird jeweils angepasst.

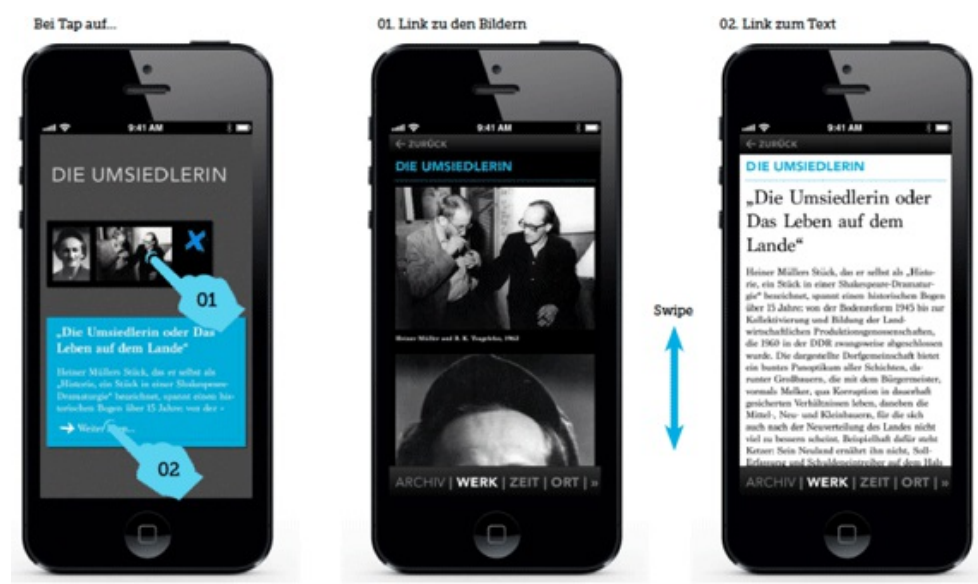



Abb.: Designvorlage für den Übergang zwischen Übersichtsansicht und Detailansicht für die beiden Elemente Bilder und Einführungstext

Natürlich wäre es zur Umsetzung dieser Anforderung denkbar, für Übersicht und Detailansichten jeweils verschiedene HTML-Dokumente mit unterschiedlichen Stylesheets zu verwenden und die Übergänge mittels Neuladen der Dokumente für die Detailansichten bzw. im Fall der Rückkehr zur Übersicht unter Verwendung der  Browser History umzusetzen. Aus den vorstehenden Erläuterungen geht jedoch hervor, dass die Detailansichten gar keine wirklich „neuen“ Inhalte verwenden, sondern nur Inhalte, die bereits in der Übersicht vorhanden und zu sehen waren, in anderer Form darstellen.

Im vorliegenden Gestaltungskonzept können die beiden Ansichtstypen damit als verschiedene Perspektiven auf *dieselben* Inhalte aufgefasst werden. Dies lässt die Verwendung eines einzigen HTML-Dokuments zur Bereitstellung dieser Inhalte als sehr nahe liegend erscheinen, während eine Verteilung auf mehrere Dokumente redundant wirkt.

Nimmt man nun an, dass ein gemeinsames HTML-Dokument für beide Ansichtstypen verwendet wird, dann stellt sich die Frage, wie dieses Dokument strukturiert sein sollte, um den Ansichtsübergang möglichst geradlinig, d. h. idealerweise ohne nennenswerte Umstrukturierungen des Dokuments zu realisieren. Diesbezüglich betrachten wir noch einmal die in der Lerneinheit HTM vorgeschlagene Dokumentenstruktur. Wir nehmen hier an, dass die einzelnen `<section>` Elemente mit einem `class` Attribut versehen sind, das den Typ des betreffenden Elements bezeichnet:



Quellcode

section Element mit class Attribut

```
001 <body>
002 <header><!-- (...) --></header>
003 <article>
004 <h1>Die Umsiedlerin</h1>
005 <!-- scrollable area -->
006 <div class="scrollview">
007 <section class="zeitdokumente">
008 <h2>Zeitdokumente</h2>
009 <!-- (...) -->
010 </section>
011 <section class="bilder">
012 <!-- (Bilder) -->
013 <!-- (...) -->
014 </section>
015 <section class="einfuehrungstext">
016 <!-- (Einführungstext) -->
017 <!-- (...) -->
018 </section>
019 <!-- (...) -->
020 </div>
021 <!-- end of scrollable area -->
022 </article>
023 <footer><!-- (...) --></footer>
024 </body>
```

Um die gewünschte Gestaltung der Detailansichten umzusetzen, müssen wir auf Ebene unserer CSS-Regeln zum einen die Überprüfung ermöglichen, ob eine Detailansicht umgesetzt werden soll und um welche Detailansicht es sich handelt. Von letzterer Information hängt nämlich nicht nur ab, welches `<section>` Element angezeigt werden soll und welche Elemente ausgeblendet werden sollen. Auch für die Entscheidung über die Formatierung der Ansicht, insbesondere für die Auswahl der Hintergrundfarbe, benötigen wird diese Information, welche wir mittels geeigneter Zuweisungen für `class` ausdrücken werden.

Zusätzlich müssen wir aber klären, auf welcher Ebene des Dokuments, d. h. auf welchem oder welchen der oben dargestellten Elemente diese Information benötigt wird. Hierfür lässt sich eine – evtl. trivial erscheinende – Regel anwenden, derzufolge für eine Unterscheidung zwischen unterschiedlichen Gestaltungsvarianten ein Element markiert werden sollte, das alle zu gestaltenden Elemente einschließt. Im vorliegenden Fall beziehen sich die Gestaltungsvarianten nur auf die Inhalte zwischen Kopf- und Fußleiste. Als Element für die Markierung bietet sich daher das `article` Element an.

Kopf- und Fußleiste in der Abbildung

in der vorherigen Abbildung "Designvorlage..." links sind diese Elemente in der Übersicht nicht zu sehen, da sie gemäß Design unabhängig von der verwendeten Ansicht dynamisch ein- und ausgeblendet werden sollen.

Für die Markierung mittels `class` Zuweisungen erlauben wir, die oben genannten Überprüfungen – liegt eine Detailansicht vor und, wenn ja, welche? – differenziert durchzuführen. Daher verwenden wir als Werte für `class` zum einen den Wert `detailview`, der uns anzeigt, dass eine Detailansicht vorliegt, zum anderen die Werte `detailview-bilder` und `detailview-einfuehrungstext`, mittels derer die jeweilige spezifische Detailansicht identifiziert werden kann. Für die Umsetzung der beiden Ansichten würde also das oben gezeigte `<article>` Element wie folgt modifiziert:



Quellcode

Detailansicht für Bilder

```
001 <!-- Detailansicht für bilder -->
002 <article class="detailview detailview-bilder">
003   <!-- (...) -->
004 </article>
```



Quellcode

Detailansicht für Einführungstext

```
001 <!-- Detailansicht für einfuehrungstext -->
002 <article class="detailview detailview-einfuehrungstext">
003   <!-- (...) -->
004 </article>
```

Betrachten wir nun einmal, ob die hier gesetzte Information ausreicht, um die Detailansichten in Abgrenzung von der Übersichtsansicht darzustellen. Das wichtigste Differenzierungsmerkmal hierfür ist der Wert `detailview` für das Wurzelement `<article>`. Diesen können wir in allen erforderlichen Regeln mittels eines Class-Selektors überprüfen, so lässt sich z. B. die Zuweisung der Stilmerkmale für die `<h1>` Überschrift in Abhängigkeit von diesem Wurzelement wie folgt umsetzen:



Quellcode

Überschrift für Detailansichten

```
001 /* Überschrift für Detailansichten */
002 .detailview h1 {
003   font-family: "Avenir Medium", sans-serif;
004   font-size: 15pt;
005   color: rgb(63,169,245);
006 }
```

Beachten Sie, dass die explizite Benennung des `<article>` Elements hier und im folgenden nicht erforderlich ist, sondern dass der Class-Selektor ausreicht, um die gewünschte Differenzierung zu erzielen. Durch diese Bezugnahme auf anwendungsspezifische `class` Werte entkoppeln wir unsere CSS-Regeln also von den konkreten Markup-Elementen, auf denen diese Werte gesetzt sind und gewinnen eine gewisse Flexibilität für etwaige Modifikationen des Markups, die zu einem späteren Zeitpunkt ggf. erforderlich sind.

Für die Zuweisung der Hintergrundfarbe können wir uns zusätzlich des `class` Werts bedienen, der uns anzeigt, welche spezifische Detailansicht vorliegt:



Quellcode

Hintergrund für Detailansicht

```
001 /* Hintergrund für Detailansicht "bilder" */
002 .detailview.detailview-bilder {
003   background-color: rgb(15,15,15);
004 }
005 /* Hintergrund für Detailansicht "einfuehrungstext" */
006 .detailview.detailview-einfuehrungstext {
007   background-color: rgb(255,255,255);
008 }
```

Reicht das Markup aber auch aus, um das Ausblenden der nicht dargestellten Elemente der Übersicht umzusetzen bzw. die Beibehaltung des ausgewählten Elements zu realisieren? Betrachten wir dafür zunächst die folgende Regel, die alle `<section>` Elemente durch Zuweisung der Stileigenschaft `display: none` ausblendet:



Quellcode

Ausblenden aller section Elemente in Detailansichten

```
001 /* Ausblenden aller section Elemente in Detailansichten */
002 .detailview section {
003   display: none;
004 }
```

Diese Regel können wir jedoch für die jeweils darzustellenden Elemente sehr einfach überschreiben:



Quellcode

Darstellung der für die Detailansichten erforderlichen Elemente

```
001 /* Darstellung der für die Detailansichten erforderlichen Elemente */
002 .detailview.detailview-bilder section.bilder {
003     display: block;
004 }
005 /code>
006 code data-language="css">
007 .detailview.detailview-einfuehrungstext section.einfuehrungstext {
008     display: block;
009 }
```

Mittels der hier verwendeten Selektoren lassen sich auch jegliche anderen Stilmodifikationen gegenüber der Darstellung eines Elements in der Übersicht umsetzen, z. B. die Darstellung des kompletten Textinhalts für „Einfuehrungstext“ im Gegensatz zum Abschneiden überschüssiger Inhalte in der Gesamtansicht. Die Verwendung des Typsektors bezüglich `<section>` ist hier allerdings nicht notwendigerweise erforderlich, zum Zweck der transparenten Abgrenzung gegenüber der vorgenannten Ausblenderegeln erscheint sie uns jedoch sinnvoll.

Der vorgenommene Einblick in die Verwendung von CSS möge an dieser Stelle genügen, um zu illustrieren, wie Ansichtswechsel, die auf den ersten Blick einen erheblichen Eingriff in die Struktur von HTML-Dokumenten erfordern, mittels geeigneter `class` Setzungen sowie einer den Gestaltungsanforderungen angemessenen HTML-Dokumentenstruktur „minimal invasiv“ umgesetzt werden können. Zur Erfüllung der oben formulierten Regel bezüglich des für die Umsetzung von Gestaltungsalternativen auszuwählenden Wurzelements konnte im vorliegenden Fall das bereits im Markup vorgesehene `<article>` Element verwendet werden.

Falls kein solches Element existierte, läge hier eine Indikation für die Einführung eines `<div>` Elements vor, das mit den erforderlichen anwendungsspezifischen `<class>` Werten ausgestattet wird. Die Hinzufügung eines neuen Wurzelements zur Markierung einer anwendungsspezifischen Gestaltungseinheit erscheint uns in jedem Fall eine bessere Lösung als die gleichermaßen denkbare Markierung der jeweiligen darunter liegenden Elemente mit den erforderlichen `class` Werten – z. B. des `<h2>` Elements und des als `<scrollview>` markierten `<div>` Elements im vorliegenden Fall.

Letztere Erwägung ist nicht zuletzt auch dadurch motiviert, dass wir die Hinzufügung bzw. die Entfernung von `class` Werten, die zur Unterscheidung zwischen zwei Gestaltungsalternativen erforderlich sind, aus JavaScript heraus initiieren müssen. So haben wir bisher ja nur die einzelnen Darstellungsalternativen hinsichtlich ihrer Dokumentstruktur und der differenzierenden `class` Werte sowie anhand der notwendigen CSS Regeln betrachtet. Die tatsächliche Umsetzung des Ansichtswechsels als Reaktion auf eine Nutzereingabe obliegt jedoch der Ausführung geeigneter JavaScript Anweisungen. Diesbezüglich ist insbesondere eine API relevant, die uns die Handhabung der Werte des `class` Attributs erheblich erleichtert. So werden die Werte des Attributs auf Ebene des DOM Objekts als [!\[\]\(aa53ad6fea213b8b2226d3077e30533a_img.jpg\) `classList` Objekt](#) repräsentiert.

Wie die nachfolgenden Beispiele zeigen, können Sie auf dieses Objekt über das `classList` Attribut eines aus dem DOM ausgelesenen Elements zugreifen. Auf `classList` stehen uns dann die sprechend benannten Funktionen `add()`, `remove()` zum Hinzufügen und Entfernen von Werten, die boolesche Funktion `contains()` zur Überprüfung eines Werts sowie die Komfortfunktion `toggle()` zur Verfügung, die als „Kippschalter“ einen nicht vorhandenen Wert hinzufügt bzw. einen vorhandenen Wert entfernt. Diese Funktion nutzen wir im folgenden Beispiel, um zwischen einer Detailansicht für „Einfuehrungstext“ und der Übersichtsansicht hin- und herzuschalten – wir nehmen an, dass wir das Element, auf dem die `class` Werte gesetzt werden sollen, außerhalb dieser Funktion bereits instantiiert haben:

```

001 /* lies das Wurzelement aus, auf dem die class Werte gesetzt/entfernt
002 werden sollen */
003 var maincontent = document.getElementsByTagName("article")[0];
004 // deklariere eine Funktion, um zwischen den Ansichten umzuschalten
005 function toggleDetailViewEinfuehrungstext() {
006     maincontent.classList.toggle("detailview");
007     maincontent.classList.toggle("detailview-einfuehrungstext");
008 }

```

Analog könnte die Funktion für das Umschalten auf die Detailansicht für „Bilder“ realisiert werden, und natürlich sind hierfür auch diverse Generalisierungen denkbar, bei denen wir die für beide Ansichtswechsel erforderlichen Anweisungen in gemeinsamen Funktionen „ausfaktorisieren“ und diese mittels Argumenten parametrisieren, z. B. wäre es denkbar, die obige Funktion wie folgt zu generalisieren:

```

001 // parametrisierte Ansichtsumschaltung
002 function toggleDetailView(viewtype) {
003     maincontent.classList.toggle("detailview");
004     maincontent.classList.toggle("detailview-" + viewtype);
005 }

```

Eine Generalisierung ist auch bezüglich der nachfolgend betrachteten Maßnahme denkbar, die noch offen ist, um den Ansichtswechsel tatsächlich durchführen zu können. So ist dafür ja eine Eingabe auf Seiten des Nutzers notwendig, auf die wir mit einem geeigneten Event Handler reagieren können. Für „Einführungstext“ können wir diesen z. B., wie aus der Gestaltungsvorlage hervorgeht, auf dem DOM-Element setzen, mit dem wir das Bedienelement „*Weiter lesen...*“ realisieren – dieses nehmen wir nachfolgend als in der Variable `gotoDetailViewEinfuehrungstextAction` gegeben an:

```

// veranlasse bei Bedienung von "weiter lesen..." einen Ansichtswechsel:
gotoDetailViewEinfuehrungstextAction.onclick = toggleDetailViewEinfuehrungstext;

```

Beachten Sie, dass die betreffende `toggle()` Funktion auch bei Aufruf des „Zurück“ Bedienelements in der Detailansicht aufgerufen werden muss und dass auch hierfür einige alternative Realisierungsmöglichkeiten existieren. Da allen Ansichten ein gemeinsames HTML-Dokument zugrundeliegt, könnte z. B. ein `onclick` Handler bezüglich „Zurück“ mittels einer Verzweigung entscheiden, aus welcher Ansicht heraus die Betätigung des Elements erfolgt und sich dafür gleichermaßen auf eine Überprüfung der `class` Werte auf `<article>` verlassen. Alternativ könnten aber bei einem Ansichtswechsel auch die Event Handler auf „Zurück“ ausgetauscht werden, was eine stärkere Modularisierung der Implementierung ermöglicht, andererseits aber auch die Umsetzung der `toggleDetailView...()` Funktionen aufwendiger machen würde.

3.2 Schrittweises Ein- und Ausblenden von Elementen

Die im vorigen erarbeiteten Grundzüge einer Arbeitsteilung zwischen HTML, CSS und JavaScript werden wir nun in einem weiteren Anwendungsfall nutzen, bei dem wir eine selbstausblendende Überlagerung einer Ansicht in Form eines „Toasts“ realisieren. Diese Elemente sind in Android weit verbreitet – die folgende Abbildung zeigt hierfür ein Originalbeispiel sowie einen von uns „nachgebauten“ Toast auf einem Android Tablet. Für die Umsetzung werden wir mit CSS *Transitions* ein bisher noch nicht eingeführtes Ausdrucksmittel nutzen, das uns in der aktuellen Version von CSS zur Verfügung steht.

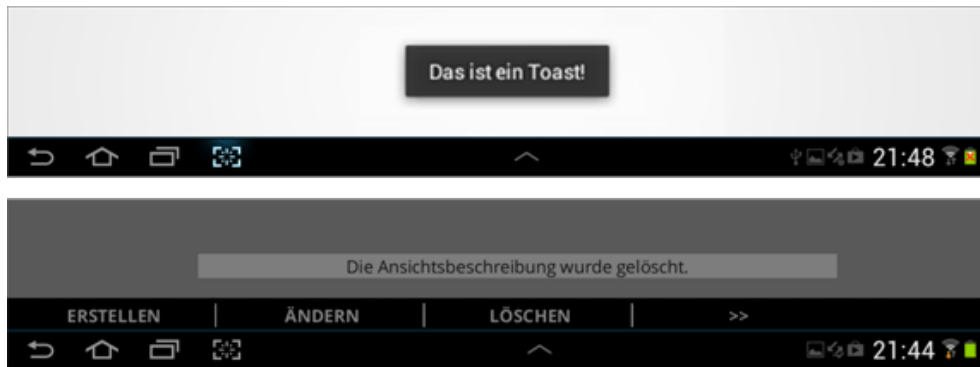


Abb.: Native Toasts in Android

CSS-Transitions

Transitions erlauben uns, allein auf Ebene von CSS dynamische Verhaltensaspekte von Ansichten festzulegen. Eingeschränkt ist ihre Verwendung dabei allerdings auf solche Verhaltensaspekte, die sich als Übergang zwischen zwei Mengen von Stileigenschaftszuweisungen beschreiben lassen. Außerdem können nur solche Stileigenschaften berücksichtigt werden, deren Wertebereich sich als numerische Skala ausdrücken lässt – dazu gehören u. a. auch Farbwerte. In der [www Mozilla Dokumentation](http://www.mozilla.org/docs/Transitions) werden Transitions anhand der folgenden Abbildung illustriert, bei der die Dimensionen eines Bildes einen Übergang von einem Ursprungszustand mit einer Menge von Eigenschaften P_s zu einem Zielzustand P_t durchlaufen.

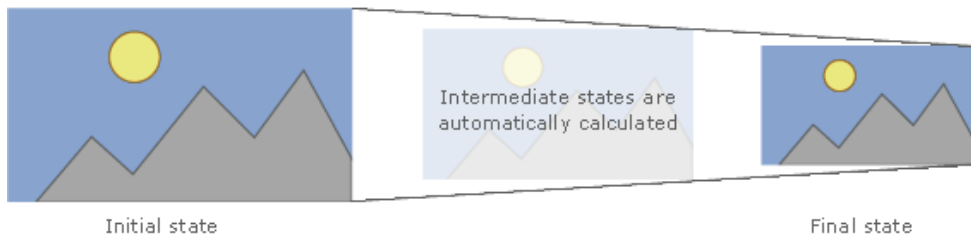


Abb.: Transition

Dem von uns in diesem Abschnitt betrachteten Beispiel liegen die drei nachfolgend verwendeten Stileigenschaften `background-color`, `color` und `opacity` zugrunde:

- P_s : {background-color: white; opacity: 0.0; color: gray;}
- P_t : {background-color: gray; opacity: 1.0; color: black;}

Um eine Transition als graduellen durch den Browser kontrollierten Übergang von P_s nach P_t zu veranlassen, können wir angeben, welche der Eigenschaften dafür verwendet werden sollen und wie lange der Übergang dauern soll. Dies resultiert dann darin, dass ausgehend vom Ursprungszustand die Wertzuweisungen bezüglich der genannten Eigenschaften über die angegebene Dauer hinweg graduell modifiziert werden, bis der für den Zielzustand angegebene Wert erreicht ist. Die folgende Transition würde bezüglich der genannten Eigenschaften z. B. nur die durch `opacity` angegebene Durchsichtigkeit des betreffenden Elements modifizieren, eine Änderung von `background-color` und `color` jedoch ohne Verzögerung und Wertanpassung durchführen.

```
/* nimm über die Dauer von 3s eine Anpassung der opacity vor */
transition: opacity 3s;
```


Sollen mehrere Stileigenschaften für eine Transition berücksichtigt werden, kann dies wie folgt notiert werden – damit können für unterschiedliche Eigenschaften grundsätzlich auch unterschiedliche Werte für die Dauer des Übergangs angegeben werden:

```
transition: opacity 3s, background-color 3s, color 3s;
```

Damit eine Transition ausgeführt wird, ist es notwendig, dass eine solche `transition` Zuweisung zu den Stilzuweisungen gehört, die CSS für den gewünschten *Zielzustand* ermittelt. Falls auch für den Ausgangszustand eine `transition` Zuweisung vorliegt, wird diese erst dann berücksichtigt, wenn dieser Zustand selbst als Zielzustand fungiert. Dies ist z. B. in dem von uns betrachteten Beispiel von Toasts der Fall. Grundlage dafür stellt zunächst ein Markup dar, in welchem wir den Toast mittels eines `<div>` Elements modellieren. Dieses Element platzieren wir innerhalb des `<body>`, aber außerhalb bzw. neben dem Wurzelement der Ansicht, über die der Toast eingeblendet werden soll:

```
001 <body>
002   <!-- (...) -->
003   <div class="toast">lorem ipsum</div>
004 </body>
```

Dem Toast weisen wir eine fixe Position zu, die wie folgt relativ zu den Dimensionen des `<body>` ermittelt wird. Falls Sie im Rahmen eines responsiven Designs verschiedene Geräteklassen bzw. Bildschirmgrößen vorsehen, wäre hier auch jeweils eine Positionierung mit absoluten Größen denkbar. Darüber hinaus setzen wir Werte für `background-color`, `color` und `opacity`, die als Ausgangspunkt und Zielzustand der Transitions für das Ein bzw. Ausblenden des Toasts dienen, wobei das Toast Element bei einer `opacity` von 0.0 unsichtbar ist. Schließlich setzen wir die `<transition>` Eigenschaft auf den Wert, den wir bereits oben gezeigt haben:



Quellcode

Position von Toasts und Stileigenschaften für inaktive Toasts

```
001 /* Position von Toasts und Stileigenschaften für inaktive Toasts */
002 .toast {
003   /* Positionierung */
004   width: 60%;
005   position: fixed;
006   text-align: center;
007   bottom: 15%;
008   left: 20%;
009   /* Ausgangszustand vor Einblenden, Zielzustand nach Ausblenden */
010   background-color: white;
011   opacity: 0.0;
012   color: gray;
013   /* Transition bezüglich der Eigenschaften, die das Ein- und Ausblenden
014      bewirken */
015   transition: opacity 3s, background-color 3s, color 3s;
016 }
```

Den Zielzustand des Einblenden eines Toasts definieren wir dann wie folgt – beachten Sie, dass hier nur noch die Stilzuweisungen erforderlich sind, die die Transition als Zielwerte verwenden soll. Die Zuweisung bezüglich `transition` ist hingegen nicht erforderlich, da sie für ein als *active* markiertes Toast-Element aus der vorstehenden Regel übernommen wird. Letzteres gilt auch für die Positioneigenschaften:



Quellcode

Stileigenschaften aktiver Toasts

```
001 /* Stileigenschaften aktiver Toasts */
002 .toast.active {
003   background-color: gray;
004   color: black;
005   opacity: 1.0;
006 }
```

Im Zuge des Einblenden eines Toasts wird also die Transparenz des Elements schrittweise von „vollständig transparent“ auf „vollständig intransparent“ (= „opak“) gesetzt und gleichzeitig Hintergrund und Schriftfarbe von weiß auf grau bzw. grau auf schwarz angepasst. Bleibt uns also nur noch die Aufgabe, das Einblenden des Toasts aus JavaScript heraus zu veranlassen. Dafür verwenden wir eine „Komfortfunktion“ `showToast()`, der auch der anzuzeigende Text übergeben werden kann. Auch hier nehmen wir wieder an, dass bereits eine für uns zugreifbare Variable mit

dem Toast-Element instantiiert wurde:

```
001 /* lies das Toast Element aus */
002 var toast = document.getElementsByClassName("toast")[0];
003
004 /* zeige einen Toast mit einem gegebenen Text an */
005 function showToast(text) {
006     toast.textContent = text;
007     toast.classList.toggle("active");
008     /* (...) */
009 }
```

Die Ausführung dieser Funktion bewirkt also zunächst, dass das Toast-Element als `active` markiert wird. Im Zuge der Neudarstellung des auf diese Weise modifizierten DOM-Objekts werden nun *beide* oben gezeigten Regeln für `.toast` und `.toast.active` angewendet, wobei die Stileigenschaften für `background-color`, `color` und `opacity` aus letzterer Regel übernommen werden. Übernommen wird aus der Regel für `.toast` aber auch die Stilmzuweisung für `transition`. Dies unterbindet eine sofortigen Zuweisung der in `.toast.active` angegebenen Stileigenschaften, und resultiert in einer schrittweisen Anpassung über einen Zeitraum von 3 Sekunden, die wir als „Einblenden des Toasts“ wahrnehmen.

Wie aber können wir nun das Ausblenden des Elements veranlassen? Denkbar wäre es, hierfür die in JavaScript verfügbare Funktion `setTimeout()` zu verwenden. Diese erlaubt uns, die Ausführung einer Folge von Anweisungen für eine angegebene Zeit zu verzögern. Dafür können die Anweisungen z. B. wie folgt in einer anonymen Funktion übergeben werden, in der wir nichts anderes tun als die `active` Auszeichnung wieder aus dem `class` Attribut zu entfernen:

```
001 function showToast(text) {
002     toast.textContent = text;
003     toast.classList.toggle("active");
004     /* initiiere das Ausblenden des Toasts nach 3.5 Sekunden */
005     setTimeout(function() {
006         toast.classList.toggle("active");
007     }, 3500);
008 }
```

Nach Ablauf der angegebenen Zeitspanne von 3,5 Sekunden wird also eine erneute DOM Änderung veranlasst, die nun dazu führt, dass die Regeln für `.toast` angewendet werden. Diese ändern insbesondere die drei Stileigenschaften, die wir für das Einblenden modifiziert hatten und setzen diese auf den Ausgangszustand zurück. Da aber auch hier die `transition` Angabe berücksichtigt wird, vollzieht sich auch diese Zurücksetzung schrittweise, und wir nehmen ein „Ausblenden“ des Toasts wahr.

`transitionend` Event

Sie sehen also erneut, wie ein Zusammenspiel von `class` Setzungen in JavaScript und Stilmzuweisungsregeln in CSS verwendet werden kann, um das dynamische Verhalten einer Anwendung zu kontrollieren. Suboptimal ist die vorgeschlagene Lösung allerdings aus dem Grund, dass hier eine inhaltliche Abhängigkeit von JavaScript Anweisungen und CSS Regeln hergestellt wird. So wäre es bei dieser Lösung nicht möglich, allein auf Ebene von CSS die Zeitdauer für das Einblenden des Toasts zu verlängern. Zusätzlich müsste noch das beim Aufruf von `setTimeout()` angegebene Zeitintervall angepasst werden – andernfalls würde ungeachtet der in CSS vorgesehenen Einblendzeit das Ausblenden nach 3.5 Sekunden veranlasst.

Um solche expliziten Abhängigkeiten zu vermeiden, steht uns auf Ebene von JavaScript ein Event namens `transitionend` zur Verfügung, das uns den Abschluss einer CSS Transition bezüglich eines DOM Elements signalisiert. Einen Event Handler bezüglich `transitionend` können wir im genannten Beispiel z. B. auf dem toast-Element setzen. Allerdings kann es erforderlich sein, diesen Event Handler beim ersten Auftreten des Events wieder zu entfernen, um seine wiederholte Ausführung zu verhindern. So würde der Toast in unserem Beispiel aufgrund der Verwendung von `toggle()` ohne eine Entfernung des Handlers in einer Schleife fortwährend ein- und ausgeblendet, da jeder Ein- bzw. Ausblendevorgang den Aufruf des Handlers auslösen würde. Dass der Handler für jede Stileigenschaft, die einer Transition unterzogen wird, einzeln aufgerufen wird, ist ein weiterer Grund dafür, ihn beim ersten Aufruf zu entfernen. Beachten Sie aber, dass hierfür der Event Handler identifizierbar sein muss, d. h. Sie können diesen nicht als anonyme Funktion implementieren:



Quellcode

Ausblenden des Toasts

```

001 /* die Funktion, die das Ausblenden des Toasts veranlasst */
002 function fadeoutToast() {
003     toast.classList.toggle("active");
004     /*entferne fadeoutToast als Event Handler für transitionend */
005     toast.removeEventListener("transitionend", fadeoutToast);
006 }
007
008 function showToast(text) {
009     toast.textContent = text;
010     toast.classList.toggle("active");
011     /* initiiere das Ausblenden des Toasts nach Abschluss der Transition */
012     toast.addEventListener("transitionend", fadeoutToast);
013 }

```

transition-delay
Property

Bei dieser Behandlung von `transitionend` wird jedoch das `toast` Element unmittelbar nach Einblenden des Toasts durch Entfernung der `active` Markierung wieder in den Zustand versetzt, der das Ausblenden veranlasst, d. h. der Toast wäre hier nicht für ein gegebenes Zeitintervall „stabil“ sichtbar. Soll letzteres aber der Fall sein, ließe sich dies durch Verwendung von `setTimeout()` bezüglich der `toggle(„active“)` Setzung erzielen. Wiederum mit Blick auf die wünschenswerte „Arbeitsteilung“ zwischen CSS und JavaScript ist dieser Lösung jedoch die Verwendung der Stileigenschaft `transition-delay` vorzuziehen, mit der die Auslösung einer Transition verzögert werden kann. Im vorliegenden Fall müsste diese für den Zielzustand des Ausblendens gesetzt werden. Soll andererseits das Einblenden des Toasts via `toast.active` keiner Verzögerung unterliegen, müsste hierfür `transition-delay` wie folgt mit einem geeigneten Wert überschrieben werden:

```

001 /* der Rückgang in den (ausgeblendeten) Default-Zustand des Toasts soll um
002     2s verzögert werden */
003 .toast {
004     transition-delay: 2s;
005 }
006
007 /* Beim Einblenden soll keine Verzögerung auftreten */
008 .toast.active {
009     transition-delay: 0s;
010 }

```

Erwähnt sei zuletzt, dass auch die hier gezeigte Lösung noch einen Schönheitsfehler aufweist, da das Toast Element im ausgeblendeten Zustand zwar nicht sichtbar, aber vorhanden ist und aufgrund der vorgenommenen Positionierung möglicherweise vor einem sichtbaren Bedienelement platziert wird und dessen Bedienbarkeit verhindert. Die Behebung dieser Problematik allein auf der Ebene von CSS scheint aber nicht möglich zu sein.

So wäre es zwar denkbar, die beiden Aktivitätszustände von Toasts mit den Werten `display: none` vs. `display: block` oder unterschiedlichen `z-index` Werten zu assoziieren. Da beide Stileigenschaften aber einen diskontinuierlichen Wertebereich haben, kann für sie keine `transition` deklariert werden. Ihre Verwendung würde daher zwar das Einblenden wie gewünscht realisieren, hätte jedoch ein abruptes Verschwinden des Toasts anstelle eines Ausblendens zur Folge. Uns erscheint es daher sinnvoller, das Toast Element in HTML als `hidden="true"` zu markieren und das Attribut in `showToast()` zunächst auf `false` und nach Abschluss des Ausblendens wieder auf `true` zu setzen. Dies erfordert aber nicht nur eine zusätzliche Verwendung von `setTimeout()` für letztere Setzung. Vielmehr scheint auch eine minimale Verzögerung zwischen der `false` Zuweisung an `hidden` und der Aktivierung des Toasts erforderlich zu sein.

Zusammenfassung

- JavaScript wird zunehmend in den Stand einer umfassenden Programmiersprache für die Steuerung clientseitig ausgeführter Anwendungen mit User Interface erhoben.
- Hinsichtlich seiner Ausdrucksmittel zeichnet sich JavaScript gegenüber anderen Sprachen wie Java oder Objective-C unter anderen dadurch aus, dass es keine Typzuweisungen an Variablen inklusive an die einer Funktion übergebenen Argumente erlaubt.
- Objekte können in JavaScript sowohl Attribute, als auch Methoden zur Verfügung stellen und eignen sich damit grundsätzlich für die Kapselung von Daten und Verhalten.
- Unter Bezugnahme auf eine MVC-Architektur lassen sich die einzelnen Schritte, die im Zuge des Verarbeitungszyklus einer Nutzereingabe durchgeführt werden müssen unter Verwendung von JavaScript umsetzen.
- Das Document Object Model (DOM) ist eine Schnittstellenspezifikation, die eine plattformunabhängige API für den lesenden und schreibenden Zugriff auf XML-Dokumente beschreibt.
- Ein wichtiger Aspekt von DOM Events ist die Umsetzung eines durch die Spezifikation beschriebenen Verhaltens, welches als Event Bubbling bezeichnet wird.
- Für den lesenden Zugriff auf das DOM-Objekt stehen in JavaScript verschiedene Funktionen zur Verfügung.
- Die JavaScript API erlaubt die Übermittlung von HTTP-Requests an einen Server und die Entgegennahme von Daten aus den vom Server an eine Anwendung übermittelten HTTP-Responses unter Beibehaltung des aktuell dargestellten DOM-Objekts.
- AJAX verbindet den Zugriff auf serverseitige Daten und Inhalte, die clientseitig zum Zweck einer Manipulation bzw. Aktualisierung der dargestellten Ansichten verwendet werden, ohne dass dafür ein Neuladen des HTML-Dokuments erforderlich ist.

Sie sind am Ende dieser Lerneinheit angelangt. Auf den folgenden Seiten finden Sie noch Übungen.

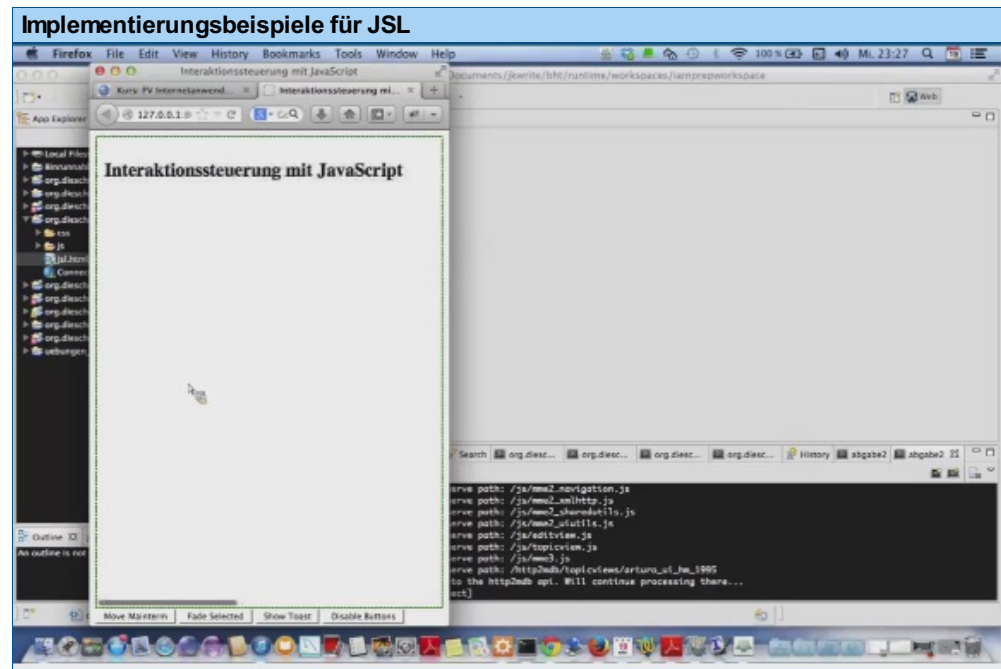
Übungen

Die Implementierungsbeispiele für die vorliegende Lerneinheit finden Sie in zwei Projekten. Die ausschließliche lokale Verwendung von JavaScript wird in `org.dieschnittstelle.iam.jsl` illustriert. Hier finden Sie auch Beispiele für das Zusammenspiel von JavaScript und CSS-Transitions. Das Projekt `org.dieschnittstelle.iam.jsr` zeigt den Zugriff auf serverseitige Ressourcen mittels `XMLHttpRequest`. Die Übungsaufgaben sind ebenfalls in zwei Gruppen zum lokalen (JSL) bzw. remote (JSR) Einsatz von JavaScript unterteilt.

Die prüfungsverbindlichen Übungen und deren Bepunktung werden durch die jeweiligen Lehrenden festgelegt.



Film



© Beuth Hochschule Berlin - Dauer: 06:24 Min. - Streaming Media 12 MB



Formulieren

Übung JSL-01**Fragen zum Implementierungsbeispiele JSL**

Schauen Sie sich die Implementierungsbeispiele zu dieser Lerneinheit an und bearbeiten Sie dann die folgenden Aufgaben und Fragen.

Spaltenlayout

- Wie ist die Nebeneinander-Anordnung der beiden `<section>` Elemente umgesetzt?
- Welche Rolle spielt dabei die Breitenzuweisung an das Element mit ID `articlecontent`? – Reduzieren Sie dafür die Breite z. B. auf 700px und schauen Sie sich den Unterschied an.

Berücksichtigen Sie außerdem den Hinweis zur Dimensionierung der Eltern-Elemente von `float` Elementen. <http://stackoverflow.com/questions>

Als Alternative zur Verwendung von `float` könnten Sie ein `inline-block` Element in Verbindung mit einer geeigneten `vertical-align` Property verwenden.

Scrolling

Verkleinern Sie die Breite des Browserfensters so, dass nur noch ein Ausschnitt der Ansicht zu sehen ist. Ihnen sollte jetzt die Möglichkeit gegeben sein, die Ansicht horizontal zu scrollen.

Durch welche Style-Property auf `<article>` wird dies erreicht?

Positionierung von Elementen

Wodurch wird die Positionierung des Toasts im unteren Bereich des Browserfensters herbeigeführt?

DOM Zugriff

Einen Überblick über die Ausdrucksmittel zur DOM-Manipulation erhalten Sie anhand einer Betrachtung aller Methoden im `demo.js` Skript und einer Vergegenwärtigung von deren Funktionalität anhand Ihrer Interaktion mit der Beispielanwendung.

Einblenden/Ausblenden

- Wie wird das Einblenden des „Toasts“ bzw. das Ausblenden der `<section>` Elemente in JavaScript veranlasst?
- Was ist auf Ebene von JavaScript erforderlich, um den „Toast“ nach einer gewissen Verweildauer wieder ausblenden zu können?

Bearbeitungszeit: 20 Minuten



Programmieren

Übung JSL-02**Umsetzung der Kopfzeile**

Setzen Sie die Kopfzeile wie nachfolgend dargestellt um. Den Pfeil „Zurück“ können Sie als „<<“ realisieren. Die Höhe der Statusbar (40px) brauchen Sie nicht berücksichtigen.

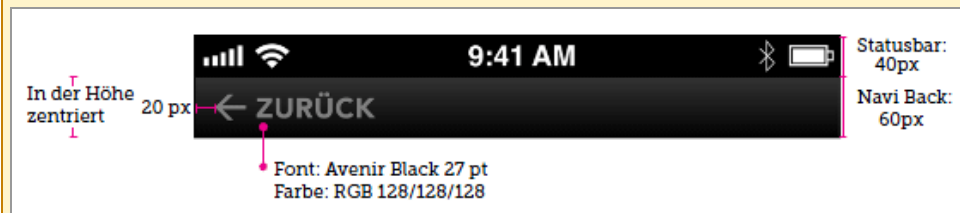


Abb.: Beispiel Kopfzeile

Anforderungen

1. Merkmale
 - (a) Fixe Höhe
 - (b) Hintergrund
 - (c) Beschriftung
 - (d) Die Kopfzeile soll ohne Zwischenraum bündig nach oben abschließen

Bearbeitungshinweise

Wählen Sie zur Realisierung des „Zurück“ Bedienelements ein geeignetes HTML-Element, z. B. `<a>` oder `<button>`.

Bearbeitungszeit: 20 Minuten



Programmieren

Übung JSL-03**Umsetzung der Detailansicht (4 Punkte)****Aufgabe**

Setzen Sie die nachfolgend beschriebene Detailansicht für Ihr Gestaltungselement „Textauszug“ so um, dass Gesamtansicht und Detailansicht dieselbe Dokumentstruktur verwenden und sich nur hinsichtlich der zugewiesenen Attribute unterscheiden. Orientieren Sie sich hinsichtlich der visuellen Gestaltung an den nachfolgend gezeigten Gestaltungsvorlagen.

Anforderungen

1. Merkmale
 - (a) Überschrift wird aus der Gesamtansicht übernommen (z. B. „Die Umsiedlerin“), die „Textauszug“ Überschrift ist nicht sichtbar.
 - (b) Hintergrund komplett weiß, kein Grau sichtbar.
 - (c) Der gesamte Text des „Textauszug“ Elements soll dargestellt werden.
 - (d) Kein horizontales Scrolling ggf. vertikales Scrolling
 - (e) Text soll an die Fensterbreite angepasst werden, d. h. die gesamte Breite mit Abständen entsprechend der Gestaltungsvorlage soll ausgenutzt und der Text soll nicht abgeschnitten werden.
 - (f) Kopfzeile soll bei vertikalem Scrolling fix bleiben. Oberhalb oder seitlich der Kopfzeile soll beim Scrollen kein Text sichtbar sein.

Bearbeitungshinweise

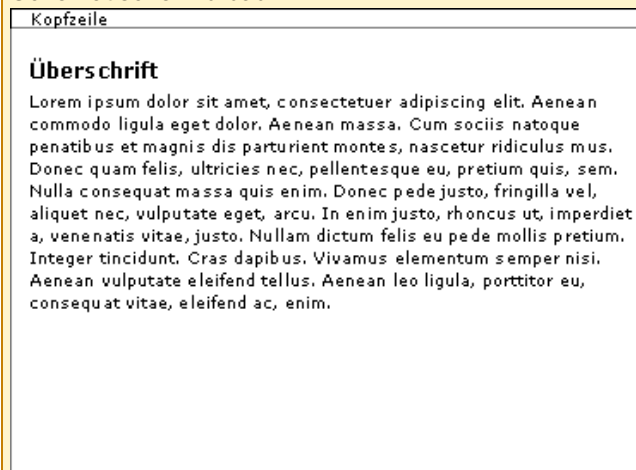
- Der Punkt bezüglich der Fixierung von Kopfzeile und Überschrift wird nur vergeben, wenn vertikales Scrolling wie beschrieben möglich ist.
- Verwenden Sie als Grundlage das in den zurückliegenden Übungen verwendete

HTML-Dokument, welches auch die anderen von Ihnen umzusetzenden Gestaltungselemente enthält.

- Versuchen Sie durch geeignete Verbindung von `class` und `id` Attributen sowie CSS-Regeln alle anderen Elemente auszublenden und nur „Textauszug“ als Detailansicht darzustellen. Je nach gewählter HTML-Struktur ist ggf. auch die Verwendung zusätzlicher `div` Elemente erforderlich.
- Wenn Sie die Attribute wieder entfernen / modifizieren, sollte ohne weitere Änderung am CSS wieder die Gesamtansicht angezeigt werden.
- Nachfolgend finden Sie zur weiteren Erläuterung einen schematischen Aufbau der Detailansicht, die Gestaltungsvorlage sowie ein Umsetzungsbeispiel.

Sonstiges

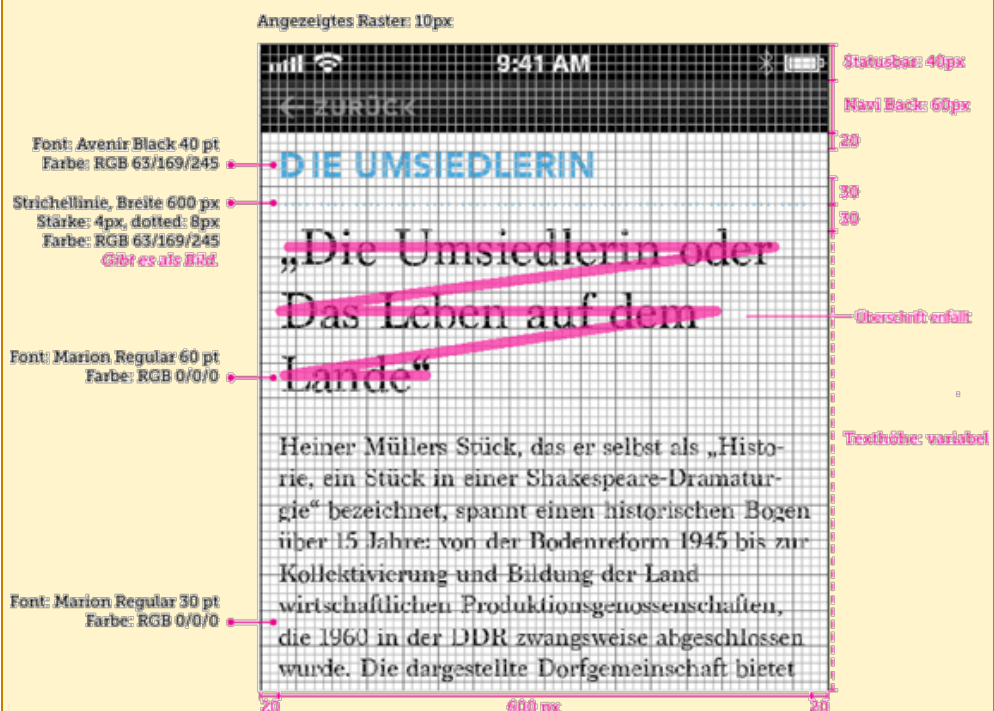
Schematischer Aufbau:



Kein horizontales Scrolling, sondern Anpassung der Breite an die Ausdehnung des Browserfensters. Ggf. vertikales Scrolling.

Gestaltungsvorlage

Setzen Sie entsprechend dieser Vorlage die Detailansicht zu „Textauszug“ um.



Umsetzungsbeispiel

<< ZURÜCK

DIE UMSIEDLERIN

14

Kate

Niet, Fondrak

FONDRAK Ich geh zum Amerikaner. Gehst Du mit?

NIET Ich bleib wo ich was krieg.

FONDRAK Scheiß drauf. Das düngt.

Pause

Bearbeitungszeit: 90 Minuten



Programmieren

Übung JSL-04

Ansichtsübergang

Aufgabe

Implementieren Sie in JavaScript den Übergang zwischen der Gesamtansicht und der von Ihnen in Übung JSL-03 entwickelten Detailansicht sowie den Übergang zurück in die Gesamtansicht.

Anforderungen

1. Die Übergänge sollen ohne Neuladen eines HTML-Dokuments, ohne Entfernung von Elementen aus dem DOM-Objekt und ohne Verwendung redundanter Inhalte umgesetzt werden.
2. Der Übergang in die Detailansicht wird durch Betätigung von „Weiter lesen“ in „Textauszug“ ausgelöst, die Rückkehr in die Gesamtansicht durch Betätigung von „Zurück“ auf der in Übung JSL-02 umgesetzten Kopfzeile.

Bearbeitungshinweise

Ihr Javascript sollte die in Übung JSL-03 manuell erstellten Zustände Ihres Dokuments für die Detailansicht und die Gesamtansicht durch geeignete Verwendung der DOM API herbeiführen.

Zu Anforderung 1: Redundant wäre z. B. die Verwendung zweier Elemente mit Texten unterschiedlicher Länge, um das „Abschneiden“ des Textauszugs in der Gesamtansicht gegenüber dem vollständigen Text in der Detailansicht zu realisieren.

Bearbeitungszeit: 60 Minuten



Programmieren

Übung JSL-05

Aus- und Einblenden (5 Punkte)

Aufgabe

Bauen Sie ein „Ausblenden/Einblenden“ der Ansichten in den in Übung JSL-04 umgesetzten Ansichtsübergang ein.

Anforderungen

1. Nach Auslösung eines Ansichtsübergangs durch den Nutzer soll zunächst die aktuelle Ansicht 2 Sekunden lang ausgeblendet werden. Danach soll die neue Ansicht 2 Sekunden lang eingeblendet werden.
2. Die Kopfzeile wird nicht durch das Ausblenden/Einblenden beeinflusst, d. h. sie soll dauerhaft unverändert sichtbar sein.

Bearbeitungshinweise

Für das Aus- und Einblenden können Sie die Style-Property `opacity` verwenden.

- `opacity: 0.0`: „vollkommen transparent“, d. h. unsichtbar
- `opacity: 1.0`: „vollkommen intransparent“, d. h. ohne durchschimmernden Hintergrund sichtbar

Ziehen Sie z. B. in Betracht, die Änderung der `opacity`-Property für das gemeinsame Wurzelement der Ansicht unterhalb der Kopfzeile umzusetzen, falls ein solches in Ihrer HTML-Struktur existiert.

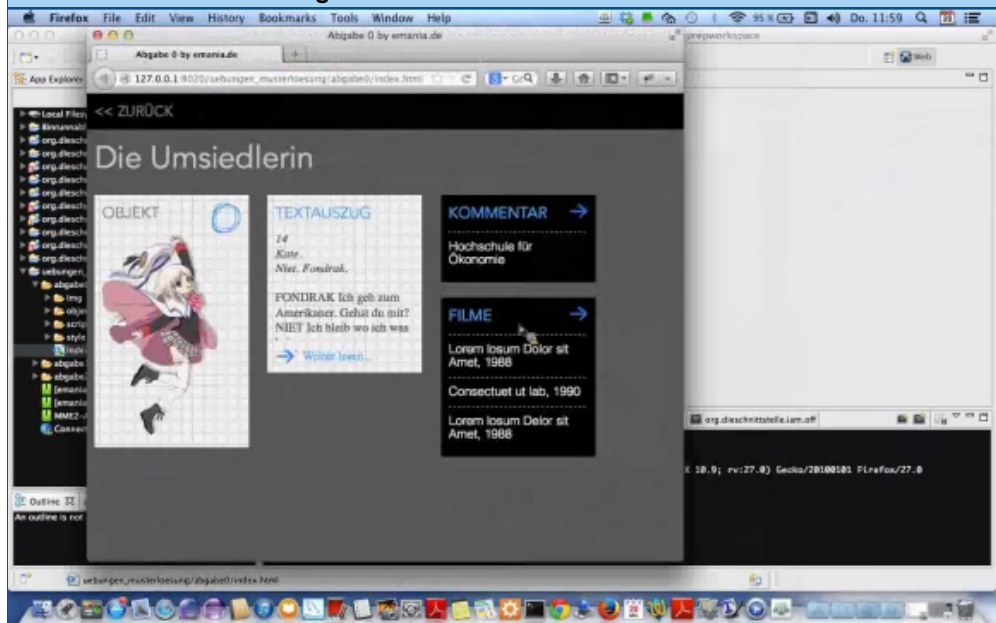
Den Ansichtswechsel nach Ausblenden der Gesamtansicht können Sie z. B. durch Setzen eines `transitionend` Event Listeners veranlassen. Eine Vorlage dafür finden Sie in der Umsetzung des „Toasts“ in den Implementierungsbeispielen.

Bearbeitungszeit: 60 Minuten



Film

Studentische Musterlösung



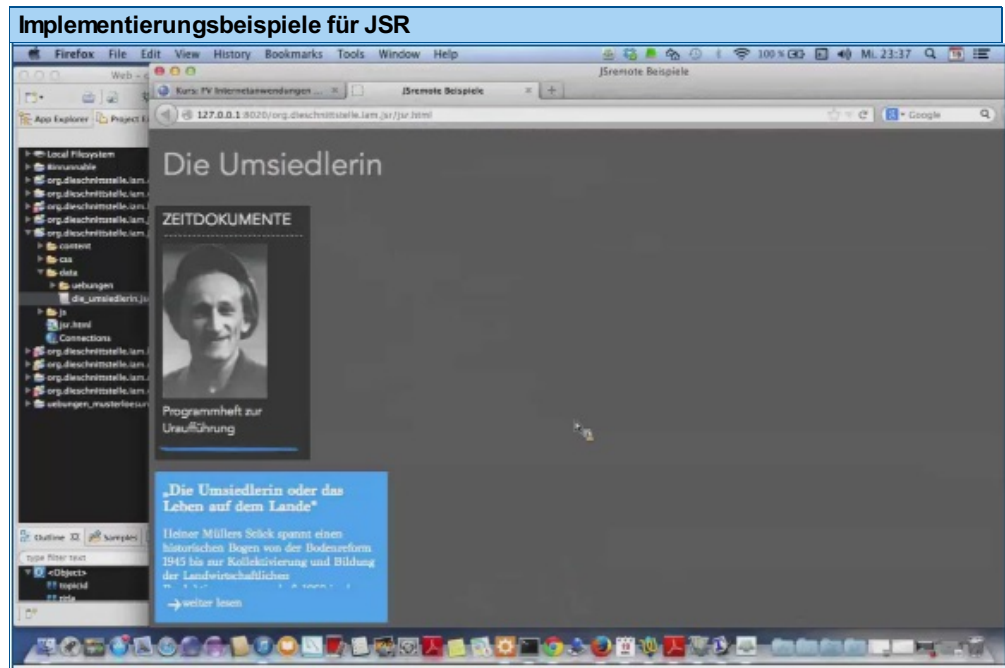
© Beuth Hochschule Berlin - Dauer: 02:12 Min. - Streaming Media 5 MB

Übungsteil JSR

Die folgenden Übungen JSR-01 und JSR-02 finden Sie im Projekt `org.dieschnittstelle.iam.jsr`. Sie zeigen den Zugriff auf serverseitige Ressourcen mittels `XMLHttpRequest`.



Film



© Beuth Hochschule Berlin - Dauer: 01:37 Min. - Streaming Media 4 MB




Übung JSR-01

Implementierungsbeispiele XMLHttpRequest und Ajax

Schauen Sie sich die Implementierungsbeispiele zu dieser Lerneinheit an und bearbeiten Sie dann die folgenden Aufgaben und Fragen.

XMLHttpRequest

- Vergegenwärtigen Sie sich die Funktionalität von `XMLHttpRequest` anhand der Implementierung der `xhr()` Funktion in der JavaScript-Datei `xhr.js` und ihrer Nutzung in `jsr.js`.
- Welche Aufgaben bei der Client-Server Interaktion übernimmt die `xhr()` Funktion, und welche werden durch die Nutzer dieser Funktion implementiert?
- Wie werden die nutzerspezifischen Aufgaben mit der durch `xhr()` implementierten Funktionalität zusammengeführt?
- Welche Medientypen werden dem Server als verarbeitbar durch den Client angezeigt, und weshalb ist die angegebene Typenmenge erforderlich? Was müsste geändert werden, um hier in Abhängigkeit vom Nutzungsfall eine gezielte Typangabe machen zu können?
- Verfolgen Sie die Logmeldungen bei Ausführung von `xhr()` in Firebug und versuchen Sie, den in der Abbildung dargestellten  clientseitigen Verarbeitungszyklus von HTTP-Requests und Responses anhand der Meldungen für `onreadystatechange` nachzuvollziehen. Weshalb passiert hier nur für den Fall, dass der „Ready State“ den Wert 4 hat, mehr als die Ausgabe einer Logmeldung?
- Welche Funktionalität aus der `onreadystatechange` Callback-Funktion könnte bereits für einen früheren „Ready State“ ausgeführt werden, und für welchen?
- Wann und wie wird ein JSON-Objekt im Request Body übermittelt? (davon machen die vorliegenden Beispiele noch keinen Gebrauch.)

AJAX

- Die create-Funktionen in `jsr.js` illustrieren die Verwendung der DOM API für den lesenden und schreibenden Zugriff auf das dargestellte HTML-Dokument.
- Kommentieren Sie aus der Methode `createVerknuepfungen()` die while Schleife aus und laden Sie das HTML-Dokument neu. Was ändert sich an der Darstellung und weshalb?
- Betrachten Sie das „Einführungstext“ Element im Firefox Inspector. Weshalb enthält das als `contentfragment` markierte `<div>` Element ein weiteres `<div>` Element?
- Halten Sie es für möglich, dass Performanceunterschiede bei der Verwendung von `querySelector()` / `querySelectorAll()` für den Zugriff auf Elemente bestehen, die alternativ durch `getElementsById()` oder `getElementsByClassName()` ausgelesen werden könnten? Schauen Sie auch hier nach
<http://jsperf.com/getelementsbyclassname-vs-queryselectorall>

Bearbeitungszeit: 60 Minuten



Übung JSR-02

Dynamischer Aufbau von Ansichten

Aufgabe

Bauen Sie das in Übung CSS-04 entwickelte Gesamtlayout dynamisch auf, analog zu den Implementierungsbeispielen.

Anforderungen

1. Die Konfigurationsdatei, die die Anordnung der Elemente beschreibt, soll in einer `onload` Funktion – wie in den Beispielen – vom Server geladen und dann für den

Aufbau der Ansicht verwendet verwendet werden.

2. Eine Wechsel der Konfiguration soll für eine ausgewählte Menge von Konfigurationsdateien ohne Eingriff in den Quellcode möglich sein. Verwenden Sie dafür ein Bedienelement, das Ihnen ein einfaches „Umschalten“ zwischen den Elementen einer Menge beliebiger Konfigurationsdateien ermöglicht. Die Ansicht soll jeweils auf Grundlage der aktuellen Konfiguration neu aufgebaut werden. Nichterfüllung dieser Anforderung hat eine pauschale Halbierung der insgesamt in JSR2 erzielten Punktzahl zur Folge.
3. Der Inhalt des „Textauszug“ Elements soll durch eine separate HTML-Datei bereitgestellt werden, die in der JSON-Beschreibung des Elements referenziert wird.
4. Die Konfiguration von „Objekt“ Elementen enthält eine Referenz auf das darzustellende Bild sowie ein `description` Attribut mit textuellem Inhalt. Dieser Text soll in einem `alert()` Dialog angezeigt werden, wenn der Nutzer auf den blauen Kringel in der rechten oberen Ecke des Elements klickt.
5. Es soll möglich sein, beliebig viele „Medienverweise“ Element anzuzeigen.
6. Die in blau dargestellte Überschrift des „Medienverweise“ Elements wird aus dem `title` Attribut des JSON-Objekts für das betreffende Element entnommen.
7. Die Elemente sollen in die im Attribut `renderContainer` angegebene Spalte im Spaltenlayout platziert werden (`left`, `middle` oder `right`)
8. Elemente, denen dieselbe Spalte zugewiesen ist, sollen entsprechend der Reihenfolge im JSON-Objekt untereinander angeordnet werden.
9. Elemente, die nicht in der Konfiguration angegeben sind, sollen nicht dargestellt werden.
10. Alle anderen Funktionen, die in den Übungen zu CSS umgesetzt wurden, inklusive des Wechsels in die Detailansicht, sollen weiterhin funktionsfähig sein.

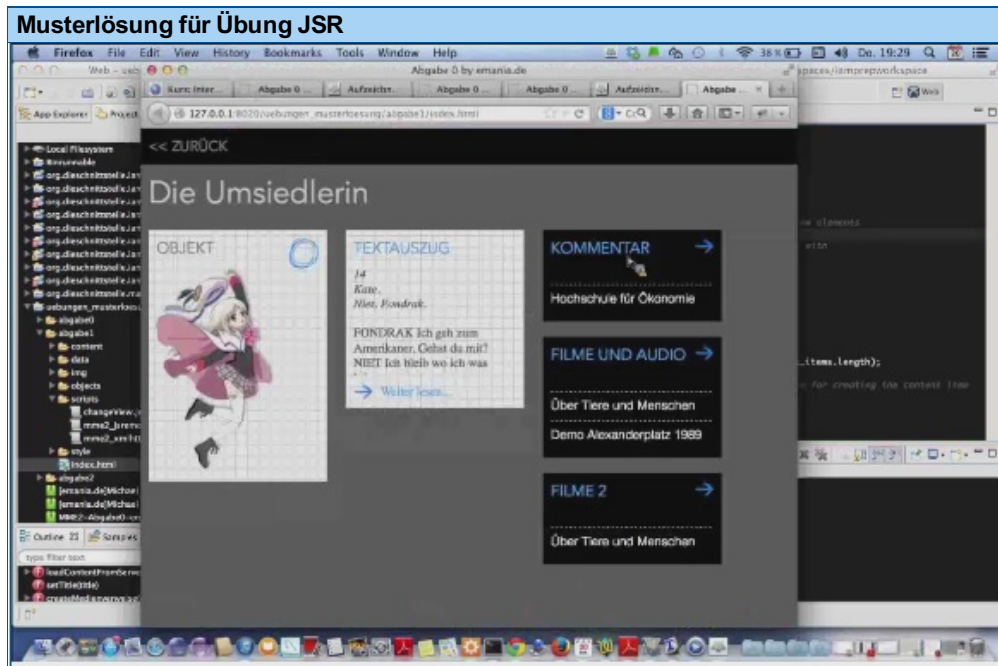
Bearbeitungshinweise

- Ihnen stehen – im Verzeichnis `data/uebungen` der Implementierungsbeispiele – drei verschiedene JSON-Konfigurationsdateien zur Verfügung, die die von Ihnen in den vorangegangenen Aufgaben bearbeiteten Elemente in verschiedener Anordnung enthalten. Bei der Abnahme wird überprüft, ob die Ansichten entsprechend der in der Konfiguration beschriebenen Anordnung aufgebaut werden können.
- Das Wechseln der Konfigurationsdatei zur Überprüfung der drei Varianten können Sie manuell umsetzen (d. h. im HTML-Dokument ersetzen + Neuladen), oder Sie bauen in Ihre Ansicht einen „Umschalt-Aktion“, bei der die Konfiguration nachgeladen wird und die Ansicht aktualisiert wird. Dafür könnten Sie z. B. die „Zurück“ Aktion in der Kopfzeile verwenden
- Übernehmen Sie diese Konfigurationsdateien sowie das `content` Verzeichnis komplett in Ihr eigenes Projekt.
- Die in den Implementierungsbeispielen enthaltenen JavaScript-Dateien können Sie in Ihrem eigenen Projekt wiederverwenden.
- Die in den Konfigurationsdateien referenzierten Bilder, Dokumente etc. können Sie durch Referenzen auf eigene Dateien ersetzen.
- **Zu Anforderung 2:** Das Laden der Inhalte für Textauszug aus einer separaten HTML-Datei können Sie analog zur Behandlung von „Einführungstext“ in den Implementierungsbeispielen umsetzen.

Bearbeitungszeit: 60 Minuten



Film

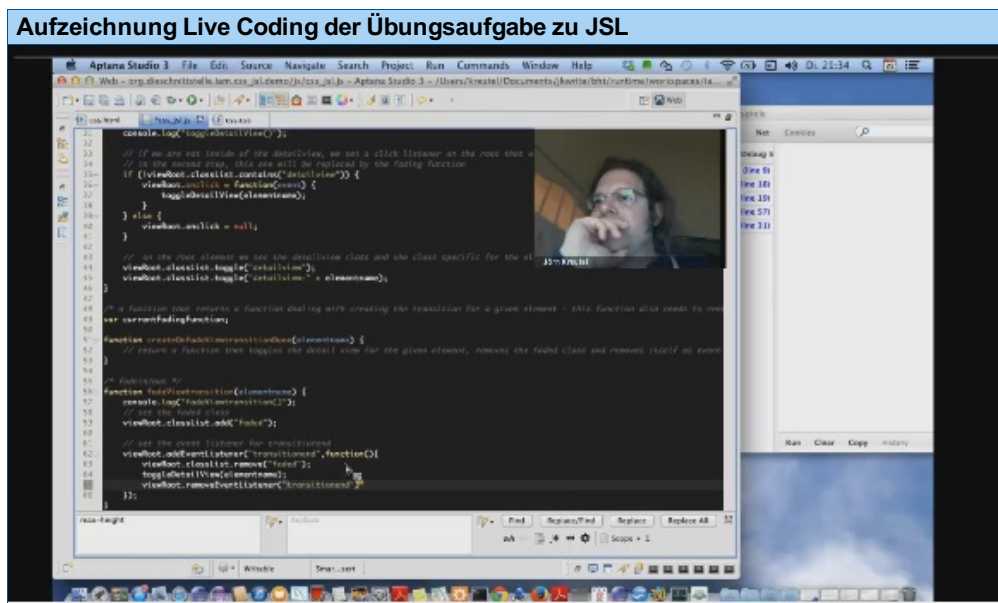


© Beuth Hochschule Berlin - Dauer: 03:20 Min. - Streaming Media 8 MB

Das folgende Video hat eine Länge von 61 Minuten. Der Autor erklärt darin das schrittweise Vorgehen der Übungen zu JSL und JSR



Film

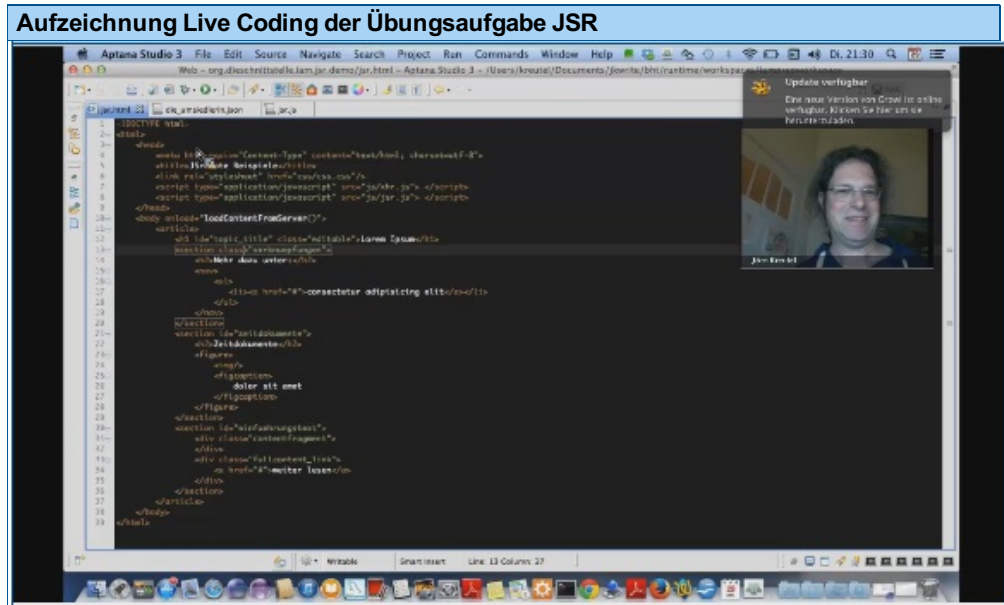


© Beuth Hochschule Berlin - Dauer: 63:30 Min. - Streaming Media 157 MB

Es folgt noch eine Aufzeichnung für die Übungen zu JavaScript Remote (JSR). Länge 46 Minuten.



Film



© Beuth Hochschule Berlin - Dauer: 46:39 Min. - Streaming Media 112 MB

Wissensüberprüfung

Versuchen die hier aufgeführten Fragen zu den Inhalten der Lerneinheit selbständig kurz zu beantworten, bzw. zu skizzieren. Wenn Sie eine Frage noch nicht beantworten können kehren Sie noch einmal auf die entsprechende Seite in der Lerneinheit zurück und versuchen sich die Lösung zu erarbeiten.



Formulieren

Übung JSL-06

Ausdrucksmittel von JavaScript

Versuchen die hier aufgeführten Fragen selbständig kurz zu beantworten, bzw. zu skizzieren.

1. Worin unterscheidet sich JavaScript als Sprache von HTML bzw. von CSS?
2. Was ist ECMAScript?
3. Nennen Sie fünf Statements, über die JavaScript verfügt.
4. Welche Operatoren verwendet JavaScript zur Überprüfung von Äquivalenz bzw. Identität zweier Werte?
5. Über welche Datentypen verfügt JavaScript?
6. Welche Funktion kann in JavaScript sowohl für das Entfernen, als auch für das Einfügen von Elementen in einen Array verwendet werden?
7. Notieren Sie zwei Formulierungen, mit denen Sie in JavaScript auf das Attribut attr eines Objekts obj zugreifen können.
8. Wie können Sie ein Attribut attr von einem Objekt obj entfernen?
9. Nennen Sie drei Aspekte, hinsichtlich derer JavaScript syntaktisch „liberaler“ ist als Java
10. Worin besteht ein dynamischer Aspekt bei der Handhabung von Datentypen in JavaScript?
11. Spielen in JavaScript Datentypen zur Laufzeit eine Rolle?
12. Müssen Sie sich in JavaScript darum sorgen, ob eine Variable einen nicht-null Wert zugewiesen hat?
13. Wie können Sie in einem if-Statement möglichst einfach überprüfen, ob eine Variable var einen Wert hat und weder undefined, noch null ist?
14. Wie lange sind die Werte globaler Variablen in JavaScript verfügbar?
15. Welche besondere Verwendungsmöglichkeit bietet Ihnen Java bezüglich Funktionen?
16. Wie können Sie eine Funktion, die Ihnen als Wert eines Arguments func übergeben wird, aufrufen?

Bearbeitungszeit: 30 Minuten



Formulieren

Übung JSL-07**Interaktionssteuerung mit JavaScript**

Versuchen die hier aufgeführten Fragen selbständig kurz zu beantworten, bzw. zu skizzieren.

1. Was gehört zur Interaktionssteuerung einer graphischen Nutzerschnittstelle?
2. Auf welche Weise kann eine graphische Nutzerschnittstelle einem Nutzer Feedback bezüglich einer Nutzereingabe geben?
3. Was unterscheidet Toasts von Dialogen?
4. Welche Rolle übernehmen DOM Events und die DOM API in Bezug auf den Verarbeitungszyklus einer MVC Architektur?
5. Was ist das „DOM Objekt“?
6. Wie können Sie in JavaScript auf das DOM Objekt zugreifen?
7. Nennen Sie 5 DOM Events
8. Wo können Sie Event Handler deklarieren, die das Auftreten von Ereignissen bezüglich der Elemente eines HTML Dokuments behandeln?
9. Welche besondere Möglichkeit bietet Ihnen die Zuweisung eines Event Handlers mittels der Methode `addEventListener()`?
10. Worin besteht die Capture Phase und worin Bubbling Phase im Verarbeitungsmodell für DOM Events?
11. Was ist ein event target?
12. Wie können Sie verhindern, dass ein DOM Event in der Bubbling Phase nicht nur vom unmittelbar angezielten Event Target, sondern zusätzlich auch von dessen Vorgängerelementen verarbeitet wird?
13. Welche beiden Funktionen stellt Ihnen JavaScript für die Darstellung modaler Dialoge zur Verfügung?
14. Welche beiden Möglichkeiten bietet Ihnen die DOM API, um aus einem Dokument alle `` Elemente auszulesen?
15. Worin unterscheiden sich die Methoden `querySelector()` und `querySelectorAll()`?
16. Wie können Sie in JavaScript ein neues ``-Element zu einer existierenden nicht-leeren ``-Liste hinzuzufügen?
17. Warum ist der Zugriff auf `innerHTML` nicht geeignet, um ein neues ``-Element zu einer existierenden nicht-leeren ``-Liste hinzuzufügen? Was können Sie stattdessen verwenden?
18. Wie können Sie dem class Attribut eines Elements element ohne Verwendung von String- Operationen einen neuen Wert hinzufügen?

Bearbeitungszeit: 30 Minuten



Formulieren

Übung JSL-08**DOM, JavaScript und CSS**

Versuchen die hier aufgeführten Fragen selbständig kurz zu beantworten, bzw. zu skizzieren.

1. Müssen Sie nach Änderung des DOM Objekts zur Neudarstellung der dargestellten Ansicht CSS aufrufen?
2. Welche Pseudoklassen stellt CSS zur Reaktion auf Nutzerinteraktion zur Verfügung?
3. Was wird durch CSS Transitions bewirkt?
4. Wie können Sie Ansichtselemente temporär aus einer dargestellten Ansicht „entfernen“?
5. Wie kann in CSS ein Ansichtselement aus einer Darstellung entfernt werden?

Bearbeitungszeit: 30 Minuten

**Übung JSL-09****XMLHttpRequest, JSON, Callbacks**

Versuchen die hier aufgeführten Fragen selbständig kurz zu beantworten, bzw. zu skizzieren.

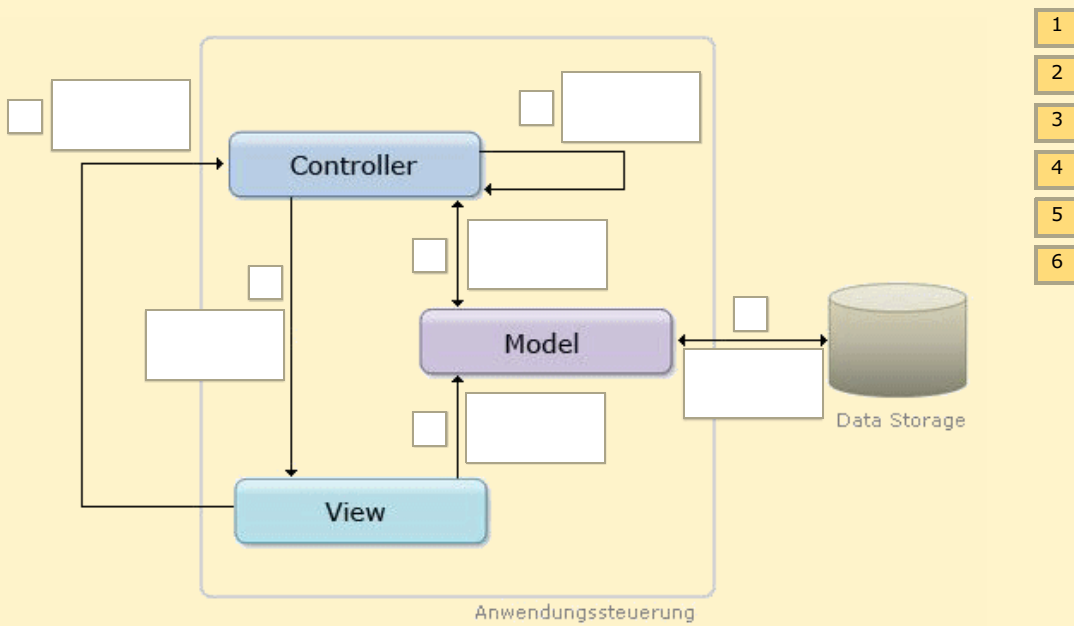
1. Wozu dient XMLHttpRequest in Web Applikationen?
2. Können HTTP Requests mit XMLHttpRequests auch synchron ausgeführt werden? Falls ja, auf welche Weise?
3. Bringen Sie die folgenden Methoden und -Attributzugriffe bezüglich eines XMLHttpRequest Objekts in eine sinnvolle Reihenfolge:
.status, .send(), .responseText, .setRequestHeader(), .open()
4. Wie können Sie auf HTML Inhalte zugreifen, die Ihnen ein Server als Response in einem XMLHttpRequest mit dem Response-Header Content-Type: text/html übermittelt?
5. Was ist JSON?
6. Handelt es sich hierbei um JSON: [{"name": "item 1", "id": 0}]? Begründen Sie Ihre Antwort.
7. Weshalb ist die Bezeichnung „Callback-Funktion“ sinnvoll?
8. Weshalb müssen Sie bei Aufruf asynchron ausgeführter Funktionen Callback-Funktionen anstelle von Rückgabewerten verwenden?
9. Formulieren Sie die Funktion f1 im folgenden Beispiel um unter der Annahme, dass f2 asynchron ausgeführt wird und sein Ergebnis einer Callback-Funktion als Argument übergibt. Dieser Wert soll dem Nutzer/Caller von f1 zur Verfügung gestellt werden:
function f1(a1) {
 var v1 = f2(a1);
 return v1;
}
10. Erläutern Sie den Begriff „AJAX“. Welche JavaScript APIs werden für die Umsetzung von „AJAX“ Funktionsmerkmalen verwendet?

Bearbeitungszeit: 30 Minuten

Übung JSL-10

MVC im Verarbeitungszyklus

Bitte ordnen Sie Reihenfolge-Nummer und Prozessvorgang den richtigen Feldern zu.



- | |
|--------------------------------|
| selects |
| displays |
| retrieves / updates / executes |
| validates / process data |
| reads / writes |
| requests processing |

?	Test wiederholen	Test auswerten
---	------------------	----------------