

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

MFM - Fortgeschrittene Aspekte von Formularen



Überblick und Lernziele

Nachdem wir in den vorangegangenen Abschnitten einige Grundlagen zur Verwendung von Formularen in Webapplikationen dargelegt haben, werden wir im folgenden als Vertiefung zunächst auf den für die Nutzung und Entwicklung von Formularen wichtigen Aspekt der „Gültigkeitsprüfung von Formulardaten“ und dann auf die „Gestaltung von Formularen“ mit CSS eingehen. Schließlich werden Sie mit „Multipart Formularen“ ein wichtiges Ausdrucksmittel kennenlernen, mittels dessen wir die Anwendungsfälle für Formulare über den Rahmen der bisher betrachteten Funktionen ausweiten können.



Lernziele

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

- Aspekte der Gültigkeit von Formularen zu unterscheiden.
- Zu erklären mittels welcher HTML-Attribute die Bedingungen für die Vollständigkeit von Formularen deklariert werden und welche Attribute zur Deklaration der Wohlgeformtheitsbedingungen verwendet werden.
- Den sinnvollen Einsatz von versteckten Felder zu beschreiben.
- Anwendungsspezifische Validierungsmaßnahmen mittels JavaScript zu entwerfen und eine Gültigkeitsprüfung durchzuführen.
- Die Möglichkeiten zur Übertragung von Multipart Formulardaten in HTML und JavaScript zu nennen.
- Gewöhnliche Formulardaten von Multipart-Formulardaten zu unterscheiden.
- Den Aufbau und Einsatz von Multipart Formularen zu beschreiben und beispielhaft praktisch anzuwenden.
- Die Verwendung eines unsichtbaren `<iframe>` Elements zur Umsetzung eines Callback-Mechanismus bezüglich des Rückgabewerts eines Multipart Requests zu erläutern.



Gliederung

Gliederung

- Formularvalidierung
- Gestaltung von Formularen
- Multipart Formulare
- Zusammenfassung
- Wissensüberprüfung
- Übungen



Zeitbedarf

Zeitbedarf und Umfang

Für die Bearbeitung der Lerneinheit benötigen Sie etwa 3 Stunden. Für die Bearbeitung der Übungen für die Beispielanwendung etwa 3 Stunden und für die Wissensfragen ca. 1,5 Stunden.

1 Formularvalidierung

Am Beginn unserer Beschäftigung mit Formularen hatten wir darauf hingewiesen, dass Formulare der strukturierten Formulierung von Anliegen zum Zweck des Informationszugriffs oder der Ausführung von Transaktionen dienen. Wir hatten außerdem bereits erwähnt, dass insbesondere unvollständige oder nicht korrekte Anliegensformulierungen bei der Bearbeitung von Formularen – erfolge sie manuell oder maschinell – detektiert werden müssen, wobei „Korrektheit“ sich auch auf die Erfüllung von Geschäftsregeln bezieht, die für die Ausführung einer Transaktion berücksichtigt werden müssen. Aus Perspektive der Formularnutzer hatten wir darauf verwiesen, dass diese möglichst früh Feedback bezüglich der Erfüllbarkeit ihrer Anliegen erhalten sollten – sei es um ihre Anliegensformulierung zu modifizieren oder das Anliegen selbst zu ändern oder aufzugeben. Verhindert werden sollte grundsätzlich, dass ein Nutzer länger als erforderlich im Unklaren bezüglich der Erfüllbarkeit eines Anliegens verbleibt, und dies schließt ausdrücklich die Zeit ein, die er mit dem Ausfüllen des Formulars verbringt.

Formulare und mobile UIs

Für Formulare in mobilen Nutzerschnittstellen ist letzterer Aspekt von besonderer Relevanz. Hier dürfte zumindest für manche Nutzer die Durchführung von Freitexteingaben auf einem Endgerät mit virtueller Tastatur eine höhere Barriere als bei Nutzung einer physikalischen Tastatur darstellen.

Freitexteingaben mit virtueller Tastatur

Diese Aussage gründet sich auf die subjektive Wahrnehmung des Autors, der sich der Generation der „digital immigrants“ zurechnet, die noch über ausgeprägte Erfahrung mit den Manifestationen der „Gutenberg Galaxis“ verfügt.

Aus diesem Grund gilt es mit Blick auf die Nutzerzufriedenheit grundsätzlich zu verhindern, dass ein Nutzer mehr Eingaben tätigt als unbedingt erforderlich. Dies betrifft umso mehr Eingaben, die sich aufgrund einer Validierungsproblematik als obsolet erweisen, welche bereits zuvor hätte detektiert werden können. Sollte z. B. bei einem Antrag auf Nachwuchsförderung ein Nutzer bereits auf Grundlage seines Geburtsdatums aus dem Kreis der Antragsberechtigten herausfallen, sollte es diesem Nutzer erspart bleiben, seine Motivation für den Antrag in Form von Freitext darzulegen.

Gültigkeitsaspekte

Die Gültigkeitsprüfung von Anliegensformulierungen ist also ein wichtiger Bestandteil insbesondere auch der maschinellen Verarbeitung von Formulardaten. Diesbezüglich sind bei den folgenden Gültigkeitsaspekten voneinander zu unterscheiden:


- **Vollständigkeit:** sind Werte für alle erforderlichen Felder vorhanden und sind die für jedes Feld eingegebenen Werte jeweils vollständig?
- **Wohlgeformtheit:** sind die eingegebenen Werte syntaktisch korrekt d. h. erfüllen Sie die Voraussetzungen für die maschinelle Verarbeitung?
- **Fachliche Gültigkeit:** entsprechen die eingegebenen Werte einzeln und insgesamt den Regeln für die betreffende Transaktion bzw. kann ein Informationszugriff auf Grundlage der Werte überhaupt erfolgreich sein?

Am Beispiel von Datumsangaben als einem häufig für Formulare erforderlichen Informationsbestandteil entsprechen diese Aspekte zum einen der Prüfung, ob eine erforderliche Angabe vorhanden ist und alle erforderlichen Dimensionen – z. B. eine ggf. notwendige Angabe des Jahres – enthält (Vollständigkeit), ob eine Angabe den verarbeitbaren Mustern für Datumsangaben entspricht (Wohlgeformtheit) und ob das angegebene Datum für das betreffende Formularfeld gültig ist. So können z. B. Anforderungen bezüglich der Zukünftigkeit oder Vergangenheit sowie der Mindest- oder Maximalabstände des Datums in Relation zum Zeitpunkt der Anliegensformulierung oder zu anderen Datumsangaben existieren.

Was nun die Verwendung von HTML angeht, so existieren hier Ausdrucksmittel, mittels derer sowohl der Aspekt der Vollständigkeit, als auch die Wohlgeformtheit von Formulardaten in gewissem Umfang deklarativ auf Ebene der Markupsprache behandelt werden können. Die Behandlung schließt hier sowohl die Prüfung des betreffenden Aspekts als auch das Verhalten im Fehlerfall ein, welches einerseits einen Hinweis auf den Fehler beinhalten sollte, andererseits aber auch verhindern sollte, dass eine Weiterbearbeitung der Formulardaten erfolgt.

Nicht deklarativ behandelt werden können jedoch die meisten Aspekte der fachlichen Gültigkeit einzelner Formularfeldwerte im vorgenannten Sinne, wie auch die Gültigkeit mehrerer voneinander abhängiger Werte, inklusive der „abhängigen Erfordernis“ von Feldern, bei der die Werte anderer Felder berücksichtigt werden müssen. Beispielsweise ist die Angabe eines Rückreisedatums in Buchungsanwendungen nur erforderlich, falls tatsächlich eine eine Rückreise gebucht werden soll... was für uns möglicherweise trivial klingt, erfordert für die maschinelle Formularbearbeitung Implementierungsaufwände, die bei Verwendung von HTML die Entwicklung von JavaScript Code beinhalten.

Nachfolgend werden wir auf die hier genannten Aspekte der Validierung eingehen und die jeweils verfügbaren Ausdrucksmittel beschreiben. Alternative Darstellungen der Validierung in HTML5 finden sich u. a. im Internet unter:

 <http://www.html5rocks.com>

1.1 Deklarative Validierung

Erfordernis

Das boolesche Attribut `required` zur Kennzeichnung der Erfordernis eines Feldwertes hatten Sie bereits kennengelernt.

Gesetzt werden kann dieses für die meisten der vorgestellten Typen von `<input>` Elementen sowie für `<textarea>`. Zur Darstellung von `<select>` Elementen zeigt ein Browser immer eine der darin enthaltenen Optionen an – bei Nichtvorhandensein einer als `selected` markierten Option ist dies der Wert des ersten `<option>` Elements innerhalb des `<select>` Elements. Falls dieses `<option>` Element jedoch leer ist oder falls es ein leeres `value` Attribut hat, kann mittels `required` zum Ausdruck gebracht werden, dass die Auswahl einer anderen Option erforderlich ist, die dem betreffenden Formularfeld einen nicht-leeren Wert zuweist. Siehe folgendes Beispiel:



Quellcode

Liste von Optionen mit Platzhalter

```
001 <!-- Liste von Optionen mit Platzhalter -->
002 <select name="elementType" required="required">
003   <option value="">select...</option>
004   <option value="zeitdokumente">Zeitdokumente</option>
005   <option value="einfuehrungstext">Einführungstext</option>
006   <option value="textauszug" disabled="disabled">Textauszug</option>
007   <option value="objekt">Objekt</option>
008 </select>
```

Für Checkboxes als reine Binärauswahlelemente kann zwar ebenfalls immer ein Wert ermittelt werden, auch hier tritt aber bei Markierung einer Checkbox als `required` ein Validierungsfehler auf, falls diese nicht ausgewählt ist. Auf diese Weise ist es möglich, Formularfelder, bezüglich derer der Nutzer zwar „keine Wahl“ hat, für die jedoch explizit seine Zustimmung oder Zurkenntnisnahme gewünscht oder erforderlich ist, mittels `required` zu behandeln – ein Beispiel hierfür ist z. B. die im Rahmen von verbindlichen Kaufabschlüssen übliche Zustimmung zu den allgemeinen Geschäftsbedingungen des Verkäufers.

Soll schließlich ein Formularfeld, dessen mögliche Werte durch eine Menge von `<input type="radio">` Elementen beschrieben werden, als erforderlich markiert werden, reicht es aus, dass das Attribut auf einem einzigen Element der Menge gesetzt wird. Alternativ kann aber auch einer der möglichen Werte durch Vollständigkeit Setzung des booleschen `checked` Attributs vorausgewählt werden.

Vollständigkeit

Die Überprüfung der Vollständigkeit findet bei Ausführung der `submit` Aktion statt, bevor ein etwaiger `onsubmit` Event Handler aufgerufen wird. Letztere Reihenfolge ist für uns insofern von großer Bedeutung, als in der vorgestellten „Web 2.0“ Behandlung von Formularen die Formulardaten in einem solchen Event Handler an die serverseitige Anwendung übermittelt werden. Wird bei der Überprüfung der Formulardaten festgestellt, dass für ein als `required` gekennzeichnetes Feld kein Wert vorliegt und haben Sie selbst keine spezifische Darstellung für diesen Fall vorgesehen, dann generiert ein Desktop-Browser üblicherweise eine Fehlermeldung wie die folgende:

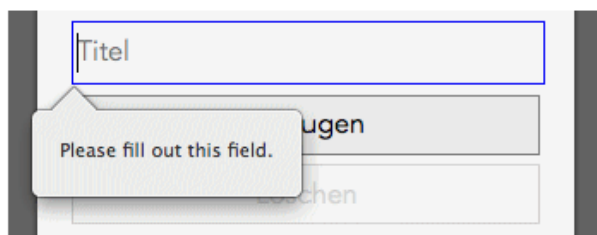


Abb.: `required` Textfeld

Das Verhalten mobiler Browser kann hiervon abweichen. So nahm eine frühere Version von Firefox auf Android z. B. eine Fokussierung auf das betreffende Feld vor und öffnet daran anschließend die Tastatur – ein expliziter Hinweis auf die Erfordernis der Eingabe erging jedoch nicht. Die Darstellung in der aktuellen (Beta-?)Version entspricht hingegen abgesehen von Unterschieden bei der Formatierung der „Sprechblase“ dem Verhalten des Desktop-Browsers.

Wie Sie für Fehlermeldungen eine eigene Gestaltungsvorlage verwenden können, wird weiter unten gezeigt.

Wohlgeformtheit

Das `type` Attribut für `input` hatten wir oben bereits als Ausdrucksmittel zur Kennzeichnung des Datentyps eines Eingabefeldes eingeführt. Für den Fall, dass ein Browser für einen der in der HTML-Spezifikation definierten Typen eine Texteingabe erfordert – z. B. weil hierfür kein spezifisches Bedienelement zur Eingabe bereitgestellt wird – und falls für den betreffenden Typen ein standardisiertes Textformat existiert, wird bei „Abschluss der Eingabe“ durch den Browser überprüft, ob der eingegebene Text den Anforderungen des Formats entspricht.

Spätestens bei Ausführung der Submit-Aktion wird dann im Fehlerfall ein Hinweis eingeblendet, der auf den erforderlichen Typ verweist. Diesbezüglich ist aber zu beachten, dass z. B. für das Vorliegen einer URL das Vorhandensein eines „.“ zur Kennzeichnung des URL-Schemas ausreichend ist. Lassen sich die Bedingungen an die einzugebenden Werte daher präziser formulieren, kann daher alternativ das gewünschte Textformat in Form eines regulären Ausdrucks formuliert werden, der als Wert eines `pattern` Attributs gesetzt wird.

Im folgenden Beispiel muss eine Texteingabe z. B. mit `https://` beginnen und danach mindestens ein (nicht leeres) Zeichen sowie einen „.“ gefolgt von mindestens zwei Zeichen enthalten:

```
<input name="fieldname" pattern="https://.{1,}\..{2,}"
title="You need to use https URLs!"/>
```

Die Angabe eines `title` Attributs erlaubt uns, einen Hinweistext anzugeben, der im Fall einer fehlerhaften Eingabe entsprechend der durch den Browser vorgesehenen Visualisierung dargestellt wird:

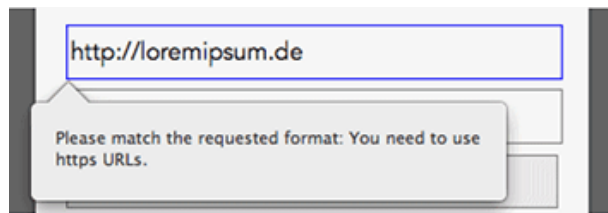


Abb.: Hinweistext bei fehlerhafter Eingabe

disabled und readonly

Von Gültigkeitsprüfungen ausgenommen sind Felder, die zwar sichtbar, aber nicht bedienbar sind. Diese können mit dem Attribut `disabled` markiert werden. Insbesondere für die Behandlung von Feldern, deren Eingabe – sei sie erforderlich oder optional – von der Befüllung anderer Felder abhängig ist, erscheint das Attribut sehr gut geeignet. Ob ein mittels `disabled` deaktiviertes Feld als „ausgegraut“ angezeigt werden soll oder vollständig ausgeblendet werden soll, kann dann auf der Ebene von CSS entschieden werden.

Erwähnt sei jedoch, dass als `disabled` markierte Felder bei Browser-gesteuerter Übermittlung der Formulardaten an eine angegebene `action`-URL ignoriert werden. Für die Darstellung von Feldern mit unveränderbaren Werten erscheint `disabled` in diesem Rahmen daher nicht geeignet. Falls Sie die Bearbeitung der Formulardaten aber selbst in einem Submit-Handler implementieren, gilt hier unsere Aussage von oben: Sie sind selbst dafür verantwortlich, die zu übermittelnden Daten „einzusammeln“, d. h. es hindert sie niemand daran, auch die Werte deaktivierter Felder zu verwenden.

Um auch in maschinenlesbarer Form auszudrücken, dass der Wert eines Feldes zwar nicht editierbar ist, aber durchaus zu den zu verarbeitenden Formulardaten gehört, können Sie das Attribut `readonly` anstelle von `disabled` verwenden.

readonly

Ein Gegenstück zur Markierung eines Feldes als `disabled` stellt in der ursprünglichen Behandlung von Formularen der Typ `hidden` für `<input>` dar. Entsprechend markierte Felder werden durch den Browser gar nicht angezeigt, ihr Wert wird jedoch den zu übermittelnden Formulardaten hinzugefügt. Bei Verwendung von Submit-Handlern für solche Daten steht es Ihnen freilich offen, ob Sie hierfür tatsächlich ein „verstecktes“ Feld nutzen oder ob Sie die Daten in anderer Form, z. B. Variablen in Ihrem Skript, für die Behandlung des Formulars verfügbar machen.

1.2 Anwendungsspezifische Validierung

Validierungsfehler

Anwendungsspezifische Validierungsmaßnahmen können sich einerseits darauf beziehen, dass die durch den Browser umgesetzte verallgemeinerte Behandlung generischer Validierungsfehler – z. B. bezüglich `required`, `type` oder `pattern` – für bestimmte Fälle oder generell modifiziert werden soll. Andererseits können solche Maßnahmen die Umsetzung anwendungsspezifischer Validierungsprüfungen beinhalten, für welche im Rahmen von HTML und JavaScript noch keine deklarativen Ausdrucksmittel zur Verfügung stehen.

Für den ersten Fall ist es notwendig, dass Sie als Entwickler auf das Auftreten von Validierungsfehlern reagieren können. Um eigene Validierungsprüfungen durchzuführen, müssen Sie entscheiden, unter welchen Umständen die Validierung durchgeführt werden soll, müssen dann überlegen, wie Sie die betreffenden Umstände erkennen können und auf welche Weise für den Fall von Validierungsfehlern das Feedback an den Nutzer Ihrer Anwendung kommuniziert werden soll. Nachfolgend werden Sie erfahren, wie beide Anforderungen mittels der Ausdrucksmittel von JavaScript eingelöst werden können.

invalid Ereignis

Grundlage für den Umgang mit Validierungsfunktionen auf der Ebene von JavaScript ist ein Ereignis namens `invalid`, das zum aktuellen Repertoire von DOM Events gehört. Dieses Ereignis wird einerseits durch den Browser ausgelöst, falls auf Grundlage validierungsbezogener Attribute die Validierungsprüfung bezüglich eines Formulars oder Formularfelds fehlschlägt. Zum anderen können Sie selbst das Ereignis auslösen, falls Sie im Zuge einer anwendungsspezifischen Gültigkeitsprüfung ein Problem detektieren. Zur Behandlung des Ereignisses müssen Sie lediglich – wir für alle Ereignisse, an denen Sie interessiert sind – einen Event Handler deklarieren, der auf das Auftreten des Ereignisses reagiert. Diesen Event Handler weisen Sie den Formularfeldern zu, bezüglich welcher Sie eine anwendungsspezifische Behandlung von Validierungsfehlern durchführen wollen, z. B. dem durch die Variable `inputSrc` referenzierten Feld im folgenden Beispiel:



Quellcode

Zuweisung eines Event Handlers

```
001 // Zuweisung eines Event Handlers für invalid
002 inputSrc.oninvalid = function(event) {
003     // Unterbinde die Default-Behandlung
004     event.preventDefault();
005
006     // Handle das Ereignis
007     /* (...) */
008 }
```

Hier nehmen wir an, dass unsere eigene Ereignisbehandlung anstelle der durch den Browser vorgesehenen Default-Behandlung ausgeführt werden soll, welche z. B. in der Darstellung einer der oben gezeigten Hinweismeldungen resultieren würde. Zu diesem Zweck rufen wir die Funktion `preventDefault()` auf dem Ereignisobjekt auf. Falls gewünscht, können Sie aber selbstverständlich auch die Default-Behandlung um eine eigene Behandlung *ergänzen* und würden dann auf diesen Aufruf verzichten.

ValidityState

Nun haben wir aber oben gesehen, dass verschiedene Gültigkeitsfehler auftreten können, beispielsweise das Nichtvorliegen eines Wertes für ein als `required` gekennzeichnetes Feld vs. das Vorliegen eines ungültigen Wertes. Um diese unterscheiden zu können, steht uns auf den Feldern eines Formulars ein Attribut namens `validity` zur Verfügung, dessen Wert ein Objekt des Typs `ValidityState` ist.

„Objekt des Typs T“

Die Formulierung „ein Objekt des Typs T“ ist in Bezug auf JavaScript zu lesen als „ein Objekt, welches das in T definierte Interface bereitstellt“.

`ValidityState` verfügt über eine Reihe selbsterklärend benannter boolescher Attribute anhand derer der jeweils vorliegende Gültigkeitsfehler ermittelt werden kann, z. B. `valueMissing`, `typeMismatch` oder `patternMismatch` als Ausdruck einer Verletzung der mittels der Attribute `required`, `type` bzw. `pattern` ausgedrückten Gültigkeitsbedingungen. Für eine vollständige Dokumentation verweisen wir auf die [www](#) Schnittstellendefinition von `ValidityState`. Falls keine Gültigkeitsproblematik bezüglich eines Feldes vorliegt, hat das Attribut `valid` auf `ValidityState` den Wert `true`. Auf dieser Grundlage kann der Browser nach Durchführung aller Validierungsprüfungen entscheiden, ob bei Betätigung eines Submit-Buttons tatsächlich das `submit` Ereignis und ggf. die Übermittlung der Formulardaten ausgelöst werden soll oder nicht.

Im obigen Fall können beispielsweise die beiden Fälle des Nichtvorhandenseins eines Wertes und des Vorhandenseins eines ungültigen Wertes wie folgt unterschieden werden. Beachten Sie hier, dass – wie für anderer DOM-Ereignisse auch – das Element, bezüglich dessen das betreffende Ereignis ausgelöst wurde, als Wert des Attributs `target` auf dem `event` Objekt gegeben ist:



Quellcode

Deklaration eines anwendungsspezifischen Event Handlers

```
001 // Deklaration eines anwendungsspezifischen Event Handlers für invalid
002 function onInvalid(event) {
003     // Unterbinde die Default-Behandlung
004     event.preventDefault();
005
006     // ermittle den Namen des Feldes
007     var fieldname = event.target.name;
008     // lies den ValidityState des Eingabeelements aus
009     var valstate = event.target.validity;
010
011     // Handle das Ereignis in Abhängigkeit vom aufgetretenen Problem
012     if (valstate.valueMissing) {
013         alert("Für " + fieldname + " muss ein Wert eingegeben werden!");
014     }
015     else if (valstate.typeMismatch || valstate.patternMismatch) {
016         alert("Der eingegebene Wert für " + fieldname + " ist ungültig!");
017     }
018     else {
019         alert("Der Wert für " + fieldname + " konnte nicht erfolgreich validiert
020             werden: " + event.target.validationMessage);
021     }
022 }
```

`validationMessage`

Im abschließenden `else` Fall sehen Sie, dass wir auf das Feld `validationMessage` des betreffenden Eingabeelements zugreifen, das ggf. eine durch den Browser gesetzte Fehlermeldung enthält. Durch Zugriff auf dieses Feld können Sie also z. B. die im vorangegangenen Abschnitt gezeigten Fehlermeldungen anwendungsspezifisch realisieren, z. B. wird im folgenden Formular hierfür eine im Erfolgsfall leere „Kopfleiste“ verwendet. Zusätzlich wird das fehlerhafte Feld durch eine rote Umrandung markiert – auf die Umsetzung hiervon bzw. Gestaltungsalternativen gehen wir im nachfolgenden Abschnitt ein:

Abb.: Fehlermeldung für Eingabefeld

Falls Sie nicht nur auf die Validierungsmaßnahmen des Browsers reagieren möchten, sondern anwendungsspezifische Gültigkeitsprüfungen durchführen wollen, können Sie im Fehlerfall mittels Aufrufs der Funktion `setCustomValidity()` auf dem überprüften Feld bzw. den in die Problematik involvierten Feldern eine eigene Fehlermeldung setzen. Damit wird dann sowohl die als String übergebene Meldung als Wert des `validationMessage` Attributs des Formularfelds gesetzt, als auch der Wert `true` für das Feld `customError` auf dem `ValidityState` Attribut des Feldes. Alleine der Aufruf von `setCustomValidity()` löst jedoch noch kein `invalid` Ereignis aus. Dafür ist zusätzlich der Aufruf der Funktion `checkValidity()` auf dem entsprechenden Feld bzw. den überprüften Feldern erforderlich. Die Implementierungsbeispiele bieten z. B. eine Auswahl an Optionen via Radiobuttons an, ohne eine Vorauswahl durchzuführen. Im Rahmen der Behandlung des `submit`-Ereignisses wird dann geprüft, ob einer der Buttons ausgewählt wurde oder nicht und ggf. eine Fehlermeldung ausgelöst:



Quellcode

Submit-Handlers mit Gültigkeitsprüfung

```

001 // Verwendung eines Submit-Handlers mit Gültigkeitsprüfung
002 document.forms["einfuehrungstextForm"].onsubmit = function(event) {
003     // überprüfe, ob eine Alternative ausgewählt wurde
004     var checkedButton = document.querySelector("input[name='contentMode']:
005         checked");
006     if (!checkedButton) {
007         // setze eine Fehlermeldung auf einem der Buttons
008         inputContentModeFirst.setCustomValidity("Eine Eingabeart muss ausgewählt
009             werden!");
010         // löse das invalid-Ereignis aus
011         inputContentModeFirst.checkValidity();
012     }
013     else {
014         // übermittle die Formulardaten
015         /* (...) */
016     }
017
018     // unterbrich die Submit-Ausführung bezüglich der action-URL
019     return false;
020 }

```

Die Gültigkeitsprüfung, das Setzen der Fehlermeldung und das Auslösen des `invalid` Ereignisses wird bei der hier gezeigten Umsetzung getrennt von der Reaktion auf das Ereignis und der weiteren Darstellung der Meldung. Auf diese Weise könnte z. B. auf dem Eingabeelement `inputContentModeFirst` ein Event Handler für `invalid` gesetzt werden, der für die gesamte Ereignisbehandlung bezüglich `invalid` zuständig ist – unabhängig davon, ob die Auslösung der Ereignisse anwendungsspezifisch erfolgt oder durch die Default-Validierung des Browsers ausgelöst wird. Bei Nutzung der bereits oben gezeigten Kopfzeile oberhalb der Eingabefelder könnte die Fehlermeldung z. B. wie folgt dargestellt werden:

Abb.: Fehlermeldung für Eingabefeld

Eingabeereignisse

Im zuvor gezeigten Beispiel haben wir die anwendungsspezifische Gültigkeitsprüfung innerhalb des Event Handlers für das `submit` Ereignis bezüglich des gesamten Formulars umgesetzt. Da für Radiobuttons die Deklaration `required` nicht zur Verfügung steht, kann auf diese Weise die Vollständigkeit der zu bearbeitenden Formulardaten verifiziert werden. Soll eine Prüfung aber bereits zu einem früheren Zeitpunkt durchgeführt werden, müssen Sie zunächst überlegen, welche Ereignisse bezüglich welcher Eingabeelemente geeignet sind, um eine Überprüfung zu initiieren. Zur Umsetzung können Sie dann für diese Ereignisse Event Handler deklarieren, die das gewünschte Verhalten implementieren. Beispielsweise

- sind für klickbare/tapbare Eingabeelemente wie Radiobuttons und Check Boxen `onclick` Handler geeignet.
- kann auf Tastaturangaben in Textfeldern durch Angabe eines `oninput` Handlers reagiert werden, welcher für jeden Tastendruck aufgerufen wird.
- wird bei Änderung der ausgewählten Option eines `<select>` Elements ein `onchange` Ereignis ausgelöst, für welches ein entsprechender Handler gesetzt werden kann. Dieses Ereignis steht auch für `<input>` Element zur Verfügung, wird aber im Gegensatz zu `oninput` erst beim Verlassen des Feldes ausgelöst, d. h. bei Fokussierung eines anderen Eingabefeldes oder bei der Betätigung eines Buttons

Für HTML5 wurde darüber hinaus auch die Einführung zweier Ereignisse `forminput` und `formchange` in Erwägung gezogen, um auf einem Feld Eingabeereignisse bezüglich anderer Felder eines Formulars zu detektieren. Der betreffende Vorschlag wurde jedoch [www mit Verweis darauf](#), dass etwaige anwendungsspezifische Funktionalität bezüglich dieser Ereignisse auch auf Grundlage der o. g. elementaren Ereignistypen umsetzbar ist, wieder [www](#) fallen gelassen.

Verwendung von Eingabeereignissen

Mit Blick auf die zu Beginn unserer Beschäftigung mit Formularen beschriebenen Anforderungen an die Funktionalität von Formularen sei darauf verwiesen, dass Eingabeereignisse nicht nur als Auslöser anwendungsspezifischer Validierungsmaßnahmen verwendet werden können. Mit Blick auf validierungsbezogene Fehlermeldungen sind sie außerdem geeignet, um eine Fehlermeldung für den Fall einer Eingabe wieder auszublenzen. So wird für die oben gezeigte Kopfzeile für Fehlermeldungen – hier referenziert durch `outputValidationMessages` – beispielsweise durch den folgenden Event Handler bei Eingabe eines Zeichens in ein Texteingabefeld zurückgesetzt:



Quellcode

Zurücksetzen der Kopfzeile zur Anzeige von Fehlermeldungen

```
001 // Zurücksetzen der Kopfzeile zur Anzeige von Fehlermeldungen
002 inputTxt.oninput = function(event) {
003     if (outputValidationMessages.innerHTML != "") {
004         outputValidationMessages.innerHTML = "";
005     }
006 }
```

Eingabeereignisse sind außerdem sehr gut geeignet, um Abhängigkeiten zwischen Formularfeldern in der Darstellung des Formulars zu reflektieren, beispielsweise können Werte für `required` oder `disabled` als Reaktion auf Ereignisse bezüglich vorhandener Auswahlelemente gesetzt werden. So wechselt z. B. der nachfolgend auf einer Menge von Radiobuttons gezeigte Event Handler u. a. den Aktivierungszustand bezüglich zweier alternativer Eingabefelder aus. Ein entsprechendes Auswechseln bezüglich des `required` Attributs ist hier nicht erforderlich, da deaktivierte Felder bei der durch `submit` ausgelösten Formularvalidierung durch den Browser ignoriert werden. Das Beispiel zeigt außerdem ein Beispiel für den Zugriff auf eine Menge von Radiobuttons, die hier als Wert des Formularfelds `contentMode` verfügbar ist:



Quellcode

Event Handler zum Umschalten

```

001 // Event Handler zum Umschalten der Aktivierung von inputSrc und inputTxt
002 var radioOnClickListener = function(event) {
003     /* (...) */
004
005     // überprüfe, welche Option ausgewählt wurde
006     if (event.target.id == "contentMode_txt") {
007         inputSrc.disabled = true;
008         inputTxt.disabled = false;
009     } else {
010         inputSrc.disabled = false;
011         inputTxt.disabled = true;
012     }
013 };
014
015 // füge den Handler den Radiobuttons hinzu
016 var radioButtons = document.forms["form_einfuehrungstext"].contentMode;
017 for (var i = 0; i < radioButtons.length; i++) {
018     radioButtons[i].onclick = radioOnClickListener;
019 }

```

Anhand der hier getroffenen Fallunterscheidung wird dem Nutzer dann wahlweise eine Freitexteingabe oder das Hochladen eines Dateiinhalts angeboten:

Abb.: Fallunterscheidung

Mit der konkreten Umsetzung der zweiten, rechts dargestellten, Alternative werden wir uns unten im Anschluss an unserer Ausführungen zur Gestaltung von Formularen beschäftigen.

2 Gestaltung von Formularen

Was die Konzeption des Markups von Formularen im Hinblick auf deren visuelle Darstellung angeht, so erscheinen hierfür insbesondere die teilweise oben bereits vorgestellten Ausdrucksmittel für semantisches Markup wie `<label>` und `<output>` sowie `<fieldset>` und `<legend>` von Interesse. Mit diesen können Formulare hinsichtlich ihrer für die Gestaltung relevanten Elemente über die notwendigerweise erforderlichen Eingabeelemente wie `<input>`, `<select>`, etc. hinaus durchstrukturiert werden, ohne dass dafür in größerem Umfang proprietäre Markuplösungen mit `<div>`, `` sowie anwendungsspezifischen class Werten erforderlich wären.

Auf dieser Grundlage können dann grundsätzlich die allgemein verwendbaren Ausdrucksmittel von CSS verwendet werden, z. B. Style-Attribute wie `color`, `background-color`, `font-size`, `height`, `width` etc. Wie wollen uns anschließend insbesondere auf die spezifischen Ausdrucksmittel konzentrieren, die bezüglich des Status von Formularfeldern in CSS zur Verfügung stehen, die bestehende Problematik bei der Gestaltung von Checkboxes und Radiobuttons darlegen sowie die Formatierung von Formularen in mobilen Browsern anhand eines ausgewählten Beispiels beleuchten. Für eine umfangreichere Darstellung der Gestaltungsaspekte von Formularen verweisen wir auf [wwwHinweise im Mozilla Developer Network](#).

Pseudoklassen für Formulare

Die Ausdrucksmittel, die auf Ebene von HTML zur Deklaration der für die Bedienung relevanten Merkmale von Formularen zur Verfügung stehen und die Sie hier bereits kennengelernt haben, werden in CSS durch Pseudoklassen ergänzt, mittels derer einige durch HTML-Attribute deklarierbaren Eigenschaften direkt und ohne Verwendung von Attributselektoren überprüfen lassen. So

- lassen sich mittels der Pseudoklassen `:required` und `:optional` den durch `required` bzw. dessen Abwesenheit markierbaren obligatorischen bzw. optionalen Elementen eines Formulars spezielle Stileigenschaften zuweisen.
- erlauben die Pseudoklassen `:disabled` und `:readonly` die Zuweisung von Stilmerkmalen für *deaktivierte* bzw. als nicht editierbar markierte Elemente.
- können Eingabeelemente hinsichtlich ihres Gültigkeitsstatus, welcher auf Ebene von JavaScript durch das `ValidityState` Objekt repräsentiert wird, mittels der aussagekräftigen Pseudoklassen `:invalid` vs. `:valid` detektiert werden und dann entsprechende Stileigenschaften zugewiesen bekommen.

Checkboxes und Radiobuttons

Checkboxes und Radiobuttons stellen als Bedienelemente für die anwendungsspezifische Gestaltung insofern eine gewisse Herausforderung dar, als ihre Formatierung nicht mittels elementarer CSS-Properties umgesetzt werden kann. Lediglich die Dimensionen der Elemente sind modifizierbar, nicht jedoch z. B. deren Farbe oder Umrandungsstärke etc. Der Grund hierfür ist, dass Browser für diese Elemente eingebaute Grafiken nutzen, welche auch die verschiedenen Zustände der Elemente – ausgewählt, nicht ausgewählt, deaktiviert – repräsentieren.

Um eine anwendungsspezifische Gestaltung umzusetzen, müssen Sie dem entsprechend verschiedene Grafiken bereitstellen und diese unter Verwendung der o. g. Pseudoklassen bzw. der Pseudoklasse `:checked` zur Detektion eines ausgewählten Elements in CSS zuweisen. Weitere Hinweise Sie z. B. auf [www http://code.stephenmorley.org/html-and-css/styling-checkboxes-and-radio-buttons/](http://code.stephenmorley.org/html-and-css/styling-checkboxes-and-radio-buttons/)

Mobile Browser

Mobile Browser verwenden u. U. für Standard-Bedienelemente von Formularen Style-Einstellungen, die von denen der entsprechenden Desktop Browser abweichen. Unten sehen Sie beispielsweise die Realisierung von Texteingabefeldern und Buttons, für die keine spezifischen Einstellungen bezüglich Hintergrundfarbe und Form vorgenommen wurden. So runden sowohl Firefox für Android (oben) als auch Safari in iOS7 (unten) – trotz Flat Designs... – Eingabefelder und Buttons ab und fügen für letztere einen Farbverlauf hinzu:

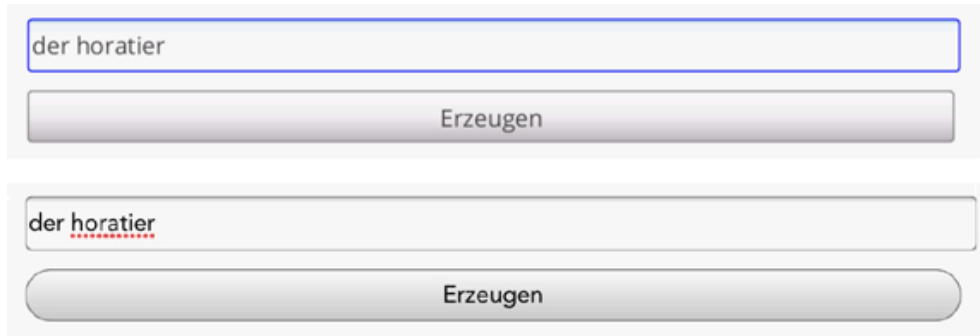


Abb.: Buttons im Standard Browser Style

Soll für die beiden mobilen Browser jeweils die gleiche Realisierung wie in Firefox für Desktop-Geräte umgesetzt werden, ist daher ein explizites Setzen von Style-Properties erforderlich, so wird z. B. durch die Setzung der `background` Property auf Android bereits die Erzeugung des Farbverlaufs unterbunden. Auf iOS ist zusätzlich das Setzen der `appearance` Property bezüglich des Webkit Browsers erforderlich, um die browsereigene Realisierungsform für den Button zu unterbinden:



Quellcode

```
001 input[type=submit],input[type=button],button {
002   background: rgb(240,240,240);
003 }
004 input {
005   -webkit-appearance: none;
006   border-radius: 0;
007 }
```

Im Resultat sehen wir dann, dass die Realisierung des Formulars auf den beiden mobilen Browsern mit der Realisierung im Desktop Firefox (ganz oben) übereinstimmt:

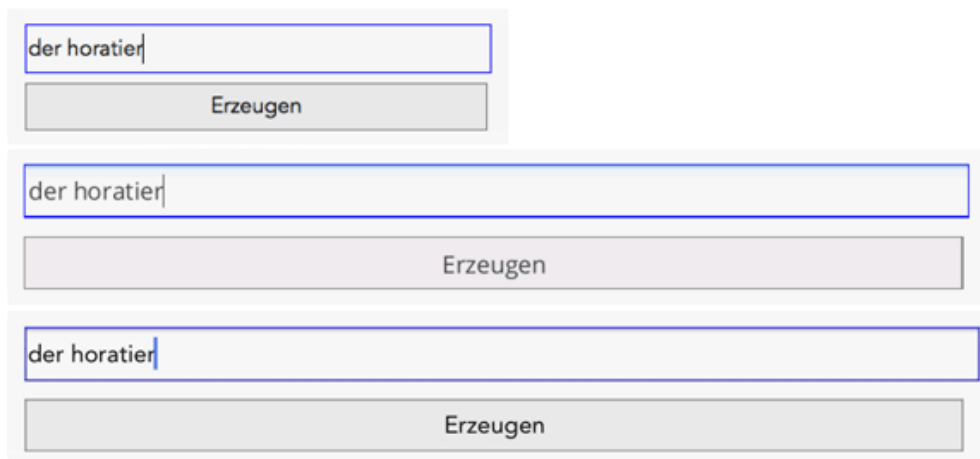


Abb.: Buttons mit CSS Style

Als Abschluss unserer Beschäftigung mit Formularen werden im nachfolgenden Abschnitt mit Multipart Formularen noch einmal auf eine Basisfunktionalität von Formularen eingegangen, die insbesondere für die „aktive“ Verwendung von Multimedia in Webanwendungen relevant ist und die die Anwendungsfälle von Formularen gegenüber den mit den bisherigen Mitteln realisierbaren Szenarien deutlich erweitert.

3 Multipart Formulare

Bisher hatten wir bezüglich der Übermittlung von Formulardaten an eine serverseitige Anwendung zwei Alternativen kennengelernt. Zum einen hatten wir die Default-Verarbeitung von Formularen beschrieben, bei der der Browser die Formulardaten als URL-kodierten String wahlweise als Query-Postfix der URL oder im Body eines HTTP `POST` Requests an die im Formular als `action` angegebene URL überträgt. Um dem Server anzuzeigen, um was für einen Typ von Daten es sich im letzteren Fall handelt, wird hierfür durch den Browser der `Content-Type` Header auf den Wert `application/x-www-form-urlencoded` gesetzt.

Alternativ dazu hatten wir gezeigt, wie Formulare Daten als JSON-Objekte im Body von eines `XMLHttpRequest` übertragen werden können. Beide Varianten stützen sich jeweils darauf, dass die zu übermittelten Daten in einem textuellen Format repräsentiert werden können. Dass auch Dateiinhalte, die in einem Binärformat vorliegen textuell repräsentierbar sind, werden wir in der nachfolgenden Lerneinheit anhand von Bilddaten sehen. Hier jedoch wollen wir eine alternative Übertragungsform für Binärdaten betrachten, die uns in HTML durch das Ausdrucksmittel sogenannter „Multipart Formulare Daten“ gegeben ist.

Multipart Formulardaten werden üblicherweise verwendet, wenn ein Formular ein oder mehrere `input` Elemente vom Typ `file` enthält. Für ihre Übertragung wird der Body von HTTP Requests – üblicherweise `POST` oder `PUT` Requests – verwendet. In diesem werden die Daten in einem spezifischen Format repräsentiert, das in folgender Abbildung zu sehen ist.

Content-Type Header:

```
Content-Type multipart/form-data; boundary=-----181051494316288526585091050
```

Body des HTTP Requests:

```
-----181051494316288526585091050
Content-Disposition: form-data; name="contentMode"

fileupload
-----181051494316288526585091050
Content-Disposition: form-data; name="src"; filename="muller.jpg"
Content-Type: image/jpeg

y0yA~JFIF~d~d~y1~Ducky~y1~Adobe~dA~y0~s0$0$,;2222;C>>>
884884C4)4ccc27cccccccccccccccccccccccccccccccccccyA~,"yA~
~
~!AQa0q"i!A2N8nBRB#r3CS?A0S~
~
~y0~?~_~Y!A"N6LJdPTDÜD35)µIY" Y!8G:B'6YB$K! #fIX+~s{0!i"}IÄlp×
n!B~888r
```

Abb.: Übertragung von Multipart
Formulardaten

Von den als Wert des Feldes `src` übertragenen Bilddaten ist nur ein Ausschnitt zu sehen.

Die Formularfeldwerte werden hier durch eine wiederkehrende Zeichenkette als Trennzeichen voneinander separiert. Jedes Feld wird außerdem mit einem eigenen Header versehen, in dem u. a. der Name des Feldes genannt wird. Für den Fall, dass der Wert des Feldes nicht textueller Form vorliegt, wird hier außerdem ein `Content-Type` angegeben. Sie sehen auch, dass für den Fall der Übertragung von Daten aus Dateien der Name der jeweiligen Datei als Wert eines `filename` Attributs angegeben wird. Auf diese Weise ist es möglich, nicht nur mehrere Felder, sondern insbesondere auch mehrere Dateiinhalte mit unterschiedlichen Content Types zu übermitteln.

Trennzeichen

Dem Trennzeichen, anhand dessen die Segmentierung der Formulardaten erfolgt, kommt eine besonderer Bedeutung für Multipart Formulare zu. Um die Lesbarkeit der Daten für den Empfänger zu gewährleisten muss in jedem Fall verhindert werden, dass die betreffende Zeichenkette zufällig in einem der Formularfeldwerte selbst auftritt. Die Zeichenkette muss also dynamisch durch den Browser bei Aufbau des Formulars und unter Annahme seiner Kenntnis der zu übertragenden Daten ermittelt werden.

In vorheriger Abbildung „Übertragung von Multipart Formulardaten“ oben sehen Sie außerdem, dass der Client im Header des HTTP-Requests nicht nur die Tatsache anzeigt, dass der Request Body Multipart Formulardaten enthält, sondern zusätzlich die Trennzeichenkette übermittelt. Auf dieser Grundlage kann dann das Auslesen der Daten serverseitig durchgeführt werden.

Wir zeigen nachfolgend zunächst, welche Möglichkeiten zur Übertragung von Multipart Formulardaten in HTML und JavaScript bestehen. Danach werden wir in Kürze auf die spezifische Behandlung von Multipart Requests in der Beispielanwendung eingehen.

3.1 Initiierung von Multipart Requests


Wie für die „gewöhnliche“ Formulardaten können auch die Daten aus Multipart Formularen Browserseitig auf zwei Wegen übermittelt werden. Zum einen verfügen Browser über eine eingebaute Funktionalität, Multipart Request auszulösen. Hierfür muss lediglich für das verwendete `<form>` Element die Attributsetzung `enctype="multipart/form-data"` vorgenommen werden. Liegt keine Intervention eines `onsubmit` Handlers vor, werden dann die Formulardaten als Multipart `POST` Request an die im Formular angegebene `action` URL übermittelt. Oben hatten wir erwähnt, dass in diesem Fall der Browser versucht, das aktuell dargestellte HTML-Dokument durch die Inhalte zu ersetzen, die ihm im Body des HTTP Responses zurückübermittelt werden.

`<iframe> target`

Hierzu existiert jedoch eine Alternative: es kann im `<form>` Element die `id` eines im Dokument vorhandenen `<iframe>` Elements als Wert eines `target` Attributs angegeben werden. Ein `<iframe>` ist gewissermaßen ein in ein Dokument „eingebettetes Browserfenster“, aus dem heraus eine Client-Server Interaktion unabhängig vom umgebenden Dokument und ohne Aktualisierung des umgebenden Inhalts initiiert werden kann.

Wird ein `<iframe>` als Wert von `target` angegeben, so hat dies zur Folge, dass der Response auf den HTTP-Request, der durch das Formular ausgelöst wurde, seinerseits unabhängig vom umgebenden Dokument zur Darstellung gebracht werden kann, d. h. es findet kein Neuladen des umgebenden Dokuments statt. Erwähnt werden sollte außerdem, dass aus einem `<iframe>` heraus auf JavaScript Funktionen des umgebenden Dokuments zugegriffen werden kann. In welcher Form diese Lösung verwendet werden kann, um in unserem konkreten Fall Dateiinhalte an den Server zu übermitteln und dessen Response unabhängig von der umgebenden Ansicht auszuwerten, werden wir im anschließenden Abschnitt aufzeigen.

FormData API

Zur Übermittlung von Multipart Formulardaten bei gleichzeitiger Beibehaltung des initiierten Dokuments im Browser existiert eine aktuelle Alternative, die analog zu unserer bisherigen Verwendung von `XMLHttpRequest` gehandhabt werden kann. Bei dieser hatten wir aus einem `onsubmit` Event Handler heraus und unter Unterbindung der `action` Behandlung des Formulars `XMLHttpRequests` mit `JSON`-Daten an den Server übermittelt und das Resultat im Callback des Requests ausgewertet. Wollen wir nun anstelle von serialisierten `JSON`-Objekten Multipart Formulardaten verwenden, so können wir hierfür die im  JavaScript Interface FormData bereitgestellte API verwenden. Diese erlaubt uns den schrittweisen Aufbau eines Objekts, das am Ende die zu übermittelnden Formulardaten repräsentieren wird.

Grundlage für das hier betrachtete Beispiel ist das nachfolgend dargestellte Formular, das in erweiterter Form auch in den Implementierungsbeispielen zu finden ist. Es dient der Erstellung von Instanzen des Typs `Einfuehrungstext` gemäß dem Datenmodell unserer Anwendung. Dafür verwenden wir `hidden` Eingabefelder, um bei der Erstellung Werte für die beiden Attribute `type` und `render_container` zu übermitteln, welche nicht durch den Nutzer über die Formularoberfläche eingegeben werden sollen. Anhand der Attributsetzungen auf `<form>` können Sie außerdem ersehen, dass das Formular auch für die Browser unterstützte Übermittlung von Multipart Formulardaten vorbereitet ist, die wir eingangs erwähnt haben:



Quellcode

Multipart Formulardaten

```

001 <form id="einfuehrungstextForm" action="http2mdb/content" method="post"
002   enctype="multipart/form-data" target="multipart_target">
003   <!-- das Feld für die Auswahl einer Datei, deren Inhalte übermittelt
004   werden -->
005   <input type="file" name="src" required="required" disabled="disabled"/>
006   <!-- zwei hidden Felder, die nicht einstellbare Attributwerte
007   übermitteln -->
008   <input type="hidden" name="type" value="einfuehrungstext"/>
009   <input type="hidden" name="render_container" value="left"/>
010   <!-- der submit Button -->
011   <input type="submit" value="Erzeugen"/>
012 </form>

```

Der nachfolgende Beispielcode wird aus einem Event-Handler aufgerufen, der auf `submit` Ereignisse bezüglich dieses Formulars reagiert. Im folgenden Beispielcode ist diesbezüglich insbesondere der Zugriff auf die Dateiinhalte interessant. Diese sind über den Wert des oben als `type="file"` markierten `input` Elements über das Array-wertige Attribut `files` auf der Repräsentation des Feldes zugänglich – beachten Sie aber mit Blick auf etwaige Sicherheitsbedenken, dass dem Zugriff via JavaScript die Auswahl der betreffenden Datei durch den Nutzer vorausgegangen ist, d. h. hiermit ist keineswegs ein Zugriff auf beliebige Dateiinhalte verbunden:



Quellcode

FormData Objekt

```

001 // erstelle das FormData Objekt
002 var formdata = new FormData();
003 // setze die Formularfeldwerte
004 formdata.append("src", form.src.files[0]);
005 formdata.append("type", form.type.value);
006 formdata.append("render_container", form.render_container.value);

```

Mehr als diese vier Zeilen Code sind für den Aufbau der Multipart Formulardaten im gegebenen Beispiel nicht erforderlich. Umzusetzen ist nun lediglich noch der Aufbau eines `XMLHttpRequest`, dem das `formdata` Objekt übergeben wird. Dies erfolgt in der `send()` Methode, der wir in den bisherigen Fällen – vermittelt über die Utility-Funktion `xhr()` aus den Implementierungsbeispielen – die im Request Body zu sendenden Daten übergeben:



Quellcode

Request Objekt

```

001 // erzeuge ein neues Request Objekt
002 var xhr = new XMLHttpRequest();
003 // setze einen einfachen Callback, um auf den Response zu reagieren
004 xhr.onreadystatechange = function() { /* (...) */};
005 // baue einen Request mit der angegebenen Methode und bezüglich der
006 // angegebenen URL auf
007 xhr.open("POST", "http2mdb/content/formdata");
008 // übermittle die Formulardaten im Request Body
009 xhr.send(formdata);

```

Abgesehen davon, dass im `<form>` Element oben als `action` eine andere URL als im `XMLHttpRequest` angegeben ist, sind für den Server die auf diesem Wege übermittelten Daten nicht von denen zu unterscheiden, die der Browser ohne Intervention unseres `onsubmit` Handlers übermitteln würde. Wie die Bearbeitung durch den Server in beiden Fällen erfolgt und was diesbezüglich für die Umsetzung der Übungsaufgaben zur vorliegenden Lerneinheit zu berücksichtigen ist, werden wir im nächsten Abschnitt abschließend aufzeigen.

3.2 Multipart Requests in der Beispielanwendung

Die serverseitig in NodeJS ausgeführte Komponente der Beispielanwendung verfügt über eine einfache Unterstützung für die Bearbeitung von Multipart Formulardaten, die anwendungsfallunabhängig ist und insbesondere auch für die Umsetzung der Übungsaufgaben verwendet werden kann. Sie basiert auf der im Sommer 2013 verfügbaren Version des NodeJS Moduls [Formidable](#), das insbesondere einen Parser für Multipart Daten bereitstellt, und modifiziert im Modul `multipart.js` ein [vorliegendes Skript](#), das den Formidable Parser exemplarisch nutzt. Initiiert wird die serverseitige Verarbeitung der Daten durch Überprüfung des Content-Type Headers im Rahmen der Verarbeitung von POST Requests in `http2mdb.js`:



Quellcode

Überprüfung des Content-Type Headers

```
001 function doPost(uri, req, res) {
002   // überprüfe, ob ein Multipart Request vorliegt
003   if (utils.startsWith(req.headers["content-type"], "multipart/form-data;")) {
004     handleMultipartRequest(uri, req, res);
005   }
006   // führe die Behandlung "gewöhnlicher" POST Requests durch
007   else {
008     /* (...) */
009   }
010 }
```

Auch die Implementierung der hier aufgerufenen Funktion `handleMultipartRequest` ist recht überschaubar. Hier wird auf dem Modul `multipart` die Bearbeitungsmethode für Multipart Requests aufgerufen und das Ergebnis an den Client zurück übermittelt:



Quellcode

Multipart Request

```
001 function handleMultipartRequest(uri, req, res) {
002   // lies die Multipart Daten aus
003   multipart.handleMultipartRequest(req, res, "../webcontent/", "content/",
004     function(ondone(formdata)) {
005       // übermittle das Ergebnis des Auslesens - formdata - an den Client
006       /* (...) */
007     });
008 }
```

Was aber passiert nun in der hier aufgerufenen Methode, welche Bewandnis hat es mit den übergebenen Argumenten und wie sehen die Daten aus, die als `formdata` an die hier angegebene Callback-Funktion übergeben und von dieser im HTTP-Response an den Client übermittelt werden? In Kürze lässt sich die Funktionalität der Funktion wie folgt zusammenfassen:

1. Sie liest die Multipart Formulardaten aus und verwendet dafür u. a. die im Header des Requests angegebene Information zur Trennzeichenkette bezüglich der Formularfelder
2. Enthält ein Formularfeld Dateiinhalte, werden diese ausgelesen und in eine neu erstellte Datei übertragen, wobei für jeden Inhaltstyp ein eigenes Verzeichnis verwendet wird. Der Wurzelpfad der Verzeichnisse, in die die Daten geschrieben werden, wird im Argument „../webcontent“ angegeben.
3. Aus den ausgelesenen Formulardaten wird ein JSON-Objekt aufgebaut. Dieses enthält für gewöhnliche Formularfelder die jeweils vom Browser übermittelten Feldwerte. Enthält ein Formularfeld Dateiinhalte, wird als Wert eine relative URL gesetzt, anhand derer die im vorangegangenen Schritt erstellte Datei identifiziert werden kann. Als initiales Segment der URL wird hier das Segment „content/“ verwendet. Außerdem wird der Content-Type der Inhalte als Wert eines `mediatype` Attributs gesetzt.
4. Unter Übergabe des JSON-Objekts als Argument wird schließlich die angegebene Callback-Funktion aufgerufen.

Im wesentlichen nimmt das `multipart.js` Modul also die Umformung der Multipart Formulardaten in ein JSON-Objekt vor, welches eine URL bezüglich der übermittelten Dateiinhalte *als Werte der Felder* enthält, welches die Dateiinhalte im ursprünglichen Formular bereitgestellt hat – da nur ein `mediatype` Wert auf dem Objekt gesetzt wird, unterstützt die Implementierung derzeit nur Daten mit höchstens einem Dateiinhalt in sinnvoller Weise. Die an den Browser zurück übermittelten Daten unterscheiden sich also nur darin von den zuvor an den Server gesendeten Daten, dass sie nun anstelle der Dateiinhalte die URLs enthalten, anhand derer die Dateiinhalte auf dem Server referenziert werden können. Für das am Beginn unserer Betrachtung von Multipart Formularen gezeigte Beispiel sehen die Rückgabewerte an den Browser z. B. wie folgt aus:



Quellcode

Rückgabewerte an den Browser

```
001 {
002   "contentMode": "fileupload",
003   "mediatype": "image/jpeg",
004   "src": "content/img/1393353081925_muller.jpg",
005   "type": "einfuehrungstext",
006   "render_container": "left"
007 }
```

Wie nun mit diesem Objekt auf Clientseite verfahren werden kann, obliegt der jeweiligen Anwendung. In den Übungsaufgaben werden Sie mit den erhaltenen Daten beispielsweise erneut auf den Server zugreifen und darauf die `createObject()` Funktion Ihrer NodeJS Implementierung aufrufen. Für Sie wichtig – im Sinne einer Entlastung – ist jedoch, dass Sie sich um die serverseitige Implementierung der hochgeladenen Daten nicht mehr zu kümmern brauchen, da diese Ihnen durch die Funktion `handleMultipartRequest()` bereits abgenommen wurde.

Beachten Sie außerdem, dass bei dem hier vorgeschlagenen Vorgehen zur weiteren Bearbeitung der als Response auf einen Multipart Request erwiderten Daten clientseitig keine Variablen verwendet werden müssen, die Sie vor Durchführung des Dateiuploads setzen und für die Weiterverarbeitung berücksichtigen müssen. Vielmehr übermitteln Sie zusammen mit den Dateiinhalten bereits alle Daten an den Server, die Sie benötigen, um den Response verarbeiten zu können – die Implementierung von `handleMultipartRequest` wird diese alle in das JSON-Objekt übertragen, das Sie als Response erhalten.

Unterschiede bestehen clientseitig hinsichtlich der bisher noch nicht explizit dargestellten Rückübermittlung der Daten an den Server jedoch in Abhängigkeit davon, ob Sie für die Übermittlung einen `XMLHttpRequest` mit `FormData` verwendet haben oder das Multipart Formular mit einem `<iframe>` als `target`. Im ersteren Fall können Sie das JSON-Objekt, das Sie für die Weiterbearbeitung benötigen, im `onreadystatechange` Callback aus dem Response auslesen – genau wie Sie es für alle anderen Fälle tun, in denen Sie JSON-Daten in einem HTTP-Response erwarten. Was aber müssen wir tun, um diese Daten bei Verwendung eines `<iframe>` in unserer Anwendung „auffangen“ zu können?

<script> und <iframe>

Bedenken Sie hierfür, dass es sich bei den Inhalten, die Sie an einen `iframe` übermitteln, um wohlgeformtes HTML-Markup handeln muss. Dazu gehören aber nicht nur Markup-Elemente, die dann im `<iframe>` zur Darstellung gebracht werden können, sondern z. B. auch das `<script>` Element. Wenn wir ein solches an den Browser zurückgeben, wird dieser es seinerseits dem `<iframe>` übergeben, der seinerseits nun versuchen wird, das Skript auszuführen. Und da, wie oben erwähnt, JavaScript Funktionen aus dem umgebenden Dokument von innerhalb eines `<iframe>` aufgerufen werden können, haben wir auch in einem solchen Skript Zugriff auf Funktionen des umgebenden Dokuments. Eine solche Funktion nutzen wir nun zur Umsetzung eines Callback-Mechanismus, der die an den Browser und in den `<iframe>` übermittelten Daten wieder unserer eigentlichen Anwendung zur Verfügung stellt. Dieser ist in folgender Abbildung dargestellt.

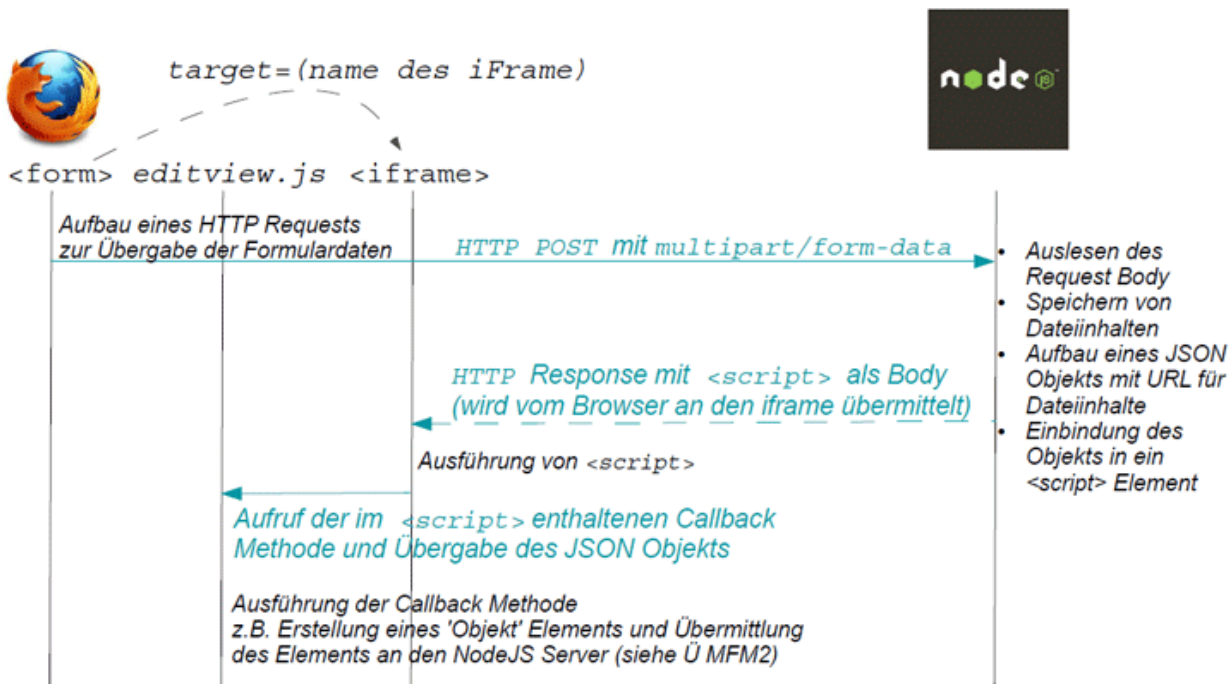


Abb.: Verwendung eines unsichtbaren <iframe> Elements zur Umsetzung eines Callback-Mechanismus bezüglich des Rückgabewerts eines Multipart Requests

Grundlage hierfür ist die serverseitige Erzeugung eines <script> Elements wie des folgenden, das das oben gezeigte JavaScript Objekt an die Callback-Funktion `onMultipartResponse()` des umgebenden Dokuments übergibt. Zusätzlich wird der URL Pfad – hier „content“ – übergeben, bezüglich dessen der Request ausgeführt wurde. Falls erforderlich, kann u. a. anhand dessen die Callback-Funktion eine Fallunterscheidung bezüglich verschiedener Alternativen ihres Aufrufs durchführen:



Quellcode

Fallunterscheidung durch Callback-Funktion

```
001 var content = {"contentMode": "fileupload",
002   "mediatype": "image/jpeg",
003   "src": "content/img/1393365686690_muller.jpg",
004   "type": "einfuehrungstext",
005   "render_container": "left"};
006 window.top.window.onMultipartResponse('/content', content);
```

JSONP

Erwähnt sei abschließend, dass der hier beschriebene Mechanismus einem unter dem Namen JSONP bekannten Implementierungsmuster vergleichbar ist. Auch hier werden dynamisch generierte <script>-Inhalte verwendet, aus denen heraus Aufrufe an bereits in einer Anwendung definierte Funktionen unter Übergabe von JSON-Objekten initiiert werden, die zuvor serverseitig in das Skript eingefügt wurden. JSONP ist eine gängige Lösung, um die Blockierung von Cross-Site-Skripting durch Browser zu umgehen. Sie soll also den Zugriff auf serverseitige Funktionen außerhalb der Domain des geladenen Dokuments erlauben, ohne dass dafür eine Nutzeraktion erforderlich ist – wie sie z. B. die Auswahl eines durch ein <a> Element bereitgestellten Links darstellt. Damit erscheint JSONP aber auch mit Blick auf Sicherheitsaspekte grundsätzlich nicht unbedenklich (siehe diesbezüglich weiter führende Überlegungen hier: <http://json-p.org/>).

Im Gegensatz zu JSONP liegt in unserem Fall jedoch kein Cross-Site-Skripting vor, da der durch den Browser ausgelöste Multipart Request der Same Origin Policy des Browsers unterliegt und deren Voraussetzungen auch erfüllt. Die vorgeschlagene Lösung ist daher lediglich mit Blick darauf als zweifelhaft anzusehen, dass mit `FormData` mittlerweile eine deutlich elegantere Variante zur Verwendung von Multipart Requests zur Verfügung steht. Beachten Sie, dass für die Umsetzung dieser Lösung ggf. der server-seitige Quellcode der Implementierungsbeispiele so angepasst werden muss, dass die JSON Objekte, die aus der Verarbeitung der Formulardaten entstehen, direkt und ohne die „Verpackung“ in ein <script> Element an den Client übermittelt werden.

Zusammenfassung

- Mit Blick auf die Nutzerzufriedenheit gilt es grundsätzlich zu verhindern, dass ein Nutzer mehr Eingaben tätigt als unbedingt erforderlich.
- Die Gültigkeitsprüfung von Anliegensformulierungen ist ein wichtiger Bestandteil der maschinellen Verarbeitung von Formulardaten. Folgende Gültigkeitsaspekte sind zu unterscheiden: **Vollständigkeit**, **Wohlgeformtheit** und die **fachliche Gültigkeit**.
- Von Gültigkeitsprüfungen ausgenommen sind Felder, die zwar sichtbar, aber nicht bedienbar sind. Diese können mit dem Attribut `disabled` markiert werden.
- Um eigene Validierungsprüfungen durchzuführen, muss der Entwickler entscheiden, unter welchen Umständen eine Validierung durchgeführt werden soll, wie die betreffenden Umstände erkannt werden und auf welche Weise das Feedback kommuniziert werden soll.
- Grundlage für den Umgang mit Validierungsfunktionen auf der Ebene von JavaScript ist ein Ereignis namens `invalid`, das zum aktuellen Repertoire von DOM Events gehört.
- Die Ausdrucksmittel, die auf Ebene von HTML zur Deklaration der für die Bedienung relevanten Merkmale von Formularen zur Verfügung stehen, werden in CSS durch Pseudoklassen ergänzt.
- Die Formularfeldwerte von Multipart Formulardaten werden hier durch eine wiederkehrende Zeichenkette als Trennzeichen voneinander separiert. Jedes Feld wird außerdem mit einem eigenen Header versehen. Auf diese Weise können mittels Multipart Formularen ggf. Inhalte mehrerer Dateien mit unterschiedlichen Inhaltstypen übertragen werden.

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

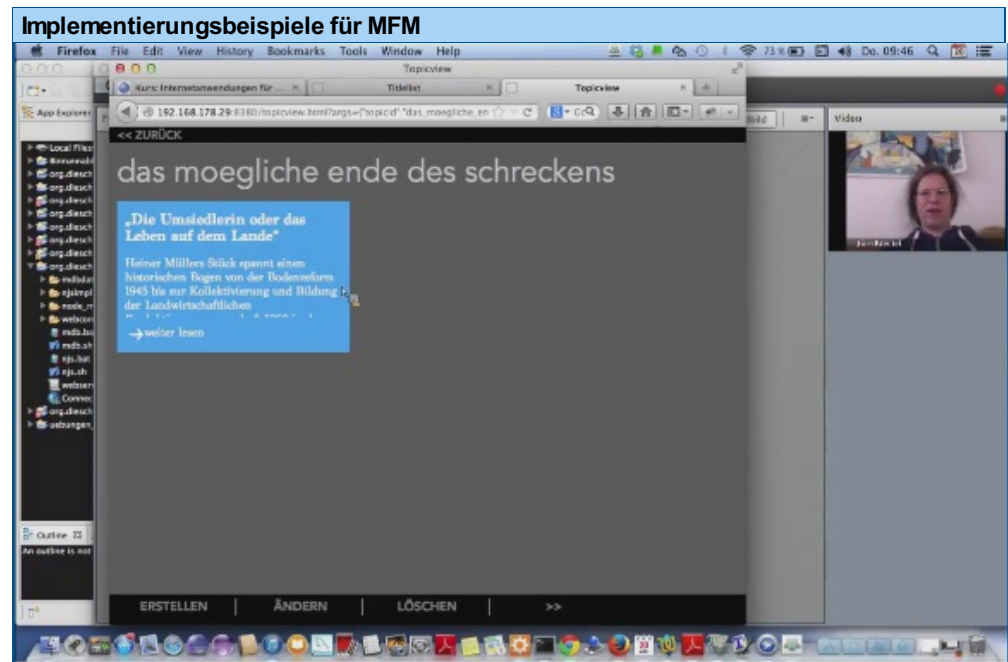
Übungen

Auch die Implementierungsbeispiele zum vorangegangenen Abschnitt finden Sie im Projekt `org.dieschnittstelle.iam.njm_frm_mfm`. Nun interessieren uns neben dem Markup in `tab_einfuehrungstext` insbesondere die Funktionen ab `initialiseEinfuehrungstextForm()` in `EditviewController.js`.

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



Film



© Beuth Hochschule Berlin - Dauer: 02:56 Min. - Streaming Media 6 MB



Übung MFM-01

Beispielanwendung

`EinfuehrungstextFormViewController.js/initialiseView()`

- In der Variable `radioOnClickListener` wird eine Funktion definiert, die auf die Betätigung eines Radiobuttons reagieren soll. Wie wird die Funktion den drei Radiobuttons zugewiesen und wie wird ermittelt, für welchen Button sie aufgerufen wird?
- In der Variable `inputOnInvalidListener` wird eine Funktion definiert, die auf das `invalid` Ereignis bezüglich mehrerer Eingabeelemente des Formulars reagieren soll. Welchen Eingabeelementen wird die Funktion zugewiesen?
- Ein direktes Auslesen des Werts eines Formularfelds, dessen mögliche Werte durch eine Menge von Radiobutton Elementen auswählbar sind, aus der Formularrepräsentation in JavaScript scheint nach Konsultation diverser Entwicklerforen nicht möglich zu sein. Wie wird im `onsubmit` Handler für das Formular `einfuehrungstextForm` ermittelt, ob ein Radiobutton ausgewählt wurde?
- Weshalb ist es in der Implementierung von `radioOnClickListener` nicht erforderlich, das `required` Attribut auf `inputSrc` bzw. `inputTxt` bei Deaktivierung des betreffenden Elements zu entfernen?

`EditviewViewController.js/submitEinfuehrungstextForm()`

- Weshalb wird im Fall der Option `fileupload` im Gegensatz zu den anderen Fällen der Wert `true` zurück gegeben und was hat dies zur Folge?
- Wie wird in der Behandlung der Option `fileuploadFormdata` auf die ausgewählte Datei zugegriffen, und weshalb kann hier davon ausgegangen werden, dass eine Datei ausgewählt wurde?

`topicview.html` und `EinfuehrungstextFormViewController.js`

- Wie wird das `<output>` Element `validationMessages` befüllt?
- Für welchen Fall bestimmen wir als Entwickler, welcher Text angezeigt werden soll?
- Wie wird im Fall einer Texteingabe oder im Fall der Änderung einer Auswahloption dafür gesorgt, dass eine ggf. angezeigte Fehlermeldung verschwindet?

`editview.css`

- Wie wird verhindert, dass die rote Default-Umrandung für als ungültig validierte Felder angezeigt wird?
- Wie wird das „Ausgrauen“ deaktivierter Eingabeelemente bewirkt?

`TopicviewViewController.js/onMultipartResponse()`

- Wer ruft diese Funktion auf und wann? Suchen Sie, falls erforderlich, im gesamten Projekt nach dem String `onMultipartResponse` (ohne Klammern).
- Wie wird der Funktion das JSON-Objekt als Argument übergeben?

Demovideo

Bearbeitungszeit: 60 Minuten



Übung MFM-02

Multipart Formular für Imgbox

Aufgabe

Erweitern Sie das CRUD Formular für Imgbox aus FRM2 um die Optionen, das anzuzeigende Bild durch Dateiauswahl hochzuladen oder es aus der Imgbox-Liste auszuwählen.

Anforderungen

1. Zur Auswahl der Optionen „URL“ und „Upload“ sollen Radiobuttons verwendet werden. In der nächsten Lerneinheit werden wir eine weitere Option hinzufügen.
2. Wird die Option „URL“ ausgewählt, dann soll die Eingabe einer URL obligatorisch sein.
3. Nur wohlgeformte URLs sollen verarbeitet werden.
4. Wird die Option „Upload“ ausgewählt, dann soll die Auswahl einer Bilddatei zum Upload obligatorisch sein.
5. Wird die Option „Liste“ ausgewählt, dann soll die Auswahl eines Imgbox-Elements aus der Imgbox-Liste obligatorisch sein.
6. Bei Auswahl der Option „Liste“ soll automatisch der Tab „Imgbox-Liste“ in den Vordergrund rücken. Alle weiteren Anforderungen zur Imgbox-Auswahl sind in Übung MFM3 formuliert.
7. Die Angabe eines Titels soll obligatorisch sein.
8. Die Angabe einer Description soll optional sein.
9. Ein Absenden des Formulars ohne Auswahl einer Option soll nicht möglich sein.
10. Falls die Option „URL“ ausgewählt ist, soll bei Absenden des Formulars die Erstellung des Objekts wie in **Übung FRM-02** erfolgen. Ein Wechsel zwischen den verschiedenen Erstellungsoptionen „URL“, „Upload“ und „Liste“ braucht für die Aktualisierung von Imgbox-Elementen nicht berücksichtigt zu werden.
11. Falls die Option „Liste“ ausgewählt ist, soll bei Absenden des Formulars nur eine entsprechende Imgbox-Referenz zum dargestellten `Topicview` hinzugefügt werden.
12. Falls die Option „Upload“ ausgewählt ist, soll bei Absenden des Formulars die ausgewählte Bilddatei auf den Server hochgeladen werden.
13. Mit dem Rückgabewert des Uploads aus Anforderung 12 soll wie in Übung FRM-02 ein neues „Objekt“ erstellt werden.

Bearbeitungshinweise

- **Anforderung 3:** Sie können dafür die in HTML vorgesehene Gültigkeitsprüfung nutzen
- **Anforderung 6:** Dafür können Sie die Funktion `showTabForElementtype()` in `EditviewViewController.js` verwenden.
- **Anforderung 9:** Sie können dafür eine geeignete Option vor-auswählen.
- **Anforderung 11:** Das Imgbox-Element braucht nicht erstellt zu werden, da es bei Auswahl aus der Liste ja bereits existiert.
- **Anforderung 12:** Sie können dafür genauso vorgehen, wie es die Implementierungsbeispiele für den Fall von „Einführungstext“ zeigen. Server-seitige Programmierung ist nicht erforderlich.

Demovideo

Bearbeitungszeit: 60 Minuten

**Übung MFM-03****Auswahl von Imgbox-Elementen aus der Liste****Aufgabe**

Erweitern Sie die Imgbox-Liste aus FRM-03 um eine Auswahlmöglichkeit, mit der die Option „Liste“ aus Übung MFM-02 umgesetzt wird.

Anforderungen

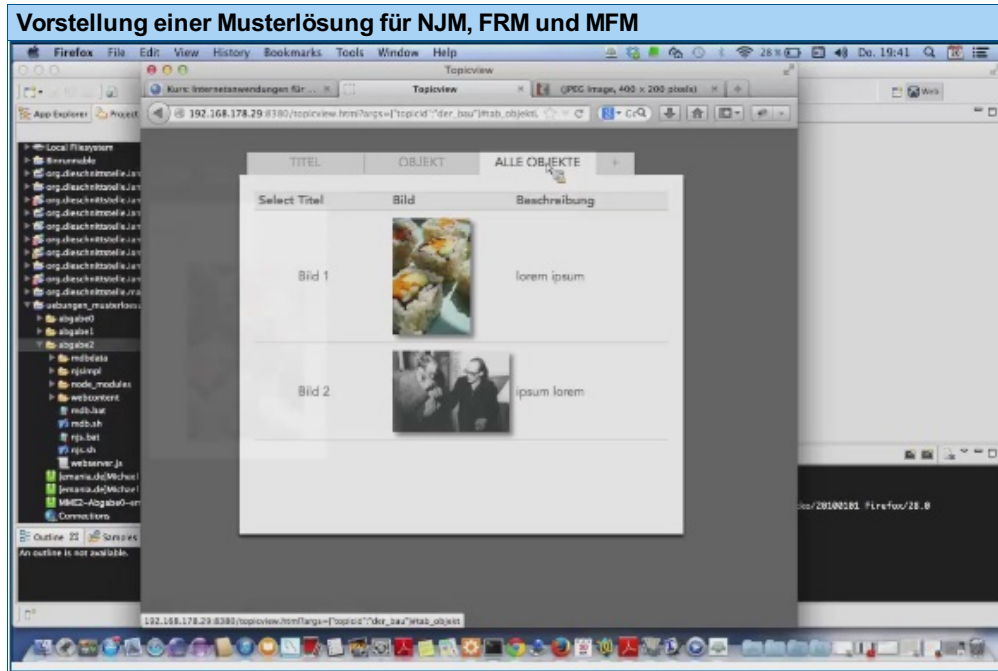
1. Erweitern Sie die Tabelle zur Darstellung der Imgbox-Elemente um eine Spalte, die für jedes Imgbox-Elemente ein Auswahlelement zur Auswahl des betreffenden Elements enthält.
2. Eine Auswahl soll nur möglich sein, wenn im Formular aus MFM-02 die Option „Liste“ ausgewählt ist.
3. Es soll immer nur ein Imgbox-Element ausgewählt werden können.
4. Wird ein Imgbox-Element ausgewählt, dann soll der Tab mit dem CRUD-Formular für elemente in den Vordergrund gebracht werden.
5. Die Attribute des Imgbox-Elements sollen dann in den entsprechenden Feldern des Formulars *nicht editierbar* angezeigt werden.

Bearbeitungshinweise

- *Anforderung 2:* d. h. auch wenn der Nutzer aktiv den Tab mit der Objektliste auswählt, soll eine Objektauswahl nur möglich sein, wenn die Option „Liste“ ausgewählt ist.
- *Anforderung 3:* ziehen Sie dafür die Verwendung einer Menge von Radiobuttons mit gleichem `name` Attribut sowie das Einbetten der Tabelle in ein `form` Element in Betracht.
- *Anforderung 5:* zu diesem Zweck müssen Sie das ausgewählte Imgbox dem Formular in geeigneter Form zur Verfügung stellen.
- Den Radiobuttons können Sie - wie in den Implementierungsbeispielen - einen gemeinsamen Event Handler zuweisen, in dem Sie überprüfen, für welches Imgbox-Elemente die Auswahl erfolgt. Weisen Sie den Radiobuttons dafür z. B. eine geeignete `id` zu.

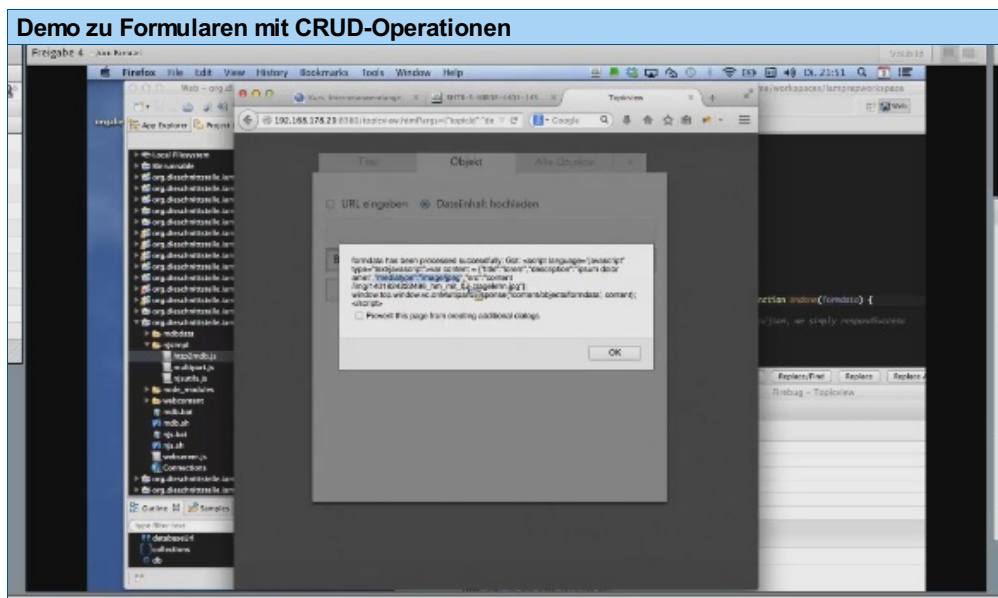
Bearbeitungszeit: 60 Minuten

Im folgenden Film stellt der Autor eine Musterlösung für die aufeinander aufbauenden Übungen der Lerneinheiten NJM, FRM und MFM vor.



© Beuth Hochschule Berlin - Dauer: 06:20 Min. - Streaming Media 13 MB

Im folgenden Film demonstriert der Autor die Umsetzung eines Formulars mit CRUD-Operationen.



© Beuth Hochschule Berlin - Dauer: 01:25 Std. - Streaming Media 184 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 MFM-04

Formularvalidierung

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

1. Welche drei Aspekte der Gültigkeit von Formularen können unterschieden werden?
2. Für welche Gültigkeitsaspekte stellt Ihnen HTML deklarative Ausdrucksmittel zur Verfügung?
3. Mittels welcher HTML-Attribute können Sie die Bedingungen für die Vollständigkeit von Formularen deklarieren und welche Attribute verwenden Sie zur Deklaration der Wohlgeformtheitsbedingungen?
4. Wann benötigen Sie für `<select>` Elemente ein `required` Attribut?
5. Wann wird durch den Browser die Vollständigkeit von Formulardaten überprüft und wann die Wohlgeformtheit von Formularfeldwerten?
6. Welche Einschränkung weist das `disabled` Attribut hinsichtlich der browser-unterstützten Übermittlung von Formulardaten an eine `action`-URL auf?
7. Werden als `disabled` markierte Felder durch den Browser hinsichtlich Vollständigkeit und Wohlgeformtheit berücksichtigt?
8. Was sind „versteckte“ Eingabefelder und wofür sind sie sinnvoll?
9. Was wird durch das `invalid` Ereignis angezeigt und wie können Sie dieses Ereignis behandeln? Was passiert, wenn Sie keine Behandlung dafür implementieren?

Bearbeitungszeit: 30 Minuten



Formulieren

Übung MFM-05

Anwendungsspezifische Validierung

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

1. Welche beiden Aspekte können hinsichtlich anwendungsspezifischer Validierung unterschieden werden?
2. Was wird durch das `ValidityState` Objekt repräsentiert und auf welcher Ebene eines Formulars ist dieses Objekt angesiedelt?
3. Was ermöglicht Ihnen die Methode `setCustomValidation()`?
4. Welche beiden Schritte sind erforderlich, um anwendungsspezifisch ein `invalid` Ereignis auszulösen?
5. Wie kann anwendungsspezifische Validierung vor Abschluss einer Formularbefüllung mittels `submit` initiiert werden?
6. Nennen Sie drei Eingabeereignisse, die bezüglich Formularelementen auftreten können.
7. Worin kann die Behandlung von Eingabeereignissen in Formularen bestehen?

Bearbeitungszeit: 20 Minuten



Formulieren

Übung MFM-06

Formatierung von Formularen

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

1. Wie können Sie auf Ebene von CSS die Darstellung von Checkboxes und Radiobuttons in Formularen beeinflussen?
2. Wie können Texteingabefelder und Button-Darstellungen beeinflusst werden?
3. Nennen Sie drei CSS Pseudoklassen, die speziell für die Gestaltung von Formularen verwendet werden können.

Bearbeitungszeit: 10 Minuten



Formulieren

Übung MFM-07

Multipart Formulare

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

1. Worin unterscheiden sich „gewöhnliche“ Formulare Daten von Multipart-Formulare Daten?
2. Können Multipart Formulare Daten mit `GET` Requests übermittelt werden? Begründen Sie Ihre Antwort.
3. Woher weiß ein Server, welche Trennzeichenkette für die Segmentierung von Multipart Formulare Daten verwendet wird?
4. Warum können in einem Multipart Formular Inhalte mehrerer Dateien mit unterschiedlichem Inhaltstyp übertragen werden?
5. Auf welche beiden Weisen können Multipart Requests initiiert werden?
6. Wie kann bei Übermittlung von Formulare Daten an die `action` URL der vom Server übermittelte Response ohne Neuladen des dargestellten Dokuments verarbeitet werden?
7. Welche Vorteile bietet die Verwendung von `FormData` Objekten zur Übermittlung von Formulare Daten?

Bearbeitungszeit: 20 Minuten