

Internetanwendungen für mobile Geräte:

Mobile Web Framework Tutorial

Virtuelle Fachhochschule und Beuth Hochschule für Technik

Wintersemester 2018/2019, Stand: 11. September 2018

Jörn Kreutel

Inhaltsverzeichnis

1	Kurzeinführung	2
1.1	MWF Funktionsumfang	2
1.2	MWF Architektur und Komponenten	4
1.3	MWF und JavaScript	6
2	Anforderungen	8
3	Vorbeitung	16
4	Umsetzung der Listenansicht	18
4.1	Gestaltung	18
4.2	Datenmodell	21
4.3	Steuerung	26
4.3.1	Erstellen eines neuen View Controller	26
4.3.2	Befüllung der Listenansicht	28
4.3.3	Auswahl von Listenelementen	32
4.3.4	Hinzufügen eines neuen Elements	32
4.3.5	Verwendung von IndexedDB	35
4.4	Fortgeschrittene Steuerungsfunktionen	38
4.4.1	Verwendung von <i>Ractive.js</i> Templates	38
4.4.2	Aktionsmenü für Listenelemente	40
4.4.3	Verwendung des MWF Entity Managers	44
4.4.4	Dialoge mit <i>GenericDialogTemplateViewController</i>	49
5	Umsetzung der Leseansicht	58
5.1	Gestaltung	58
5.2	Steuerung	61
6	Verwendung von Event-Notifikationen zur Reaktion auf CRUD Operationen	68

1 Kurzeinführung

Das vorliegende Tutorial dient als Einführung in die Ausdrucksmittel des *Mobile Web Frameworks* MWF, das zum Zweck der Verwendung in unserer Lehrveranstaltung entwickelt wurde. Ziel von MWF ist es zum einen, die Entwicklung von Webanwendungen für mobile Geräte mit typischen Ansichtstypen wie Listenansichten, Detailansichten und Editieransichten gegenüber deren Umsetzung auf Basis der ‘rohen’ Funktionen der Webtechnologien HTML, CSS und JavaScript zu vereinfachen. Zugleich soll es erlauben, in der Lehre den Bezug der mit seiner Hilfe umgesetzten Funktionsmerkmale zu den genannten grundlegenden Ausdrucksmitteln von Webtechnologien herzustellen und damit einen ‘Blick hinter die Kulissen’ zu werfen. Die nachfolgenden Kapitel tun ausschließlich dem erstgenannten Aspekt Genüge und zeigen die Verwendung des Frameworks zur Umsetzung einer konkreten Anwendung, deren Anforderungen vorab in Form von Übungsaufgaben aus dem Lehrprogramm der Veranstaltung beschrieben werden. Im folgenden geben wir einen Überblick über den derzeit verfügbaren Funktionsumfang von MWF und gehen dann kurz auf die Verwendung objektorientierter Ausdrucksmittel von JavaScript bei der Anwendungsentwicklung mit MWF ein.

1.1 MWF Funktionsumfang

Wesentliche Inspirationsquelle von MWF sind die Ausdrucksmittel, die das *Android Java Framework* für die Entwicklung nativer mobiler Applikationen bereitstellt. Aus Android und – auch begrifflich – aus iOS ist die Idee entnommen, dass der Aufbau der Ansichten einer Anwendung und die Reaktion auf die Interaktion der Nutzer mit den dargestellten Bedienelementen durch spezifische **View Controller** Komponenten gesteuert wird¹ und dass diese Komponenten einen **Lebenszyklus** durchlaufen². Demnach existiert ein View Controller nicht notwendigerweise nur solange, wie die von ihm gesteuerte Ansicht im Vordergrund sichtbar ist, sondern überdauert auch den Übergang in etwaige Folgeansichten, bis er seinerseits zum Zweck der Darstellung einer Vorgängeransicht verlassen wird. Während seiner Lebensdauer kann er auf **Lebenszyklusereignisse** wie die Erzeugung, die Darstellung im Vordergrund, das temporäre Pausiertwerden und seine letztendliche Entfernung reagieren und ggf. die dargestellten Ansichten anpassen, z.B. wenn während des Pausiertseins Änderungen an den darzustellenden Daten vorgenommen worden sind. An Android und iOS orientiert sich ferner die Idee, dass es eine gesamte Laufzeit einer Anwendung überdauernde Zustandbehaftete Komponente gibt, die als **Applikation** bezeichnet wird und auf die die View Controller zugreifen können. Wie **Activities** in Android kommunizieren View Controller in MWF nicht direkt miteinander; jegliche Interaktion zwischen ihnen wird durch das Framework vermittelt.

Was die Anbindung von View Controllern an die Oberfläche der von ihnen gesteuerten Ansich-

¹ Siehe zur Erläuterung der in Android als *Activities* bezeichneten View Controller und für iOS die Darstellung in <http://developer.android.com/guide/components/activities.html> bzw. <https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/>.

² Siehe z.B. die Erläuterungen zum Lebenszyklus für *Activities* auf <http://developer.android.com/reference/android/app/Activity.html>.

ten angeht, so unterstützt MWF das ***bidirektionale Data Binding*** zwischen den Ansichten einer Anwendung und den in einer Ansicht dargestellten bzw. durch Nutzerinteraktion mit der Ansicht manipulierten Daten, wie es z.B. im JavaScript GUI Framework *AngularJS*³ gebräuchlich ist.⁴ Zu diesem Zweck bindet MWF das Templating Framework *RactiveJS*⁵ ein. Wie AngularJS oder *jQuery Mobile*⁶ ermöglicht MWF die Entwicklung von ***Single Page Applications***, bei denen Ansichtsübergänge stets ohne Neuladen von Dokumenten im Vordergrund vorstatten gehen, wodurch u.a. der JavaScript Ausführungskontext der Anwendung über die gesamte Nutzungsdauer verfügbar ist. Die in AngularJS und anderswo vorgesehene Möglichkeit, einzelne Teilansichten mit ***Routen*** als individuellen in der Adresszeile des Browsers darstellbaren URLs zu assoziieren, besteht in MWF nicht, daher kann zur Navigation zwischen Ansichten der ‘Zurück’-Buttons des Browsers bzw. des mobilen Endgeräts nicht verwendet werden – dessen Einsatz führt vielmehr zum sofortigen Verlassen der zugegriffenen Anwendung. Auf dem derzeitigen – vorläufigen – Entwicklungsstand von MWF ist es ferner nicht möglich, einzelne Ansichten einer Anwendung bei Verwendung im Hintergrund nachzuladen, d.h. alle Ansichten müssen in einem gemeinsamen HTML Dokument deklariert werden.

Die Anbindung der View Controller Funktionen an die darunter liegende ***Model*** Schicht einer Anwendung, die die von einer Anwendung verwendeten Datenbestände beinhaltet und den Zugriff auf diese mittels lesender und schreibender Operationen ermöglicht, wird in MWF durch eine im Hintergrund aktive ***Entity Manager*** Komponente gewährleistet. Diese kapselt u.a. die konkrete Umsetzung der Datenzugriffe, welche wahlweise auf server-seitigen oder lokalen, d.h. auf dem jeweiligen Endgerät liegenden, Datenbeständen operieren können. Auch damit wird ein für mobile Anwendungen wesentlicher Funktionsaspekt adressiert. Darüber hinaus gewährleistet der Entity Manager, dass für jede im Datenbestand vorliegende und eindeutig identifizierbare Instanz eines anwendungsspezifischen Datentyps zur Laufzeit maximal eine ***Entity Instanz*** im Speicher existiert, deren Identität über alle lesenden und schreibenden Zugriffe hinweg fortbesteht. Funktional und begrifflich ist der Entity Manager von MWF inspiriert durch die gleichnamige Komponente der *Java Persistence API (JPA)*⁷ – nicht nur die Transaktionalität schreibender Zugriffe unterstützt er jedoch im Ggs. zu letzterem nicht... Der MWF Entity Manager ist aber dazu in der Lage, unter Verwendung eines auch an anderen Stellen einer Anwendung verfügbaren ***Notifikationsmechanismus*** die Ergebnisse lesender und schreibender Zugriffe als ***Ereignisse*** an alle daran interessierten View Controller einer Anwendung zu kommunizieren. Damit unterstützt er eine – allerdings durch die View Controller vermittelte – Anbindung der Ansichten an das Model einer Anwendung über ein ***Publish-Subscribe*** Pattern⁸, wie es in der ursprünglichen Auslegung des ***Model View Controller***

³ <https://angularjs.org/>

⁴ Siehe für Data Binding in AngularJS <https://docs.angularjs.org/guide/databinding>.

⁵ <http://ractive.js.org/>

⁶ <https://jquerymobile.com/>

⁷ <http://www.oracle.com/technetwork/java/javase/tech/persistence-jsp-140049.html>

⁸ Siehe hierfür z.B. <http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>

Architekturmusters vorgesehen war.⁹

Zur **Modularisierung** des von einer Anwendung verwendeten JavaScript Codes – inklusive der Komponenten des MWF Frameworks selbst – wird *RequireJS*¹⁰ verwendet. Dies beinhaltet die Bereitstellung der von einem gegebenen JavaScript Modul erforderlichen Module über einen **Dependency Injection** Mechanismus¹¹.

Entwickelt und getestet wurde MWF mit einem aktuellen Firefox Browser für MacOS sowie einem Chrome Browser für Android 5 auf einem Google/LG Nexus 5 Smartphone, der Betrieb in Chrome und Safari für MacOS sowie in Chrome und Firefox für alle anderen Plattformen sollte ohne Probleme möglich sein. Als nicht funktionsfähig erwies sich MWF bisher in Safari für iOS9. In Chrome für iOS traten zwar unerwünschte Abweichungen bei der Formatierung von Eingabeelementen in Popup-Dialogen auf, abgesehen davon sollte Chrome für die Verwendung in iOS aber geeignet sein.

1.2 MWF Architektur und Komponenten

Einen Überblick über die Architektur des MWF Frameworks und die Einbettung individueller Anwendungen – ‘Applications’ – in dessen Rahmen finden Sie in Fig. 1. Diese nimmt außerdem eine Zuordnung der verschiedenen Klassen des Frameworks zum Model View Controller Architekturmuster vor.

Die Rollen, die den einzelnen Klassen und Komponenten bzw. deren Instanzen in der MWF Architektur zukommen, lassen sich wie folgt charakterisieren. Mit Unterstreichung markierte Klassen sind anwendungsspezifisch erweiterbare **Komponenten**:

- View Controller: baut Ansichten auf, steuert die Nutzerinteraktion mit den Bedienelementen einer Ansicht, führt lesende und schreibende Zugriffe auf Daten durch, initiiert Ansichtsübergänge, stellt ‘Komfortfunktionen’ für Listenansichten bereit
- Application: stellt ansichtsübergreifende Funktionen bereit, z.B. das ‘Umschalten’ zwischen lokalen und/oder remote Datenzugriffen, initialisiert Datenzugriffsoperationen beim Start der Anwendung
- Entity: Entity-Instanzen repräsentieren ‘die von einer Anwendung verwendeten Daten’, Entity-Typen beschreiben deren Struktur und ggf. Verhalten, stellen CRUD-Operationen für den lokalen oder remote Datenzugriff bereit und verbergen die Implementierung der Operationen
- EntityManager: verwaltet Entity-Instanzen; gewährleistet dabei, dass für jede mittels _id identifizierbare Entity zur Laufzeit höchstens eine Instanz im Speicher der Anwendung

⁹ Siehe hierfür z.B. <https://www.duo.uio.no/handle/10852/9621>

¹⁰ <http://requirejs.org/>

¹¹ Siehe z.B. die Erläuterungen zu Dependency Injection in der Dokumentation von AngularJS: <https://docs.angularjs.org/guide/di>

Framework

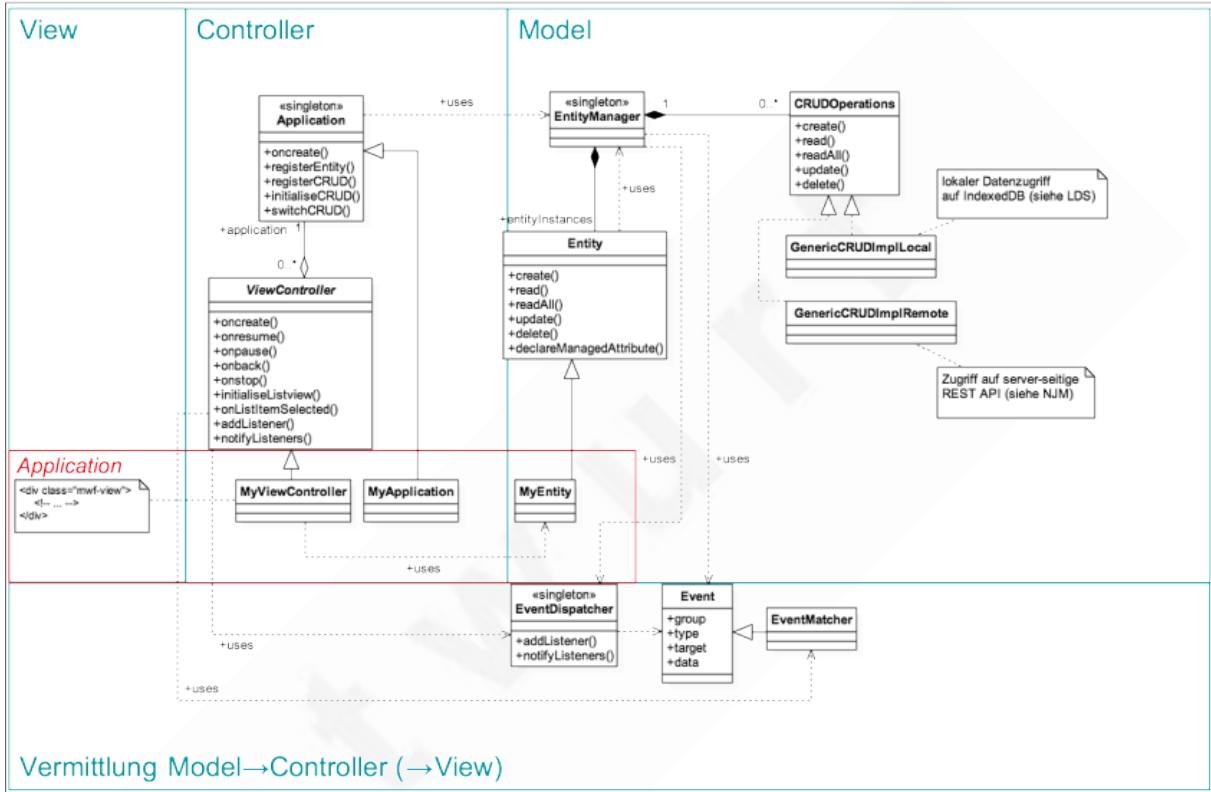


Fig. 1: Gesamtarchitektur des MWF Frameworks

existiert, verwaltet nicht kompositionale Assoziationen zwischen Entities. Ruft CRUD Operationen auf und generiert Events mit dem Ergebnis der Operation (`read`, `created`, `updated`, `deleted`)

- `EventDispatcher`: ermöglicht die Registrierung von Listenern für Events, verbreitet Events an die registrierten Listener, insbesondere View Controller.

Die folgenden Lebenszyklusmethoden für View Controller werden derzeit durch das Framework unterstützt. Unbedingt erforderlich ist lediglich die anwendungsspezifische Implementierung der `oncreate()` Funktion für jeden View Controller einer Anwendung:

Funktion	Aufruf durch MWF
<code>oncreate()</code>	Nach Erzeugen einer View Controller Instanz
<code>onresume()</code>	Bei erstmaliger und wiederholter Darstellung/‘Einblenden’ der durch die View Controller Instanz gesteuerten Ansicht
<code>onpause()</code>	Bei erstmaligem und wiederholtem Ausblenden der Ansicht
<code>onstop()</code>	Bei Entfernen der Ansicht.

Die Benennung und Funktionsweise dieser Lebenszyklusmethoden entspricht weitgehend der in Fig. 2 markierten Untermenge von Lebenszyklusmethoden in Android. Darauf hinaus lässt sich eine Analogie zu den dort ebenfalls dargestellten Lebenszyklusmethoden in iOS herstellen.

1.3 MWF und JavaScript

Die Implementierung von MWF verwendet die Ausdrucksmittel, die JavaScript zur *objektorientierten Programmierung* bereit stellt. Die Anwendung von MWF bei der Umsetzung einer konkreten Anwendung erfordert die Verwendung dieser Ausdrucksmittel insofern, als die anwendungsspezifischen Ausprägungen der generischen Komponententypen `Application`, `View Controller` und `Entity` jeweils die betreffende Oberklasse des MWF Frameworks beerben müssen. Hierfür erlaubt MWF in der aktuellen Version die Verwendung der ***Class-Syntax*** für die Deklaration von Objekttypen, die der Notation von Klassen in Java sehr nahe kommt.

Abgesehen davon machen Anwendungen, die MWF verwenden, recht hohen Gebrauch von Callback-Funktionen, wie sie aber auch bei einfacheren Anwendungsfällen von JavaScript gebräuchlich sind, z.B. bei der Registrierung von Event Listenern für DOM Events auf den Bedienelementen einer Ansicht. Callback-Funktionen in JavaScript sind ***Closures***, in denen lokale Variablen des Ausführungskontexts, in dem die Funktion erzeugt wurde, dauerhaft gebunden sind und bei Ausführung der Funktion ausgewertet werden können. (siehe <https://developer.mozilla.org/en/docs/Web/JavaScript/Closures>) Die aktuelle Version des Tutorials deklariert

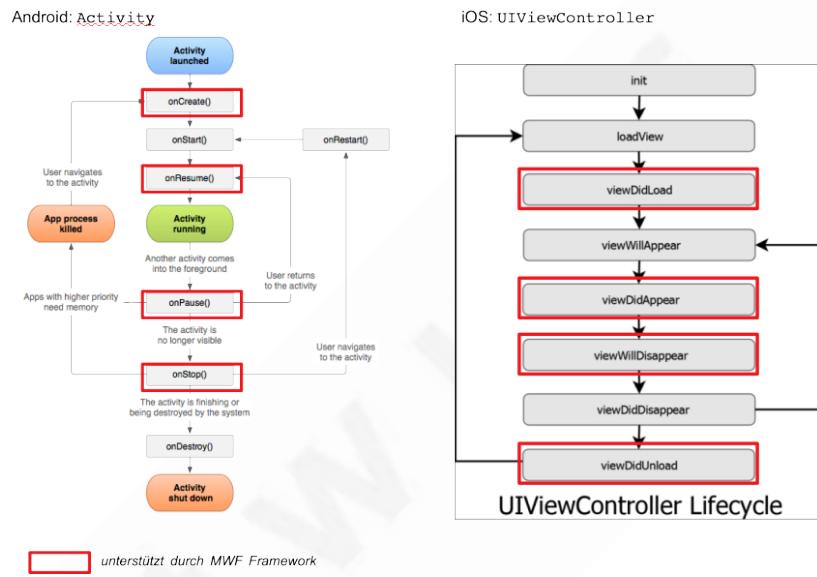


Fig. 2: Lebenszyklusmethoden in Android (links) und iOS (rechts)

Callback-Funktionen als **Pfeifunktionen**, was vorteilhaft gegenüber der vermutlich noch vertrauteren Notation mittels `function` Keyword ist, da hier die Notwendigkeit des expliziten Bindings der Funktionen an das die Funktion instantiierende Objekt mittels `bind()` entfällt (siehe für die Verwendung von `bind()` https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Function/bind).

Änderungen der bestehenden Implementierung und API von MWF, z.B. die Nutzung von Promises¹² anstelle von Callback-Funktionen oder von ‘echten’ HTML5 Templates¹³ anstelle des derzeit praktizierten ‘Ausschneidens’ entsprechend markierter Elemente aus dem DOM Objekt beim Start der Anwendung sind denkbare zukünftige Weiterentwicklungen des derzeitigen Entwicklungsstands.

2 Anforderungen

Die Anforderungen an die hier entwickelte Anwendung werden nachfolgend anhand von Übungsaufgaben beschrieben.

¹² https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise

¹³ <https://developer.mozilla.org/en/docs/Web/HTML/Element/template>

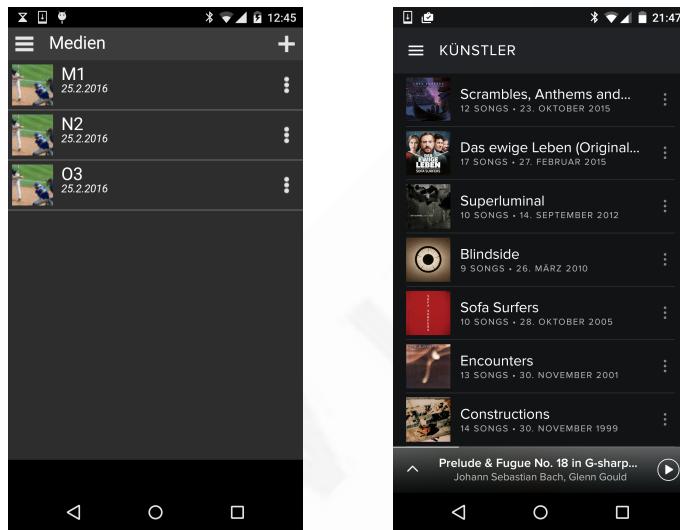
Ü MWF1 Listenansicht

(19 Punkte)

Aufgabe

Setzen Sie die nachfolgend links gezeigte und beschriebene Listenansicht gestalterisch und funktional um. Die Gestaltung dieser Ansicht wie auch der weiter in den folgenden Ansichten umgesetzten Dialoge orientiert sich am Beispiel der Spotify-App für Android.

Die Bepunktung dieser und der folgenden Aufgaben zu MWF berücksichtigt die Tatsache, dass alle genannten Anforderungen durch das angebotene Tutorial zur Lerneinheit MWF abgedeckt werden.



Anforderungen

1. Datenmodell

Die hier und im folgenden zu entwickelnde Anwendung soll es ermöglichen, `MediaItem` Objekte zu verwalten. Für den Zweck der Aufgaben zu MWF verfügen `MediaItem` Objekte über die folgenden Attribute:

- Name (**1 P.**)
- Bildquelle (**1 P.**)
- Erstellungsdatum des `MediaItem` Objekts (**1 P.**)

2. Gestaltung

- Die Ansicht soll aus Kopfzeile, Hauptbereich und Fußzeile aufgebaut sein. (**1 P.**)
- Die Ansicht soll exakt 100% Höhe und Breite der verfügbaren Anzeigefläche einnehmen und ohne horizontales oder vertikales Scrolling realisiert werden. (**1 P.**)

- In der Kopfzeile sollen linksbündig ein ‘Sandwich’-Icon und die Überschrift ‘Medien’ sowie rechtsbündig ein ‘+’-Icon dargestellt werden. (**2 P.**)
- Listenelemente sollen linksbündig ein Bild-Element einen Titel mit darunter platziertem Datum sowie rechtsbündig ein ‘Optionen’-Icon verwenden. (**3 P.**)
- Im Titel- und Untertitelelement sollen Name bzw. Erstellungsdatum des `MediaItem`, im Bild-Element der Inhalt der Bildquelle des Objekts dargestellt werden (**3 P.**)

3. CRUD Operationen

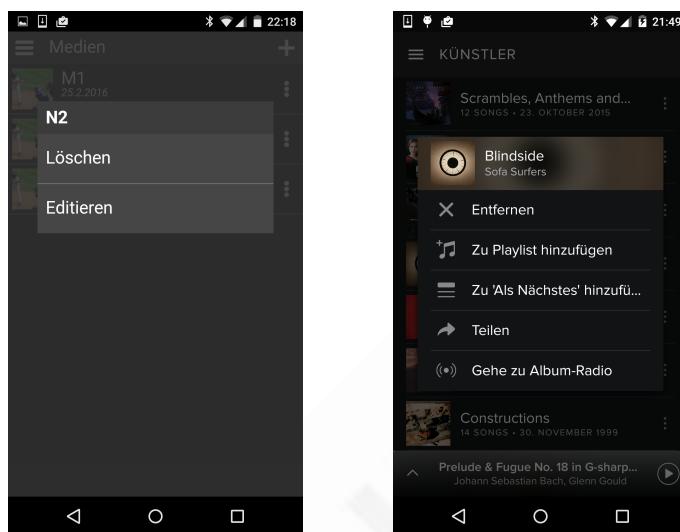
- Beim Zugriff auf die Listenansicht sollen `MediaItem` Objekte aus einer lokalen IndexedDB Datenbank ausgelesen und in der Listenansicht dargestellt werden. (**2 P.**)
- Bei Betätigen des ‘+’-Bedienelements soll die Erstellung eines neuen Elements unter Verwendung des in MWF3 umgesetzten Dialogs erfolgen. (**1 P.**)
- Alle nachfolgend umgesetzten CRUD Operationen sollen bezüglich der lokalen Datenbank durchgeführt werden. (**3 P.**)

Ü MWF2 Aktionsmenü für Listenelemente

(8 Punkte)

Aufgabe

Erstellen Sie ein als Dialog realisiertes Aktionsmenü für Listenelemente



Anforderungen

- Das Aktionsmenü soll in einer Titelleiste den Namen des ausgewählten Listenelements anzeigen und darunter die beiden Aktion ‘Löschen’ und ‘Editieren’ anbieten. (2 P.)
- Das Aktionsmenü soll bei Betätigung des ‘Optionen’-Icons eines Listenelements geöffnet werden. (1 P.)
- Bei Click/Tap außerhalb des Menüs wird dieses wieder geschlossen. (1 P.)
- Wird das Aktionsmenü geöffnet, dann soll die Listenansicht über einen kurzen Zeitraum hinweg partiell ausgeblendet werden. (1 P.)
- Bei Auswahl der ‘Löschen’-Aktion soll des betreffende `MediaItem` aus der lokalen Datenbank gelöscht und aus der Listenansicht entfernt werden. (2 P.)
- Bei Auswahl der ‘Editieren’-Aktion soll der in MWF3 entwickelte Dialog zum Ändern eines `MediaItem` dargestellt werden. (1 P.)

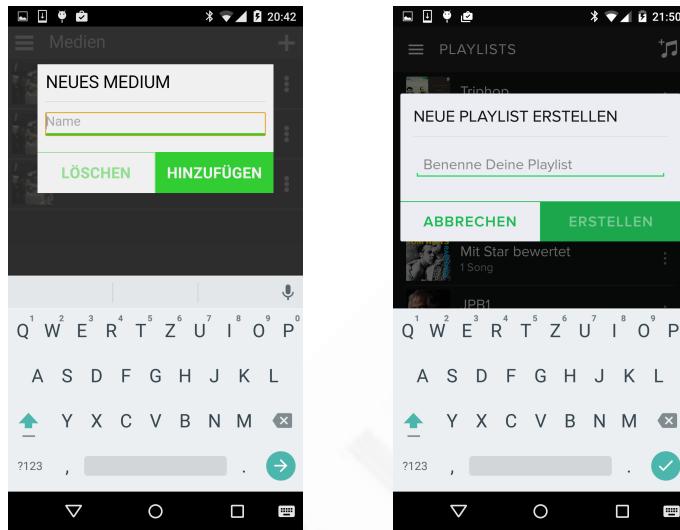
Bearbeitungshinweise

- Anforderung 5** In NJM2 werden Sie hier vor dem Löschen einen Rückbestätigungsdialog umsetzen.

Ü MWF3 Dialog zur Erstellung und Modifikation von Elementen (13 Punkte)

Aufgabe

Ermöglichen Sie die Erstellung und Modifikation von neuen Elementen durch einen Dialog mit Texteingabefeld.



Anforderungen

- Der Dialog soll über eine Titelzeile verfügen, in der je nach Anwendungsfall wahlweise der Text 'Neues Medium' oder 'Medium editieren' angezeigt wird. (**1 P.**)
- Der Dialog soll ein Texteingabefeld bereit stellen, das je nach Anwendungsfall entweder leer ist oder den Namen des zu editierenden `MediaItem` Objekts darstellt. (**1 P.**)
- Beim Öffnen des Dialogs soll das Texteingabefeld fokussiert werden, u.a. um ohne weitere Nutzerinteraktion die Anzeige der Tastatur auf mobilen Geräten zu veranlassen. (**1 P.**)
- Der Dialog soll außerdem über zwei Bedienelemente verfügen, die das Löschen bzw. Erstellen oder Modifizieren von `MediaItem` Objekten veranlassen. (**2 P.**)
- Das 'Löschen'-Element soll nur bei Editieren eines bestehenden Elements aktiv sein. (**1 P.**)
- Die Betätigung des 'Löschen'-Elements soll das dargestellte `MediaItem` aus der verwendeten Datenbank entfernen und die Listenansicht aktualisieren. (**2 P.**)
- Das Bedienelement zum Erstellen bzw. Modifizieren soll je nach Anwendungsfall unterschiedlich beschriftet sein. (**1 P.**)

8. Je nach Anwendungsfall soll bei Betätigung des Elements **Anforderung 7** entweder ein neues MediaItem erstellt bzw. ein bestehendes modifiziert und die Listenansicht entsprechend aktualisiert werden. (**3 P.**)
9. Bei Öffnen des Dialogs soll die Listenansicht über einen kurzen Zeitraum hinweg ausgeblendet werden, Click/Tap außerhalb des Dialogs schließt diesen. (**1 P.**)

Ü MWF4 Leseansicht

(11 Punkte)



Aufgabe

Ergänzen Sie die Anwendung um eine Ansicht, in der ein ausgewähltes `MediaItem` dargestellt wird.

Anforderungen

1. In der Kopfzeile der Leseansicht sollen neben dem ‘Sandwich’-Icon der Name des `MediaItem` Objekts und rechtsbündig ein ‘Papierkorb’-Icon dargestellt werden. (2 P.)
2. In der Fußzeile soll linksbündig ein ‘Zurück’-Icon angezeigt werden. (1 P.)
3. Der Hauptteil der Ansicht zeigt den Inhalt der Bildquelle des `MediaItem` über die gesamte Bildschirmbreite ohne Abschneiden des Bildes und ohne horizontales Scrolling, ggf. mit vertikalem Scrolling. (2 P.)
4. Geöffnet wird die Leseansicht bei Auswahl eines `MediaItem` in der Listenansicht. (1 P.)
5. Die Rückkehr zur Listenansicht wird durch Betätigung des ‘Zurück’-Icons veranlasst. (1 P.)
6. Der Übergang von der Listenansicht zur Leseansicht und zurück erfolgt unter kurzem Aus- und Einblenden der Ansichten. (1 P.)
7. Die Betätigung des ‘Papierkorb’-Icons veranlasst das Löschen des Icons aus der verwendeten Datenbank und die Rückkehr zur Listenansicht. **Hier und in allen folgenden Anforderungen, die eine ‘Rückkehr’ in eine Vorgängeransicht beschreiben, werden Punkte nur dann**

vergeben, falls die Rückkehr tatsächlich als solche umgesetzt wird. Die Verwendung von `nextView()` ist hierfür nicht zulässig..(2 P.)

8. Wird ein `MediaItem` in der Leseansicht gelöscht, dann ist dieses nach Rückkehr in die Listenansicht nicht mehr in der Liste enthalten. (1 P.)

3 Vorbeitung

Laden Sie das MWF Codegerüst aus Github.

- Starten Sie Webstorm.
- Wählen Sie die Option *Check out from Version Control*.
- Wählen Sie die Option *Git*.
- Geben Sie als *Git Repository URL* die folgende URL ein: `https://github.com/dieschnittstelle/org.dieschnittstelle.iam.mwf.skeleton.git`
- Wählen Sie ein *Parent Directory* und einen *Directory Name* Ihrer Wahl.
- Führen Sie *Clone* aus und lassen Sie Webstorm das Projekt öffnen.
- Wählen Sie in *Preferences → Languages & Frameworks → JavaScript* als *JavaScript Language Version ‘ECMAScript 6’* aus.

Das Git Projekt wird geklont und nach Bestätigung in WebStorm geöffnet.

Richten Sie eine NodeJS Run Konfiguration für das Projekt ein.

- Laden Sie NodeJS in einer für Ihren Entwicklungsrechner geeigneten Version herunter: <https://nodejs.org/en/download/>
- Installieren Sie NodeJS.
- Wählen Sie aus dem *Run* Menü die Option *Edit Configurations....*
- Wählen Sie nach Ausführung von + (*Hinzufügen*) die Option *Node.js*.
- Weisen Sie als *Node Interpreter* das `node` Executable aus dem `bin` Verzeichnis der NodeJS Installation zu, falls dieses Feld nicht schon vorausgefüllt ist.
- Weisen Sie der Konfiguration einen Namen Ihrer Wahl zu.
- Wählen Sie als *Working Directory* Ihr Projektverzeichnis.
- Wählen Sie als *JavaScript file* die Datei `webserver.js` im Projektverzeichnis.

Unter dem angegebenen Namen können Sie den NodeJS Server starten, der die Anwendung bereitstellt.

- Beenden Sie die Einrichtung mit *Apply* und schließen Sie das Fenster mit *Close*
- Weitere Hinweise finden Sie hier: <https://www.jetbrains.com/webstorm/help/node-js.html>

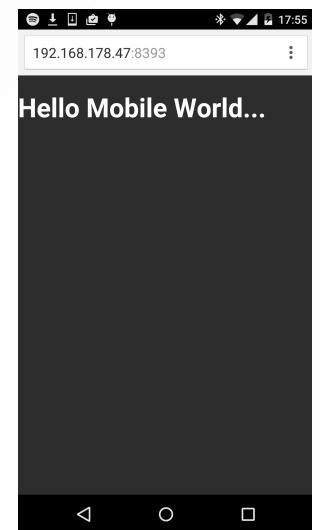
Starten Sie NodeJS.

- Wählen Sie aus dem *Run* Menü die Option *Run...*
- Wählen Sie die im vorangegangenen Schritt erstellte Konfiguration aus
 - nach erstmaliger Auswahl wird die Konfiguration direkt als Option im *Run* Menü angezeigt.

NodeJS wird gestartet und eine Konsole mit den Ausgaben des Prozesses geöffnet.

Greifen Sie mit dem Browser auf die Anwendung zu.

- Stellen Sie bei Verwendung von Firefox sicher, dass die Option ‘*Chronik anlegen*’ in den Datenschutzeinstellungen von Firefox aktiv ist.
- Klicken Sie auf die URL, die in der NodeJS Konsole als Link angezeigt wird.



4 Umsetzung der Listenansicht

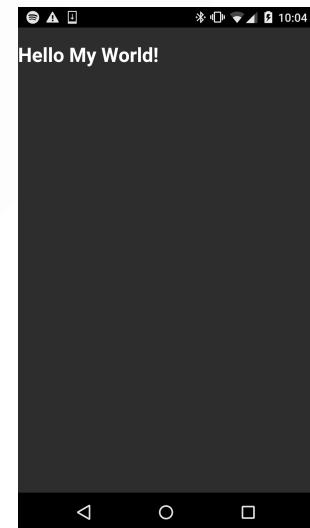
4.1 Gestaltung

Aktivieren Sie den Styling-Modus.

Den Styling-Modus können Sie verwenden, um die Ansichten Ihrer Anwendung zunächst statisch und unabhängig von der Anwendungssteuerung mit JavaScript zu gestalten. Zur Aktivierung markieren Sie die Ansicht, die Sie gestalten wollen, mit der Klasse `mwf-styling`. Beim Starten der Anwendung werden nur die Elemente, die diese Klasse besitzen, und ihre Kind-Elemente im DOM beibehalten. Alle anderen Elemente werden entfernt. Falls Sie nicht die Überlagerung von Ansichten mit Dialogen testen möchten, sollte nur ein `mwf-view` Element markiert werden.

[Datei: app.html]

```
<!--- ... --->
<body>
    <div class="mwf-view_mwf-view-initial_mwf-styling" data-
        -mwf-viewcontroller="MyInitialViewController">
        <h2>Hello My World!</h2>
    </div>
</body>
```

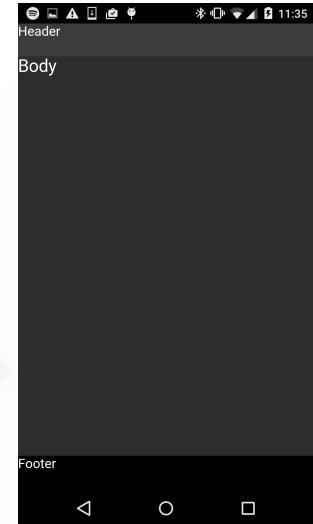


Erläuterungen:

- Als **Ansicht** wird eine vollständige Bildschirmseite bezeichnet, die bei mobilen Anwendungen üblicherweise aus einer Kopfleiste, einem Hauptbereich sowie ggf. einer Fußleiste besteht.
- Ansichten werden in `app.html` als Kind-Elemente von `<body>` deklariert.
- Ansichten werden als `<div>`-Elemente mit Klasse `mwf-view` deklariert.
- Innerhalb des `<div>` Wurzelements einer Ansicht kann grundsätzlich beliebiges HTML Markup verwendet werden.
- Der dynamische Aufbau von Ansichten, die Reaktion auf Nutzereingaben und ggf. der Übergang zu Folgeansichten oder zu vorangegangenen Ansichten wird durch **View Controller** gesteuert.
- Der Name des View Controllers einer Ansicht wird als Wert des Attributs `data-mwf-viewcontroller` angegeben.
- Im Styling-Modus werden View Controller nicht verwendet.

Deklarieren Sie Kopf- und Fußzeile sowie den Hauptbereich der Ansicht

```
<!-- ... -->
<div class="mwf-view_mwf-view-initial_mwf-styling" data-mwf-
    viewcontroller="MyInitialViewController">
    <header> Header</header>
    <main>
        Body
    </main>
    <footer>Footer</footer>
</div>
```

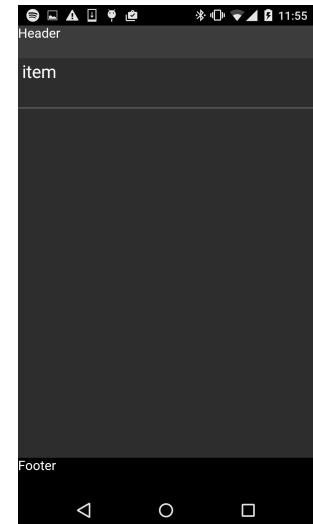


Erläuterungen:

- Kopf- und Fußzeile werden mit `<header>` bzw. `<footer>` markiert.
- Für den Hauptbereich der Ansicht wird ein `<main>`-Element verwendet.
- Die generischen Style-Zuweisungen für die drei Grundbestandteile einer Ansicht bewirken, dass diese bei fixer Höhe von Kopf- und Fußleiste zusammen immer exakt die verfügbare Anzeigefläche ausfüllen.

Fügen Sie die Listenansicht ein

```
<!-- ... -->
<main class="mwf-scrollbar">
    <ul class="mwf-listview">
        <li class="mwf-listitem">
            item
        </li>
    </ul>
</main>
```



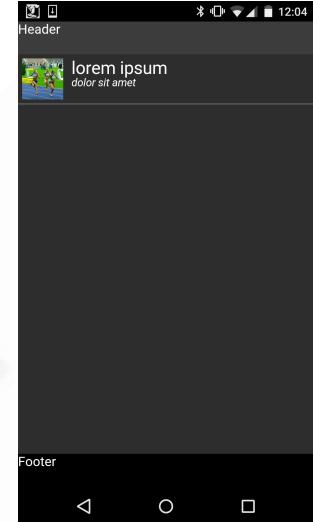
Erläuterungen:

- Für Listenansichten und einzelne Listenelemente werden die dafür vorgesehenen HTML-Tags `` und `` verwendet, die zusätzlich mit den Klassen `mwf-listview` bzw. `mwf-listitem` versehen werden.
- Die Zuweisung von `mwf-scrollbar` zum Eltern-Element der Listenansicht ermöglicht vertikales Scrolling der Liste, falls erforderlich.

Gestalten Sie die Ansicht für die einzelnen Listenelemente

Fügen Sie zunächst das Bildelement auf der linken Seite sowie recht daneben das Element für die Titel und Untertitel hinzu.

```
<!-- ... -->
<li class="mwf-listitem__mwf-li-title-subtitle">
    
    <div class="mwf-li-titleblock">
        <h2>lorem ipsum</h2>
        <h3>dolor sit amet</h3>
    </div>
</li>
```



Erläuterungen:

- Die Klasse `mwf-li-title-subtitle` wählt eine in Titel und Untertitel untergliederte Ansicht für Listenelemente aus.
- Das `<div>`-Element mit Klasse `mwf-li-titleblock` dient als Container-Element für Titel und Untertitel.
- Elemente, die als `mwf-left-align` platziert sind, werden am linken Rand ihres Eltern-Elements bzw. rechts neben einem bereits existierenden Geschwister-Element platziert.

Fügen Sie dann noch das Aktionsmenü-Element mit ‘Optionen’-Icon am rechten Rand des Listenelements hinzu.

```
<!-- ... -->
<li class="mwf-listitem__mwf-li-title-subtitle">
    <!-- ... -->
    <div class="mwf-li-titleblock__mwf-right-fill">
        <!-- ... -->
    </div>
    <button class="mwf-imgbutton mwf-img-options-vertical
        mwf-right-align">
        </button>
</li>
```



Erläuterungen:

- `<button>`-Elemente mit Klasse `mwf-imgbutton` dienen zur Darstellung von Icons.
- Die Auswahl des darzustellenden Icons erfolgt über die Zuweisung einer Klasse der Form `mwf-img-<iconname>`. Die Deklaration dieser Klassen finden Sie im Stylesheet `lib/mwf/css/mwfIcons.css`.

- Durch Zuweisung von `mwf-right-align` wird das betreffende Element am rechten Rand des umgebenden Eltern-Elements platziert bzw. rechts neben einem dort bereits platzierten Element. Falls das Eltern-Element, wie im hier vorliegenden Fall, außerdem am linken Rand ausgerichtete Elemente enthält, muss das letzte dieser Elemente mit der Klasse `mwf-right-fill` markiert werden.

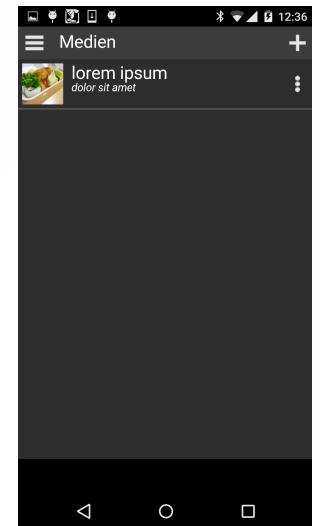
Wissenswertes:

- Eigene Icon-Klassen können Sie unter Verwendung der Icons in `lib/mwf/css/img/glyphicons/png` z.B. in `css/mystyle.css` deklarieren.

Gestalten Sie schließlich die Kopfleiste der Listenansicht.

Fügen Sie ein Hauptmenü-Element am linken Rand der Kopfleiste, eine daran angrenzende Überschrift sowie ein Aktionselement zum Hinzufügen neuer Inhalte am rechten Rand hinzu und leeren Sie die Fußleiste.

```
<!--- ... --->
<header>
  <button class="mwf-imgbutton mwf-img-sandwich
    mwf-left-align"></button>
  <h1 class="mwf-left-align mwf-right-fill">Medien</h1>
  <button class="mwf-imgbutton mwf-img-plus
    mwf-right-align"></button>
</header>
<!--- ... --->
<footer></footer>
```



Erläuterungen:

- Sie sehen hier weitere Verwendungsbeispiele für die Ausrichtungsklassen `mwf-left-align`, `mwf-right-fill` und `mwf-right-align` sowie für die Realisierung von Aktionselementen mit Icons mittels `mwf-imgbutton`.

Die visuelle Gestaltung der Listenansicht ist nun abgeschlossen. Im nächsten Abschnitt werden Sie die Ansicht auf Basis der in Ihrer Anwendung bereits vorhandenen und der durch den Nutzer neu erstellten Daten dynamisch aufbauen. Zuvor werden wir dafür noch das Datenmodell der Anwendung vorbereiten.

4.2 Datenmodell

Erstellen Sie einen JavaScript Objekt-Typ für Medieninhalte, d.h. die Bilder und Videos die in Ihrer Anwendung verwendet werden.

Medieninhalte werden durch die folgenden Attribute beschrieben:

- einen Titel
- eine Beschreibung
- ein Datum, das den Zeitpunkt bezeichnet, an dem Sie das Medium zu Ihrer Anwendung hinzugefügt haben.
- eine URL, die das Medium identifiziert
- einen Content-Typ, d.h. ob es sich um ein Bild oder Video handelt
- eine Content-Bereitstellungsart, d.h. ob das Medium durch einen externen Anbieter (z.B. <https://placeimg.com>) bereit gestellt wird oder ob Sie es selbst in die Anwendung geladen haben.

Deklarieren Sie zunächst in js/model/MyEntities.js die Klasse MediaItem als Unterklasse von Entity. Diese wird durch MWF definiert und stellt verschiedene generische Funktionen für die Umsetzung lesender und schreibender Operationen bereit.

[Datei: MyEntities.js]

```
// TODO-REPEATED: add new entity type declarations here
class MediaItem extends EntityManager.Entity {
    constructor() {
        super();
    }
}
```

Erläuterungen:

- Die in aktuellen Versionen von JavaScript verfügbare Deklarationsmöglichkeit von Objekt-Typen als ‘Klassen’ verbirgt die Details des in JavaScript angewendeten Mechanismus der **prototypenbasierten Vererbung** und suggeriert syntaktisch eine **klassenbasierte Vererbung**, wie sie z.B. in Java umgesetzt ist. Tatsächlich erfolgt die Vererbung aber nach wie vor durch Instantiierung eines Prototyp-Objekts pro Klasse, das über die durch die Klasse selbst und deren Oberklassen eingeführten Attribute und Methoden verfügt und mit dem alle Instanzen der Klasse assoziiert sind.
- Die `constructor()`-Methode kann wie in Java zur Instantiierung von Attributen eines Objekts verwendet werden (siehe den folgenden Schritt), der explizite Aufruf des Konstruktors der Superklasse mittels `super()` ist dafür erforderlich.

- Neben der `constructor()`-Methode können Klassen über beliebige weitere Methoden verfügen sowie spezifische Getter- und/oder Setter-Methoden für ausgewählte Attribute deklarieren (siehe unten).
- Alle für eine Klasse deklarierten Methoden sind *öffentlich*, d.h. können auf jeder Instanz der Klasse aufgerufen werden. Eine syntaktisch vergleichbar intuitive Möglichkeit zur Deklaration privater Methoden besteht derzeit nicht.
- Weitere Hinweise zu Klassen in JavaScript finden Sie hier: <https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes>

'Deklarieren' Sie dann die Attribute von `MediaItem` und weisen Sie ihnen die Werte der Konstruktorargumente bzw. Default-Werte zu.

```
constructor(name, src, contentType) {
  super();
  this.name = name;
  this.description = "";
  this.added = Date.now();
  this.src = src;
  this.srcType = null;
  this.contentType = contentType;
}
```

Erläuterungen:

- Bei Verwendung von Klassen können Attribute, wie hier umgesetzt, im Konstruktor durch Referenz auf die jeweilige Instanz der Klasse mittels `this` instantiiert werden. Zuvor muss der Aufruf von `super()` erfolgen. Eine explizite Deklaration der Attribute außerhalb des Konstruktors ist weder erforderlich, noch möglich.
- Alle Attribute einer Klasse sind *öffentlich*, d.h. können für jede Instanz der Klasse zugegriffen werden. Eine syntaktisch intuitive Möglichkeit zur Deklaration privater Attribute besteht nicht.
- Neben den im Konstruktor oder anderen Methoden einer Klasse gesetzten Attributen können für jede Instanz einer Klasse beliebige Attribute im Zuge der Verwendung der betreffenden Instanz gesetzt und ggf. ausgelesen werden.

- Das Attribut soll `srcType` dazu dienen, im Zuge der weiteren Bearbeitung des Übungsprogramms für ein `MediaItem` festzuhalten, ob es durch Angabe einer externen URL oder mittels Dateiupload erstellt wurde. Im Ggs. dazu drückt `contentType` aus, ob es sich bei dem betreffenden `MediaItem` um ein Bild oder ein Video handelt.

Deklarieren Sie dann Getter-Methoen für zwei weitere, abgeleitete, Attribute von `MediaItem`.

```
class MediaItem extends EntityManager.Entity {

    /* ... */

    get addedDateString() {
        return (new Date(this.added)).toLocaleDateString();
    }

    get mediaType() {
        if (this.contentType) {
            var index = this.contentType.indexOf("/");
            if (index > -1) {
                return this.contentType.substring(0, index);
            }
            else {
                return "UNKNOWN";
            }
        }
        else {
            return "UNKNOWN";
        }
    }
}
```

Erläuterungen:

- Abgeleitete Attribute sind Attribute, deren Werte auf Basis der Werte anderer Attribute ermittelt werden.
- In JavaScript Klassen werden abgeleitete Attribute durch das Schlüsselwort `get` gefolgt von einer argumentslosen Funktion mit dem Namen des Attributs deklariert.
- Im obigen Beispiel werden die beiden Getter-Methoden für jede Instanz `mi` von `MediaItem` bei Verwendung der Ausdrücke `mi.addedDateString` bzw. `mi.mediaType` aufgerufen.

- Analog zu Getter-Methoden können Setter-Methoden durch Verwendung des Schlüsselworts `set` gefolgt von einer einargumentigen Funktion mit dem Namen des zuzugreifenden Attributs deklariert werden.

Lassen Sie `MediaItem` durch `MyEntities` exportieren.

```
// TODO-REPEATED: do not forget to export all type declarations
return {
    MyEntity: MyEntity, /*!!!*/
    MediaItem: MediaItem
}
```

Erläuterungen:

- Die Hinzufügung zur `return`-Anweisung des `MyEntities` Moduls ist erforderlich, damit Sie `MediaItem` in anderen Modulen Ihrer Anwendung verwenden können.

Bereiten Sie die lokale IndexedDB Datenbank Ihrer Anwendung für die Speicherung von `MediaItem` Instanzen vor.

Greifen Sie mit dem Browser auf Ihre Anwendung zu und führen Sie in der Entwickler-Konsole die folgende Anweisung aus, um die bestehende Datenbank zu löschen – diese wurde beim erstmaligem Zugriff auf die Anwendung erstellt:

```
window.indexedDB.deleteDatabase("mwftutdb")
```

[Datei: `MyApplication.js`]

```
// TODO-REPEATED: add new entity types to the array of object
// store names
GenericCRUDImplLocal.initialiseDB("mwftutdb", 1, ["MyEntity",
    "MediaItem"], () => {
    /*...*/
});
```

Beim nächsten Zugriff auf die Anwendung wird in der lokalen Datenbank ein **Object Store** für Instanzen von `MediaItem` erstellt.

Erläuterungen:

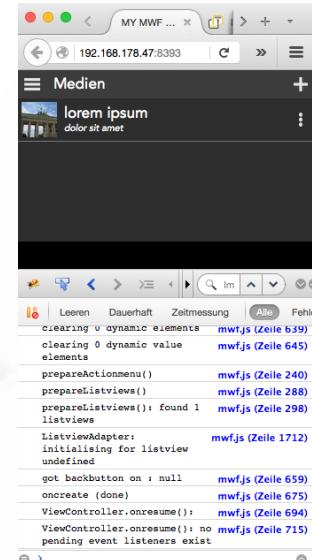
- MWF sieht vor, dass für jeden `Entity`-Typ einer Anwendung ein eigener Object Store in IndexedDB vorliegt, in dem die Instanzen des Typs persistiert werden.

Deaktivieren Sie den Styling-Modus

Entfernen Sie die Klasse `mwf-styling` aus dem `class`-Attribut der Listenansicht und öffnen Sie die Entwicklerkonsole des Browsers. Wenn Sie nun auf die Anwendung zugreifen, sollten in der Entwicklerkonsole die nebenan gezeigten Meldungen dargestellt werden.

[Datei: app.html]

```
<!--- ... --->
<div class="mwf-view mwf-view-initial mwf-styling" data-mwf-
    viewcontroller="MyInitialViewController">
    <!--- ... --->
</div>
```



4.3 Steuerung

Die nachfolgenden Abschnitte setzen die Steuerung der Listenansicht schrittweise um und führen dafür z.T. vorläufige Lösungen ein. Beispielsweise setzen wir den View Controller für die Listenansicht zunächst mit hardcodierten Daten um und werden diese Version im weiteren Verlauf durch eine Implementierung mit IndexedDB als lokaler Datenbank und in der Endausbaustufe durch Verwendung des MWF Entity Managers ersetzen.

4.3.1 Erstellen eines neuen View Controller

Erstellen Sie eine neue View Controller Implementierung für die Listenansicht.

- Erstellen Sie im Verzeichnis `controller` eine Kopie der Datei `ViewControllerTemplate.js`
- Benennen Sie die Kopie um in `ListviewViewController.js`
- Ersetzen Sie in `ListviewViewController.js` alle Vorkommen von `ViewControllerTemplate` durch `ListviewViewController`.

[Datei: ListviewViewController.js]

```
define(["mwf", "entities"], function(mwf, entities) {

    class ListviewViewController extends mwf.ViewController {
        /*...*/
    }
}
```

```

    // and return the view controller function
    return ListviewViewController;
}) ;

```

Erläuterungen:

- Alle anwendungsspezifischen Controller-Klassen erben die durch das MWF Framework bereitgestellte Klasse `ViewController`. Diese stellt u.a. die **Lebenszyklusmethoden** `oncreate()`, `onresume()`, `onpause()` sowie weitere verallgemeinerbare Funktionalität für die Steuerung der Ansichten einer Anwendung bereit, z.B. Methoden für den Aufbau und die Modifikation von Listenansichten.

Registrieren Sie `ListviewViewController.js` als Modul.

[Datei: `Main.js`]

```

requirejs.config({
    baseUrl: "",

    // TODO-REPEATED: add all new modules here, e.g.
    // ViewController implementations
    paths: {
        /* ... */

        /* application libraries: view controllers */
        MyInitialViewController: 'js/controller/
        MyInitialViewController', /*!!!!*/
        ListviewViewController:
        'js/controller/ListviewViewController'
    }
}) ;

```

Geben Sie das `ListviewViewController` Modul als Dependency beim Start der Anwendung an

[Datei: `Main.js`]

```

// TODO-REPEATED: add new ViewControllers to the dependency
// array
requirejs(["mwf", "GenericDialogTemplateViewController", "
    MyApplication", "MyInitialViewController",
    "ListviewViewController"],
    function(mwf) {
        mwf.onloadApplication();
    }) ;

```

Erläuterungen:

- Da die anwendungsspezifischen View Controller dynamisch durch die generische Implementierung des MWF Frameworks instantiiert werden, müssen sie zuvor in einem anwendungsspezifischen Modul bzw. in `Main.js` als Dependency deklariert werden. Andernfalls sind die View Controller Module nicht geladen, und die Instantiierung schlägt fehl.

Vergeben Sie eine innerhalb von `app.html` eindeutige ID für die Ansicht und weisen Sie ihr `ListviewViewController` als Controller zu.

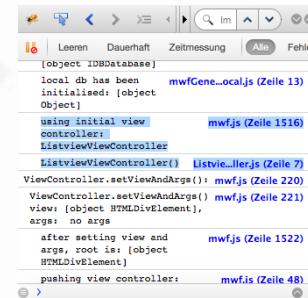
[Datei: `app.html`]

```
<div class="mwf-view_mwf-view-initial" id="mediaOverview" data-  
    mwf-viewcontroller="ListviewViewController">  
    <!-- ... -->  
</div>
```

Erläuterungen:

- Unter der angegebenen `id` kann die Ansicht innerhalb der Anwendung als Ziel eines Ansichtsübergangs identifiziert werden.

Beim erneuten Zugriff auf die Anwendung sollten Sie in den Logmeldungen der Konsole ein Log des Konstruktorauftrags von `ListviewViewController()` finden.



4.3.2 Befüllung der Listenansicht

Markieren Sie das in `app.html` erstellte Listenelement als Template und assoziieren Sie es mit der Listenansicht.

[Datei: `app.html`]

```
<!-- ... -->  
<ul class="mwf-listview"  
    data-mwf-listitem-view="myapp-listitem">  
    <li class="mwf-listitem_mwf-li-title-subtitle_mwf-template"  
        data-mwf-templatename="myapp-listitem">  
        <!-- ... -->  
    </li>  
</ul>
```

Erläuterungen:

- **Templates** sind wieder verwendbare Bestandteile einer Ansicht, die von View Controller Implementierungen mit dynamischen Inhalten befüllt werden können.
- Die Assoziation eines Templates für Listenelemente mit einer Listenansicht erfolgt über die Attribute `data-mwf-template-name` auf `` und `data-mwf-listitem-view` auf ``.
- Wird ein View Controller instantiiert, durchsucht MWF die von diesem verwendete Ansicht nach Listenansichten und bereitet diese für die Befüllung durch den View Controller vor.

Erstellen Sie im Konstruktor des View Controllers Dummy-Daten zur Befüllung der Ansicht und deklarieren Sie Attribute für die spezifischen Bedienelemente der Ansicht.

[Datei: ListviewViewController.js]

```
/*....*/
class ListviewViewController extends mwf.ViewController {

    constructor() {
        super();
        console.log("ListviewViewController() ");

        this.items = [
            new
                entities.MediaItem("m1", "https://placeimg.com/100/100/city"),
            new
                entities.MediaItem("m2", "https://placeimg.com/200/150/music"),
            new
                entities.MediaItem("m3", "https://placeimg.com/150/200/culture")
        ];

        this.addNewMediaItem = null;
    }

    /*....*/
}
```

Erläuterungen:

- Das Attribut `addNewMediaItem` wird später mit einer Referenz auf das ‘Hinzufügen’ Bedienelement instantiiert werden.
- Eine solche Deklaration als Attribut ist insbesondere für Bedienelemente erforderlich, die in mehr als einer Methode des Controllers verwendet

werden. Für andere Bedienelemente kann die explizite Deklaration als Konvention angesehen werden, die einen Überblick über alle Elemente einer Ansicht ermöglicht, für die der Controller zuständig ist. In Android ist eine solche Deklaration als Konvention üblich, in iOS ist sie erforderlich, um die Möglichkeit der durch das Framework ermöglichten automatischen Instantiierung von Bedienelementen im Controller nutzen zu können.

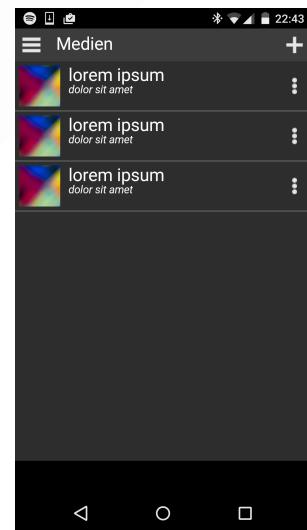
- Eine Entkopplung des Controllers von den spezifischen Bedienelementen einer Ansicht wird in MWF durch die Nutzung von *Ractive.js* Templates und die durch letztere unterstützte Ereignisbehandlung möglich, siehe dafür Abschnitt 4.4.4 unten.

Befüllen Sie die Listenansicht mit den Dummy-Daten in der Implementierung der `oncreate()` Funktion.

[Datei: `ListviewViewController.js`]

```
/*...*/
class ListviewViewController extends mwf.ViewController {
    /*...*/
    oncreate(callback) {
        // TODO: do databinding, set listeners, initialise the
        // view
        this.initialiseListview(this.items);

        // call the superclass once creation is done
        super.oncreate(callback);
    }
}
```



Nach erneutem Zugriff auf die Anwendung sollte die Listenansicht mit drei – identischen – Elementen angezeigt werden. *Erläuterungen:*

- Der Aufruf von `initialiseListview()` führt dazu, dass die Listenansicht mit den Elementen des dem Aufruf übergebenen Arrays neu aufgebaut wird. Für jedes Element des Arrays wird durch MWF eine neue Instanz des Tempates für die Listenelemente erstellt. Die Befüllung des Templates mit den Daten des darzustellenden Elements muss durch die Anwendung erfolgen.
- Bei `oncreate()` handelt es sich um eine **Lebenszyklusfunktion**, die MWF auf View Controller Instanzen unmittelbar nach Erzeugen einer neuen Instanz und noch vor Darstellung der durch den View Controller gesteuerten Ansicht auuft. Weitere Lebenszyklusfunktionen sind nach-

folgend beschrieben:

Funktion	Aufruf durch MWF
oncreate()	Nach Erzeugen einer View Controller Instanz
onresume()	Bei erstmaliger und wiederholter Darstellung/‘Einblenden’ der durch die View Controller Instanz gesteuerten Ansicht
onpause()	Bei erstmaligem und wiederholtem Ausblenden der Ansicht
onstop()	Bei Entfernen der Ansicht.

- Lebenszyklusfunktionen wird grundsätzlich eine Callback-Funktion übergeben, die nach Abschluss der Lebenszyklusfunktion aufgerufen werden muss, um die Darstellung bzw. das Ausblenden der betreffenden Ansicht abzuschließen.
- Anwendungsspezifische Lebenszyklusfunktionen müssen am Ende die betreffenden Funktionen des Obertyps aufrufen und dieser die Callback-Funktion übergeben, z.B. `super.onCreate(callback)` im hier behandelten Beispiel.

Befüllen Sie die einzelnen Elemente der Listenansicht mit den Attributen des darzustellenden Listenelements.

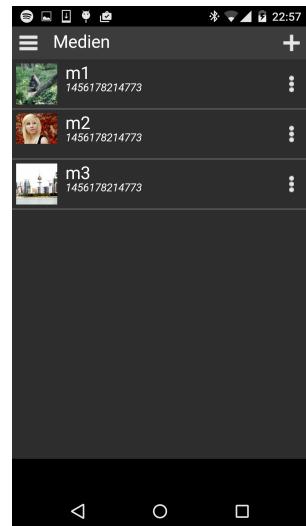
Implementieren Sie dafür die Methode `bindListView()`, die im Codegerüst des View Controllers vorgesehen ist:

[Datei: `ListviewViewController.js`]

```
/*...*/
bindListView(viewid, itemview, item) {
    // TODO: implement how attributes of item shall be displayed
    // in itemview
    itemview.root.getElementsByTagName("img")[0].src = item.src;
    itemview.root.getElementsByTagName("h2")[0].textContent =
        item.name;
    itemview.root.getElementsByTagName("h3")[0].textContent =
        item.added;
}
```

Erläuterungen:

- `bindListView()` wird für jedes Element des in der Listenansicht darzustellenden Arrays aufgerufen.
- Übergeben wird jeweils eine für das betreffende Element erzeugte Instanz des Templates für Listenelemente (`itemview.root`) sowie das darzustellende Element des Arrays (`item`).



- Übergeben wird außerdem die ID der Listenansicht – diese muss nur berücksichtigt werden, falls eine Ansicht mehr als eine Listenansicht verwendet.
 - Die Befüllung der Listenelemente kann durch Zugriff auf dessen DOM Struktur und Setzen der geeigneten Attribute der DOM Elemente erfolgen.
-

4.3.3 Auswahl von Listenelementen

Reagieren Sie auf die Auswahl von Listenelementen durch die Nutzer der Anwendung.

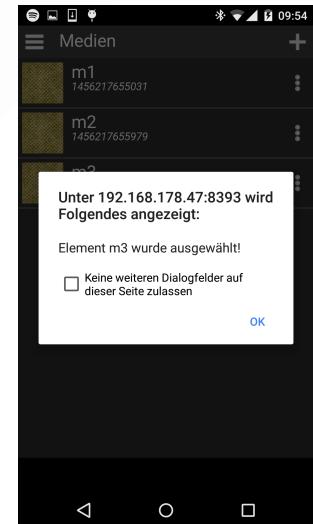
Als Reaktion werden wir hier nur einen `alert`-Dialog mit dem Namen des ausgewählten Elements anzeigen. Weiter unten werden wir einen Übergang zu einer ‘Leseansicht’ für MediaItem umsetzen. Implementieren Sie die Funktion `onListItemSelected()`, die im Codegerüst für View Controller bereits vorgesehen ist:

[Datei: ListviewViewController.js]

```
/*...*/
onListItemSelected(listitem, listview) {
    // TODO: implement how selection of listitem shall be
    // handled
    alert("Element " + listitem.name + " wurde ausgewählt!");
}
```

Erläuterungen:

- Bei Auftreten eines `onclick` Ereignisses auf einem Element der Listenansicht ruft MWF auf dem View Controller, der die Steuerung der Listenansicht vornimmt, die Funktion `onListItemSelected()` auf.
- Übergeben wird der Funktion das Objekt, das im ausgewählten Listenelement dargestellt wird, sowie die ID der Listenansicht – wie bereits erwähnt, muss diese nur berücksichtigt werden, wenn eine Ansicht mehr als eine Listenansicht enthält.



4.3.4 Hinzufügen eines neuen Elements

Ermöglichen Sie das Hinzufügen eines neuen Elements zur Liste

Dafür lesen wir das betreffende Bedienelement aus und setzen darauf einen `onlick`-Listener. Dieser generiert zunächst ein hardcodiertes `MediaItem` Objekt, wobei sich die Objekte hinsichtlich des Werts des `added` Attributs unterscheiden werden.

Weisen Sie dem Bedienelement zunächst eine geeignete ID zu.

[Datei: app.html]

```
<!-- ... -->
<header>
    <!-- ... -->
    <button class="mwf-imgbutton_mwf-img-plus_mwf-right-align"
        id="addNewMediaItem"></button>
</header>
```

Instantiiieren Sie das Bedienlement in `oncreate()`.

[Datei: ListviewViewController.js]

```
/*...*/
oncreate(callback) {
    // TODO: do databinding, set listeners, initialise the view
    this.addNewMediaItemElement =
        this.root.querySelector("#addNewMediaItem");

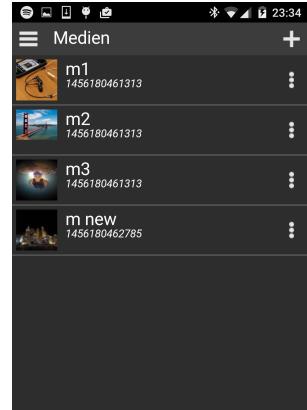
    this.initialiseListview(items);
    // call the superclass once creation is done
    super.oncreate(callback);
}
```

Erläuterungen:

- Auf die durch einen View Controller gesteuerte Ansicht kann in den Lebenszyklusfunktionen grundsätzlich via `this.root` zugegriffen werden, z.B. um Bedienelemente der Ansicht auszulesen.
- Der Zugriff auf `this.root` ist erst ab dem Aufruf von `oncreate()` durch MWF möglich.

Weisen Sie dem Bedienelement einen `onclick`-Listener zu.

Der Listener soll ein neues `MediaItem` Objekt erzeugen und es zur Listenansicht hinzufügen.



```

/*
oncreate(callback) {
    // TODO: do databinding, set listeners, initialise the view
    this.addNewMediaItemElement = this.root.querySelector("#
        addNewMediaItem");
    this.addNewMediaItemElement.onclick = () => {
        this.addToListview(new entities.MediaItem("m_new", "https
            ://placeimg.com/100/100/city"));
    });
/*
}

```

Erläuterungen:

- Für die Hinzufügung eines neuen Elements zur Listenansicht steht die Funktion `addToListview()` zur Verfügung.
- Auch für dieses neue Element wird vor Darstellung in der Liste zunächst die oben implementierte Funktion `bindListView()` aufgerufen.
- Die Notation des Event Listener als **Pfeilfunktion** (engl. *Arrow-Function*) ist eine Notationsform für anonyme, am Ort ihrer Verwendung deklarierte Funktionen, die eine Alternative zur Deklaration mittels `function` Keyword darstellt. Eine Funktion mit n Argumenten `function(arg1, ..., argn) { /*...*/ }` wird als Pfeilfunktion wie folgt notiert: `(arg1, ..., argn)=>{...}`
- Pfeilfunktionen zeichnen sich im Ggs. zur Notation mit `function` Keyword insbesondere dadurch aus, dass sie keinen eigenen Kontext für die Interpretation des Keywords `this` Keywords bilden, das auch im obigen Codeausschnitt verwendet wird. Innerhalb einer Pfeilfunktion hat `this` denselben Wert, den es außerhalb der Funktion hat, oben ist das die Instanz von `ListviewViewController`, auf der die `oncreate()` Funktion aufgerufen wurde, die den Event Listener setzt. Würde der Listener mit `function`-Keyword deklariert, müsste diese Instanz explizit durch Aufruf von `bind()` an den Listener gebunden werden, damit der Aufruf von `addToListview()` möglich ist, d.h.: `function() { /*...*/ }.bind(this)`
- Die äußeren runden Klammern um die Deklaration der Pfeilfunktion `((=>{ /*...*/ }))` dienen hier und im folgenden der expliziten Markierung der Funktion und müssen nicht notwendigerweise verwendet werden.
- Pfeilfunktionen sind Lambda-Ausdrücke, wie sie mittlerweile mit nahezu identischer Syntax auch in Java verfügbar sind – dort werden sie mit einem einfachen Pfeil (`->`) anstelle eines Doppelpfeils (`=>`) notiert.

- Mehr zu Pfeilfunktionen in JavaScript finden Sie hier: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions

4.3.5 Verwendung von IndexedDB

Ermöglichen Sie es dem Nutzer zunächst, die Datenbank jederzeit wieder zurückzusetzen.

Dafür verwenden wir das im vorliegenden Tutorial und den weiter führenden Übungsaufgaben nicht genutzte ‘Hauptmenü’-Element, das wir bereits zur Listenansicht hinzugefügt haben. Markieren Sie das Bedienelement in geeigneter Weise:

[Datei: app.html]

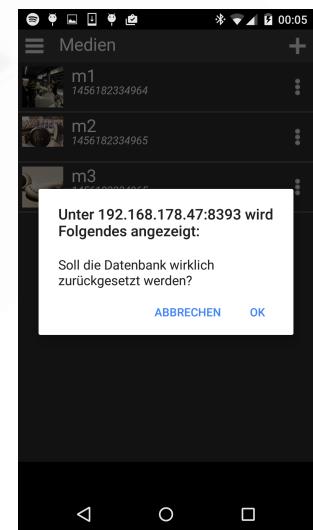
```
<!-- ... -->
<header>
    <button class="mwf-imgbutton_mwf-img-sandwich_mwf-left-align"
        " id="resetDatabase"></button>
    <!-- ... -->
</header>
```

Lesen Sie das Element im View Controller aus:

[Datei: ListviewViewController.js]

```
/*...*/
constructor() {
    super();
    this.resetDatabaseElement = null;
}

oncreate(callback) {
    /*...*/
    this.resetDatabaseElement =
        this.root.querySelector("#resetDatabase");
}
```



Weisen Sie einen geeigneten onclick-Listener zu:

```
oncreate(callback) {
    /*...*/
    this.resetDatabaseElement.onclick = () => {
        if (confirm("Soll die Datenbank wirklich zurückgesetzt werden?")) {
            indexedDB.deleteDatabase("mwftutdb");
        }
    });
}
```

Importieren Sie das Modul `GenericCRUDImplLocal` in `ListviewViewController`.

[Datei: `ListviewViewController.js`]

```
/*...*/
define(["mwf", "entities", "GenericCRUDImplLocal"], function(mwf,
    entities, GenericCRUDImplLocal) {
    /*...*/
})
```

Erläuterungen:

- `GenericCRUDImplLocal` ist ein generischer Wrapper um die IndexedDB API, der die CRUD Operationen `read()`/`readAll()`, `create()`, `update()` und `delete()` bereit stellt und für beliebige Objekt-Typen instantiiert werden kann. Voraussetzung für die erfolgreiche Ausführung der Operationen ist die Existenz eines Object Stores für den betreffenden Objekttyp.
- Den Object Store für `MediaItem` haben Sie oben bereits in `MyApplication.js` deklariert.

Instantiiieren Sie `GenericCRUDImplLocal` für `MediaItem`.

[Datei: `ListviewViewController.js`]

```
/*...*/
class ListviewViewController extends mwf.ViewController {

    constructor() {
        super();
        /*...*/

        this.crudops =
            GenericCRUDImplLocal.newInstance("MediaItem");
    }

    oncreate(callback) {
        /*...*/
    }
}
```

Das `items`-Attribut, das wir oben hardcodiert mit einem Array von `MediaItem` Elementen instantiiert haben, können Sie aus dem Code löschen.

Befüllen Sie die Listenansicht mit dem Resultat der `readAll()` Methode.

```

/*...*/
oncreate(callback) {

    /*...*/
    this.crudops.readAll((items) => {
        this.initialiseListview(items);
    });
    /*...*/
}

```

Erläuterungen:

- Alle CRUD Operationen auf IndexedDB werden asynchron ausgeführt und erfordern die Übergabe einer Callback-Funktion, der das Ergebnis der Operation übergeben wird.
- GenericCRUDImplLocal als Wrapper um die IndexedDB API erfordert daher ebenfalls die Übergabe von Callback-Funktionen zur Entgegennahme des Ergebnisses einer Operation.

Rufen Sie für die Erstellung eines neuen MediaItem Elements die create() Methode auf und aktualisieren Sie die Listenansicht.

```

/*...*/
oncreate(callback) {

    /*...*/
    this.addNewMediaItemElement.onclick = () => {
        this.crudops.create(new entities.MediaItem("m", "https://
            placeimg.com/100/100/city"), ((created) =>
        {
            this.addToListview(created);
        })
    );
    /*...*/
}

```

Erläuterungen:

- Auch create() erfordert die Übergabe einer Callback-Funktion, der das neu erstellte Objekt als Argument übergeben wird. Dieses unterscheidet sich von dem Objekt, das an create() übergeben wurde, durch die Zuweisung eines _id Attributs, das die Identifikation des Objekts in der Datenbank bzw. im verwendeten Object Store, erlaubt.

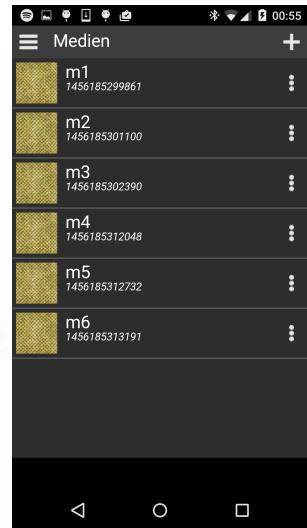
Modifizieren Sie die Darstellung von `MediaItem` Objekten der Listenansicht geringfügig.

Stellen Sie den Namen des Objekts gefolgt von dessen `_id` dar:

```
/*...*/  
bindListItemView(viewid, itemview, item) {  
    /*...*/  
    itemview.root.getElementsByTagName("h2")[0].textContent =  
        item.name+item._id;  
    /*...*/  
}
```

Fügen Sie die `_id` des Objekts auch der bei Auswahl eines Listenelements angezeigten Meldung hinzu:

```
/*...*/  
onListItemSelected(listitem,listview) {  
    // TODO: implement how selection of listitem shall be  
    // handled  
    alert("Element " + listitem.name +  
        listitem._id + " wurde ausgewählt!");  
}
```



4.4 Fortgeschrittene Steuerungsfunktionen

4.4.1 Verwendung von *Ractive.js* Templates

Als Alternative zum manuellen **Data Binding**, bei dem die Darstellung von Instanzen des Datenmodells in den Ansichten einer Anwendung – und umgekehrt – mittels Zugriff auf die DOM Elemente der Ansicht implementiert wird, ermöglicht MWF die Verwendung der Template Engine *Ractive.js* (<http://ractive.js.org/>). Nachfolgend werden wir zunächst die Implementierung der Data Binding Funktion `bindListItemView()` durch die Verwendung von Ractive.js Templates ersetzen und im folgenden weitere Ansichten der Anwendung mit Ractive.js umsetzen. Siehe zur Dokumentation der Ausdrucksmitte von Ractive.js die o.g. Website, insbesondere die dort verfügbaren Tutorials: <https://ractive.js.org/tutorials/hello-world/>. *Markieren Sie das Template für die Elemente der Listenansicht als Template, für das Ractive.js Data Binding angewendet werden soll*

[Datei: app.html]

```
<!-- ... -->  
<ul class="mwf-listview" data-mwf-listitem-view="myapp-listitem  
    ">
```

```
<li class="mwf-listitem_mwf-li-title-subtitle_mwf-template_mwf-databind" data-mwf-templatename="myapp-listitem">  
/</ul>
```

Erläuterungen:

- Elemente, die zusätzlich zur Klasse `mwf-template` die Klasse `mwf-databind` enthalten, können Ractive.js Expressions verwenden.
- Für Elemente von Listenansichten wird die Befüllung der Templates automatisch durch MWF veranlasst, für andere Anwendungsfälle siehe die Umsetzung der Leseansicht weiter unten.

Verwenden Sie innerhalb des Templates Ractive.js Expressions, um auf die Attribute des darzustellenden `MediaItem` Objekts zuzugreifen.

```
<li class="mwf-listitem_mwf-li-title-subtitle_mwf-template_mwf-databind" data-mwf-templatename="myapp-listitem">  
      
    <div class="mwf-li-titleblock">  
        <h2>{{name}} {{_id}}</h2>  
        <h3>{{added}}</h3>  
    </div>  
    <!-- ... -->  
</li>
```

Erläuterungen:

- MWF übergibt zur Befüllung des Templates das darzustellende Objekt an Ractive.js, d.h. in den Expressions können wir direkt auf die Attribute dieses Objekts zugreifen.

Entfernen Sie jetzt die Funktion `bindListItemView()` aus Ihrer View Controller Implementierung oder kommentieren Sie sie aus.

[Datei: `ListviewViewController.js`]

```
/*...*/  
bindListItemView(viewid, itemview, item) {  
    /*...*/  
}
```

Die Darstellung der Listenansicht sollte sich gegenüber dem vorherigen Entwicklungsstand nicht ändern.

4.4.2 Aktionsmenü für Listenelemente

Bauen Sie das Aktionsmenü zunächst als Bestandteil der Ansicht Ihrer Anwendung in `app.html` auf.

Verwenden Sie für das Menü ein `<div>` Element als Schwester-Element der `mediaOverview` Ansicht. [Datei: `app.html`]

```
<!-- ... -->
<div class="mwf-view_mwf-view-initial" id="mediaOverview" data-
    mwf-viewcontroller="ListviewViewController">
    <!-- ... -->
</div>
<div data-mwf-templateName="mediaItemMenu"
    class="mwf-listitem-menu mwf-template mwf-databind
    mwf-dialog mwf-popup">

</div>
<!-- ... -->
```

Erläuterungen:

- Die Klasse `mwf-listitem-menu` markiert das `<div>` Element als Menü für Listenelemente im Sinne von MWF.
- Die Klassen `mwf-template` und `mwf-databind` zeigen an, dass für die Darstellung des Menüs Ractive.js verwendet werden soll.
- Mittels `mwf-dialog` und `mwf-popup` wird bewirkt, dass das Menü als Popup-Dialog realisiert wird.
- Das Menü könnte auch als Kind-Element der `mediaOverview` Ansicht deklariert werden. Die hier vorgeschlagene Realisierung als Schwester-Element erleichtert uns aber das Testen im Styling-Modus.

Fügen Sie dem Menü eine Überschrift und die auszuwählenden Aktionen hinzu.

```
<!-- ... -->
<div data-mwf-templateName="mediaItemMenu" class="mwf-listitem-
    menu_mwf-databind_mwf-template_mwf-dialog_mwf-popup">
    <header>
        <h2 class="mwf-dyncontent ">{ {name} } { {_id} } </h2>
    </header>
    <main>
        <ul>
```

```

        <li class="mwf-li-singletitle_mwf-menu-item">Löschen</li>
        <li class="mwf-li-singletitle_mwf-menu-item">Editieren</li>
    </ul>
</main>
</div>

```

Erläuterungen:

- Die Verwendung eines `<header>` Elements in Verbindung mit einem `<main>` Element bewirkt innerhalb eines als `mwf-dialog` markierten Elements eine Darstellung des Menü-Dialogs mit Kopfzeile und Hauptbereich gemäß den Anforderungen. Eine alternative Dialog-Gestaltung werden wir unten umsetzen.
- Anhand der Klasse `mwf-li-singletitle` wird die Gestaltung von Listenelementen mit einfacherem Titel (im Ggs. zur Verwendung von Titel und Untertitel) für die auszuwählenden Optionen übernommen.
- `mwf-menu-item` markiert die Elemente der Liste als Menüoptionen.
- Die Klasse `mwf-dyncontent`, die auf dem `<h2>` Element gesetzt ist, bewirkt, dass überlanger Text ausgepunktet wird.

Testen Sie die Darstellung des Menüs

Weisen Sie dafür der Listenansicht und dem Menü-Element jeweils die Klasse `mwf-styling` zu und fügen Sie außerdem die Klassen `mwf-dialog-shown` bzw. `mwf-shown` hinzu.

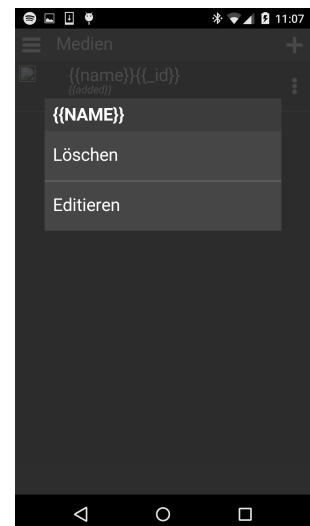
```

<!— ... —>
<div class="mwf-view_mwf-view-initial_mwf-styling"
     mwf-dialog-shown" data-mwf-viewcontroller="ListviewViewController" id="mediaOverview">
    <!— ... —>
</div>
<div data-mwf-templateName="mediaItemMenu" class="mwf-listitem-
    menu_mwf-databind_mwf-template_mwf-dialog_mwf-popup_
    mwf-styling
    mwf-shown">
    <!— ... —>
</div>

```

Erläuterungen:

- `mwf-shown` bewirkt die Darstellung des Dialogs.



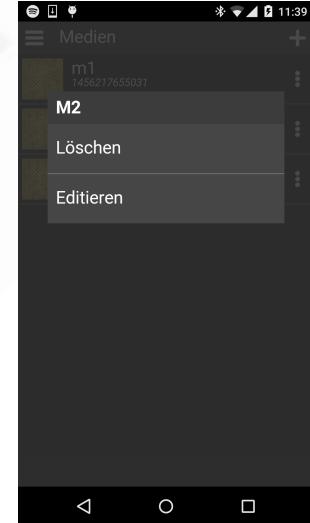
- `mwf-dialog-shown` auf der Ansicht, die den Dialog verwendet, veranlasst deren partielles Ausblenden.

Entfernen Sie für die weiteren Entwicklungsmaßnahmen nun die im letzten Schritt hinzugefügten Klassen `mwf-styling`, `mwf-shown` und `mwf-dialog-shown`!

Assoziieren Sie das Menü mit der Listenansicht und verwenden Sie das bereits vorhandene ‘Optionen’-Bedienelement in der Ansicht der Listenelemente als Element zum Öffnen des Menüs.

```
<!-- ... -->
<div class="mwf-view_mwf-view-initial_mwf-styling_mwf-dialog-
    shown" data-mwf-viewcontroller="ListviewViewController" id=
    "mediaOverview">
<!-- ... -->
<main class="mwf-scrollview">
    <ul class="mwf-listview" data-mwf-listitem-view="myapp-
        listitem" data-mwf-listitem-menu="mediaItemMenu">
        <li class="mwf-listitem_mwf-li-title-subtitle_mwf-
            template_mwf-databind" data-mwf-templateName="

            myapp-listitem">
            <!-- ... -->
            <button class="mwf-imgbutton_mwf-img-options-
                vertical_mwf-right-align_
                mwf-listitem-menu-control"></button>
        </li>
    </ul>
</main>
<!-- ... -->
</div>
```



Erläuterungen:

- Die Zuweisung der beiden Klassen bewirkt, dass bei Betätigung des Menü-Bedienelements der Menü-Dialog für das betreffende Listenelement dargestellt wird.

Implementieren Sie im View Controller zwei öffentliche Funktionen für die im Menü angebotenen Aktionen.

[Datei: `ListviewViewController.js`]

```
/*...*/
class ListviewViewController extends mwf.ViewController {
```

```

/*...*/
deleteItem(item) {
    this.crudops.delete(item._id, () => {
        this.removeFromListview(item._id);
    }));
}

editItem(item) {
    item.name = (item.name + item.name);
    this.crudops.update(item._id, item, () => {
        this.updateInListview(item._id, item);
    }));
}
}

```

Erläuterungen:

- Die `editItem()` Funktion bewirkt lediglich eine Änderung des Namens des Objekts, das dem ausgewählten Listenelement zugrunde liegt.
- Nach Ausführung der `delete()` und `update()` CRUD Operationen wird durch Aufruf von `removeFromListview()` bzw. `updateInListview()` die Listenansicht aktualisiert.

Assoziieren Sie die implementierten Funktionen mit den Optionen des Menüs

[Datei: app.html]

```

<!-- ... -->
<div data-mwf-templatename="mediaItemMenu" class="mwf-listitem-
    menu_mwf-databind_mwf-template_mwf-dialog_mwf-popup">
    <!-- ... -->
    <main>
        <ul>
            <li class="mwf-li-singletitle_mwf-menu-item"
                data-mwf-targetaction="deleteItem">Löschen</li>
            <li class="mwf-li-singletitle_mwf-menu-item"
                data-mwf-targetaction="editItem">Editieren</li>
        </ul>
    </main>
</div>

```

Erläuterungen:

- Ist das Attribut `data-mwf-targetaction` auf einer Menüoption gesetzt, ruft MWF auf dem View Controller, der die Listenansicht verwendet, die angegebene Funktion auf und übergibt ihr das Objekt, das in dem ausgewählten Listenelement dargestellt wird.

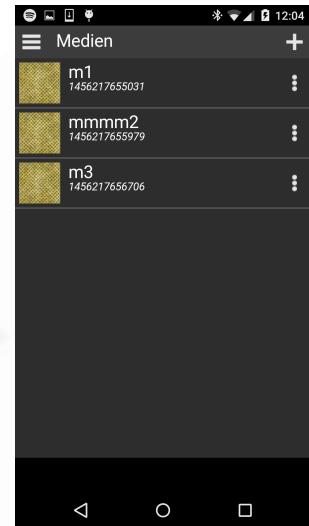
Rufen Sie aus der Funktion `onListMenuItemSelected()` in View Controller die generische Implementierung ihres Ober-Typs auf.

[Datei: ListviewViewController.js]

```
/* */
onListMenuItemSelected(option, listitem, listview) {
    // TODO: implement how selection of option for listitem
    // shall be handled
    super.onListMenuItemSelected(option, listitem,
        listview);
}
```

Erläuterungen:

- Die Funktion ermöglicht die anwendungsspezifische Umsetzung der Auswahl von Optionen des Aktionsmenüs.
- Unsere Implementierung ruft lediglich die im Obertyp des anwendungsspezifischen View Controllers implementierte generische Funktionalität auf, die im vorangegangenen Schritt beschrieben wurde.
- Alternativ zum Aufruf der Funktion des Obertyps könnte die Funktion auch aus dem anwendungsspezifischen View Controller entfernt werden.



4.4.3 Verwendung des MWF Entity Managers

Alternativ zum direkten Aufruf von CRUD Operationen wie bisher bietet MWF auch die Möglichkeit, die Operationen indirekt über einen sogenannten **Entity Manager** zu initiieren. Hierfür ist es erforderlich, die zu verwendenden CRUD Operationen für jeden anwendungsspezifischen Datentyp beim Entity Manager zu registrieren. Der Aufruf der Operationen kann dann für `create()`, `update()` und `delete()` direkt als Funktionsaufruf auf den Entity-Objekten erfolgen, wie nachfolgend gezeigt wird. Die Operationen `read()` und `readAll()` werden auf dem Entity-Datentyp aufgerufen. Intern werden alle diese Funktionsaufrufe durch den Entity Manager unter Verwendung der zum Zeitpunkt des Aufrufs aktivierten Implementierung der CRUD Operationen umgesetzt, was z.B. den einfachen ‘Austausch’ der Operationen erlaubt, z.B. wenn diese wahlweise nur lokal auf einer IndexedDB oder als remote Operationen auf einem server-seitigen Datenspeicher umgesetzt werden sollen.

Der Entity Manager gewährleistet, dass für jede durch ihre `_id` eindeutig identifizierten Entity zur Laufzeit der Anwendung tatsächlich maximal eine Instanz im Speicher existiert. Damit ist sichergestellt, dass alle Komponenten

der Anwendung, die eine Entity nutzen, z.B. verschiedene View Controller, die einander aufrufen, jeweils genau dieselbe Instanz der Entity verwenden. Dies gilt auch dann, wenn auf dieser Instanz Update-Operationen ausgeführt werden. Ein weiteres Funktionsmerkmal des Entity Managers – die Kommunikation des Resultats von CRUD Operationen mittels Ereignissen – werden wir am Ende des Tutorials kennen lernen.

Wir werden zunächst die vorbereitenden Schritte zur Verwendung des Entity Managers durchführen und dann die bestehenden Aufrufe von CRUD Operationen entsprechend den Möglichkeiten des Entity Managers modifizieren.

Registrieren Sie `MediaItem` als Entity und registrieren Sie die auf Instanzen von `MediaItem` anzuwendenden CRUD Operationen

Dafür können Sie in `MyApplication.js` die für `MyEntity` vorgesehenen auskomentierten Codezeilen kopieren und für `MediaItem` anpassen:

[Datei: `MyApplication.js`]

```
// TODO-REPEATED: add new entity types to the array of object
// store names
GenericCRUDImplLocal.initialiseDB("mwftutdb", 1, ["MyEntity", "MediaItem"], () => {
    /* */
    // TODO: do any further application specific initialisations here
    this.registerEntity("MediaItem", entities.MediaItem, true);
    this.registerCRUD("MediaItem", this.CRUDOPS.LOCAL,
        GenericCRUDImplLocal.newInstance("MediaItem"));
    this.registerCRUD("MediaItem", this.CRUDOPS.REMOTE,
        GenericCRUDImplRemote.newInstance("MediaItem"));

    // activate the local crud operations
    this.initialiseCRUD(this.CRUDOPS.LOCAL, EntityManager);

    /* */
});
```

Wissenswertes:

- Wie Sie den Anweisungen entnehmen können, werden hier bereits die Operationen für lokalen *und* remote Datenzugriff registriert. Aktiviert werden dann die lokalen Operationen, die wir auch schon bisher verwendet haben.

Verwenden Sie den EntityManager in der View Controller Implementierung

Hierfür können wir zunächst die Abhängigkeit zu `GenericCRUDImplLocal` und

die Deklaration der Variable `crudops` entfernen, da wir für den Aufruf von CRUD Operationen nur noch die ohnehin bereits importierten Typen des Datamodells benötigen, die im Modul `entities` enthalten sind:

[Datei: `ListviewViewController.js`]

```
define(["mwf", "entities", "GenericCRUDImplLocal"], function(mwf,
    entities, GenericCRUDImplLocal) {
    /*....*/
    class ListviewViewController extends mwf.ViewController {

        constructor() {
            super();
            /*....*/
            this.crudops =
                GenericCRUDImplLocal.newInstance("MediaItem");
        }
    }
}
```

Ersetzen Sie den `readAll()` Funktionsaufruf

Hier und im folgenden sind die zu ersetzenenden Aufrufe auskommentiert dargestellt. Darunter finden Sie jeweils den betreffenden Aufruf unter Verwendung des Entity Managers.

[Datei: `ListviewViewController.js`]

```
oncreate(callback) {
    /*....*/
    //this.crudops.readAll((items) => {
    //    this.initialiseListview(items);
    //});
    entities.MediaItem.readAll((items) => {
        this.initialiseListview(items);
    });
    /*....*/
}
```

Erläuterungen:

- Beachten Sie, dass die ‘Verwendung des Entity Managers’, von der hier die Rede ist, auf Ebene des von Ihnen verfassten Quellcodes nicht sichtbar ist. Sie verwenden für den Aufruf der CRUD Operationen die von Ihnen deklarierten anwendungsspezifischen Datentypen bzw. – wie in den nachfolgenden Fällen – deren Instanzen.
- Tatsächlich ‘verwendet’ wird der Entity Manager intern vom Typ `Entity`, der der Obertyp aller Typen des anwendungsspezifischen Datenmodells ist.

Ersetzen Sie den `create()` Funktionsaufruf

Lagern Sie dafür den Aufruf von `create()` zunächst in eine separate Funktion aus – entsprechend den Funktionen `editItem()` und `deleteItem()`, die wir oben verwendet haben:

[Datei: `ListviewViewController.js`]

```
oncreate(callback) {
    /*...*/
    this.addNewMediaItemElement.onclick = () => {
        //this.crudops.create(new entities.MediaItem("m", "https://
        placeimg.com/100/100/city"), ((created) => {
        //
        this.addToListview(created);
        //}));
        this.createNewItem();
    });
    /*...*/
}

/*...*/
createNewItem() {
    var newItem = new
        entities.MediaItem("m", "https://placeimg.com/100/100/city");
    newItem.create(() => {
        this.addToListview(newItem);
    });
}
/*...*/
```

Erläuterungen:

- Sie sehen hier, wie CRUD Operationen direkt auf einer Instanz eines anwendungsspezifischen Datentyps aufgerufen werden können. Die Funktionsweise des Entity Managers stellt sicher, dass auch das Ergebnis der Operation auf die Instanz des Datentyps übertragen wird, auf dem die Operation aufgerufen wurde. Im Fall von `create()` beinhaltet dies insbesondere die Zuweisung einer `_id` durch die verwendete Datenbank an die neu erstellte Entity. Aus diesem Grund können Sie in der Callback-Funktion den an dieser Stelle aktualisierten Wert von `newItem` an die Listenansicht übergeben.

Ersetzen Sie nun noch die CRUD Operationen `delete()` und `update()`.

```
/*...*/
deleteItem(item) {
```

```

//this.crudops.delete(item._id,(() => {
//    this.removeFromListview(item._id);
//}));

item.delete(() => {
    this.removeFromListview(item._id);
});
}

editItem(item) {
    item.name = (item.name + item.name);

    //this.crudops.update(item._id,item,(() => {
    //    this.updateInListview(item._id,item);
    //}));

    item.update(() => {
        this.updateInListview(item._id,item);
    });
}

```

Passen Sie die Darstellung von `MediaItem` in der Listenansicht an und verwenden Sie hierfür das formatierte Erstellungsdatum der Objekte.

[Datei: app.html]

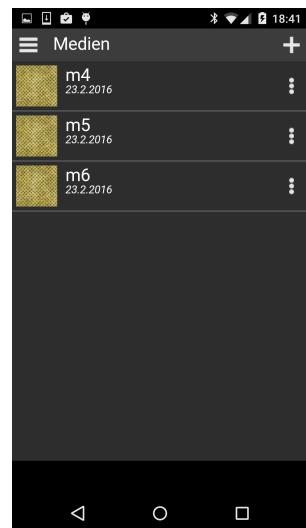
```

<!-- ... -->
<ul class="mwf-listview" data-mwf-listitem-view="myapp-listitem"
    " data-mwf-listitem-menu="mediaItemMenu">
    <li class="mwf-listitem_mwf-li-title-subtitle_mwf-template_
        mwf-databind" data-mwf-templatename="myapp-listitem">
        <!-- ... -->
        <div class="mwf-li-titleblock">
            <h2>{{name}} {{_id}}</h2>
            <h3>{{addedDateString}}</h3>
        </div>
        <!-- ... -->
    </li>
</ul>

```

Erläuterungen:

- Erst jetzt ist es ohne Ausnahmen möglich, auf das durch einen typspezifischen Getter definierte Attribut `addedDateString` erfolgreich zuzugreifen, da der EntityManager sicherstellt, dass alle Entities, die durch CRUD Operationen in IndexedDB oder einer server-seitigen Datenbank persistiert wurden, nach dem Auslesen als Instanzen des betreffenden Entity-Typs verfügbar sind.



- Bei direktem Aufruf der CRUD Operationen wurden Entities hingegen nach dem Auslesen aus der verwendeten Datenquelle als ‘gewöhnliche JavaScript Objekte’ (Pojos) repräsentiert. Die typspezifisch deklarierten Attribute waren auf diesen jedoch nicht vorhanden.
- Um ein aus einer Datenbank ausgelesenes Pojo als Entity zu instantiiieren, fügt der Entity Manager *vor* dem Schreiben in die Datenbank ein zusätzliches Attribut `@typename` dem zu schreibenden Objekt hinzu. Wenn der angegebene Typ – wie `MediaItem` – beim EntityManager registriert wurde, kann das Pojo nach dem Auslesen in eine Entity des betreffenden Typs überführt werden.

4.4.4 Dialoge mit `GenericDialogTemplateViewController`

Die bisher hardcodierten Aktionen für die Erstellung und Aktualisierung von `MediaItem` Objekten werden wir im folgenden durch einen Popup-Dialog partiell durch den Nutzer kontrollierbar machen. Dafür verwenden wir einen generischen View Controller für Dialoge auf Basis von Ractive.js Templates, den das MWF Framework im Modul `GenericDialogTemplateViewController` bereit stellt. Die Erstellung eines Dialogs erfordert dann nur die Deklaration der Dialogansicht in HTML sowie beim Öffnen des Dialogs die Angabe der in den GUI Elementen der Ansicht darzustellenden Daten und die Angabe von Funktionen, die bei Betätigung der Bedienelemente des Dialogs ausgeführt werden sollen.

Bauen Sie zunächst im Styling-Modus die Ansicht des Dialogs auf.

Setzen Sie den Dialog dafür als Schwester-Element von `mediaOverview` um und versehen Sie ihn mit den erforderlichen Attributen.

[Datei: `app.html`]

```
<!-- ... -->
<div class="mwf-view_mwf-view-initial_mwf-styling
    mwf-dialog-shown" data-mwf-viewcontroller="
        ListviewViewController" id="mediaOverview">
    <!-- ... -->
</div>
<!-- ... -->
<div class="mwf-dialog mwf-popup mwf-template mwf-databind
    mwf-styling mwf-shown"
    data-mwf-templatename="mediaItemDialog">

</div>
```

Deklarieren Sie `GenericDialogTemplateViewController` als View Controller für den Dialog.

```
<!-- ... -->
<div class="mwf-dialog_mwf-popup_mwf-template_mwf-databind_
  mwf-view-component mwf-styling mwf-shown"
  data-mwf-viewcontroller="GenericDialogTemplateViewController"
  data-mwf-templatename="mediaItemDialog">

</div>
```

Erläuterungen:

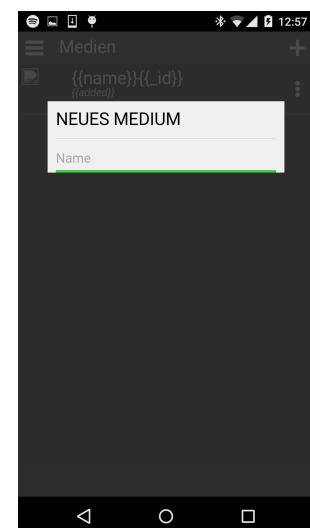
- Mit `mwf-view-component` werden Ansichtselemente ausgezeichnet, die durch einen eigenständigen View Controller gesteuert werden sollen. Bisher wird diese Funktionalität durch MWF nur ansatzweise unterstützt, beispielsweise können Ansichten mit eigenen View Controllern noch nicht ineinander eingebettet werden.
- Der View Controller wird dann durch das Attribut `data-mwf-viewcontroller` identifiziert.

Fügen Sie eine Überschrift und ein Formular mit Texteingabe-Element hinzu

```
<!-- ... -->
<div class="mwf-dialog_mwf-popup_mwf-template_mwf-databind_mwf-
  view-component_mwf-styling_mwf-shown" data-mwf-
  viewcontroller="GenericDialogTemplateViewController" data-
  mwf-templatename="mediaItemDialog">
  <main>
    <h2>Neues Medium</h2>
    <form id="itemEditForm">
      <input name="name" autocomplete="off" class=""
        mwf-autofocus" type="text" placeholder="Name"
        required="required"/>
    </form>
  </main>
</div>
```

Erläuterungen:

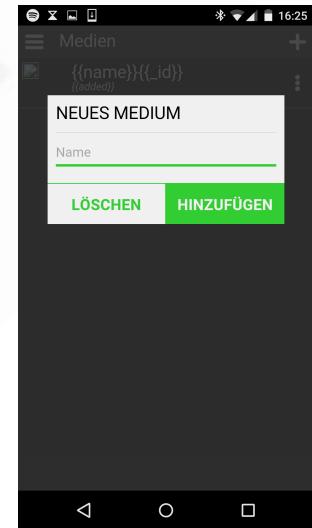
- Im Ggs. zum Aktionsmenü für Listenelemente wird dieser Dialog nicht mit einer separaten Kopfleiste, sondern mit einer Überschrift innerhalb des `<main>` Elements realisiert.



- Um dem Nutzer die Kontrolle über die auszuführenden CRUD Operationen zu ermöglichen, verwendet der Dialog ein Formular mit einem Texteingabefeld, in dem der Name von `MediaItem` Objekten angegeben und modifiziert werden kann.
- Die Klasse `mwf-autofocus` bewirkt, dass bei Öffnen des Dialogs der Fokus auf dem Texteingabefeld gesetzt wird.

Fügen Sie zwei Bedienelemente für das Löschen bzw. Erstellen/Aktualisieren eines `MediaItem` hinzu.

```
<main>
  <h2>Neues Medium</h2>
  <form id="itemEditForm">
    <input name="name" autocomplete="off" class="mwf-autofocus" type="text" placeholder="Name" required="required" value="" />
    <div class="mwf-buttonbar">
      <button class="mwf-left-align mwf-button" type="button">Löschen</button>
      <input class="mwf-button-prio mwf-right-align mwf-button" type="submit" value="Hinzufügen"/>
    </div>
  </form>
</main>
```



Erläuterungen:

- Das `<div>`-Element mit Klasse `mwf-buttonbar` in Verbindung mit den beiden als `mwf-button` markierten Kindelementen bewirkt die gewünschte Formatierung der Bedienelemente, die die gesamte Breite des Dialogs einnehmen.
- Anhand der Klasse `mwf-button-prio` wird das damit gekennzeichnete Bedienelement visuell stärker gegenüber dem anderen Element hervorgehoben.
- Die Zuweisung von `mwf-left-align` und `mwf-right-align` bewirkt, dass den Bedienelementen jeweils 50% der verfügbaren Breite zugewiesen werden, d.h. das Layout ist derzeit nur für Buttonleisten mit einem oder zwei Kind-Elementen realisierbar.
- Die Unterscheidung der Bedienelemente hinsichtlich ihres Typs `submit` vs. `button` entspricht der üblichen Unterscheidung für HTML Formulare

zwischen ‘formular-absendenden’ Aktionen und ‘sonstigen’ Aktionen, deren Verarbeitung die Felder des Formulars und deren Werte nicht notwendigerweise berücksichtigt.

Ergänzen Sie den Dialog nun um Angaben zum Data Binding zur Behandlung von Eingabeereignissen.

Zur Ereignisbehandlung können wir innerhalb eines Ractive.js Templates die Namen ereignisbehandelnder Funktionen angeben, die bei Anwendung des Templates bereit gestellt werden müssen.

```
<form id="itemEditForm" on-submit="submitForm">
  <input name="name" autocomplete="off" class="mwf-autofocus"
    type="text" placeholder="Name" required="required" value
    ="{{item.name}}"/>
  <div class="mwf-buttonbar">
    <button class="mwf-left-align_mwf-button" type="button"
      on-click="deleteItem">Löschen
    </button>
    <input class="mwf-button-prio_mwf-right-align_mwf-button"
      type="submit" value="Hinzufügen"/>
  </div>
</form>
```

Erläuterungen:

- Der Wert des `<input>` Texteingabefelds wird an den Wert des Attributs `name` eines dem Template unter dem Namen `item` übergebenen Objekts gebunden.
- Ereignisbehandelnde Funktionen werden in Ractive.js mittels Attributen der Form `on-<ereignisname>` notiert.
- Der vorliegende Dialog deklariert zwei solcher Funktionen, die auf das Absenden des Formulars und auf die Betätigung des ‘Löschen’-Buttons reagieren, mittels der beiden Zuweisungen für `on-submit` auf `<form>` bzw. `on-click` auf `<button>`.
- Siehe für weitere Hinweise zur Ereignisbehandlung in Ractive.js:
<https://ractive.js.org/concepts/#event-management>

Entfernen Sie jetzt wieder die Styling-Klassen `mwf-styling` sowie `mwf-dialog-shown` und `mwf-shown` von den `<div>` Elementen für `mediaOverview` und `mediaItemDialog`.

Öffnen Sie den Dialog aus dem View Controller der Listenansicht

Der Dialog soll sowohl für die Erstellung eines neuen `MediaItem` Elements, als auch für die Änderung eines bestehenden Elements verwendet werden. Wir ersetzen daher die bisherige Implementierung der beiden Funktionen `editItem` und `createNewItem`.

Verwenden Sie den Dialog in `createNewItem()`

Setzen Sie `createNewItem()` wie folgt um:

[Datei: `ListviewViewController.js`]

```
createNewItem() {
    var newItem = new entities.MediaItem("", "https://placeimg.com/100/100/city");
    this.showDialog("mediaItemDialog", {
        item: newItem,
        actionBindings: {
            submitForm: ((event) => {
                newItem.create(() => {
                    this.addToListview(newItem);
                });
            })
        }
    });
}
```

Erläuterungen:

- Identifiziert wird ein Dialog entweder durch seine `id` oder – wenn er als Template umgesetzt ist – durch den Wert seines `data-mwf-templateName` Attributs.
- Mittels Aufruf von `showDialog()` im View Controller und Angabe des Identifikators des Dialogs wird der Dialog geöffnet. Diese Funktion wird durch die Oberklasse des View Controllers bereit gestellt.
- Als zweites Argument kann an `showDialog()` ein Argument-Objekt übergeben werden. Im Falle des Aufrufs von Dialogen, die mit `GenericDialogTemplateViewController` realisiert werden, enthält dieses Argument-Objekt zum einen die Daten, die in Ractive.js Data Binding Expressions verwendet werden – hier das `item` Attribut. In einem weiteren Attribut `actionBindings` wird für jedes der im Template verwendeten Attribute zur Ereignisbehandlung – z.B. `on-submit`, `on-click` – eine Funktion übergeben, die durch Ractive.js aufgerufen wird, wenn das betreffende Ereignis auftritt.
- Die Angabe einer `deleteItem` Funktion ist für den vorliegenden Fall, in dem wir ein neues `MediaItem` erstellen, nicht erforderlich. Siehe am En-

de dieses Abschnitts weitere Hinweise, wie die Aktion für diesen Fall deaktiviert werden kann.

Wenn Sie die Anwendung im nun erreichten Zustand neu laden, sollte bei Betätigung der ‘+’-Aktion der Dialog geöffnet werden. Nach Angabe eines Namens und Betätigung von ‘Hinzufügen’ sollte das neue Element in der Liste angezeigt werden. Beachten Sie aber, dass nun bei jeder Ausführung von ‘Hinzufügen’ die Anwendung neu geladen wird.

Unterbinden Sie das Neuladen der Anwendung bei Ausführung der `on-submit` Funktion

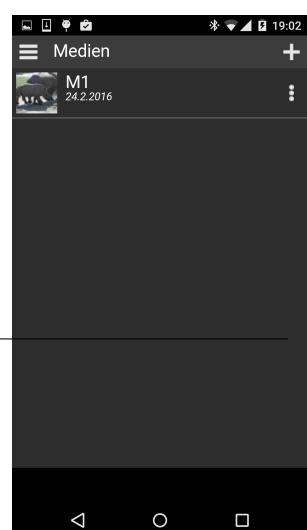
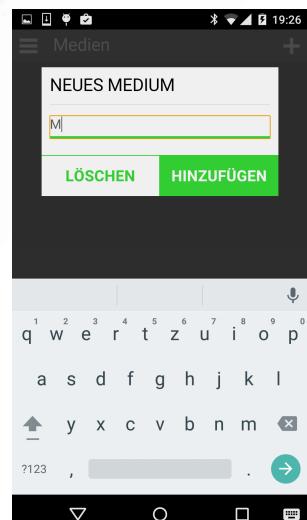
Wie für gewöhnliche HTML Formulare müssen Sie auch für Formulare in Reactive.js Templates die durch den Browser unterstützte Übermittlung der Formulardaten an den Server beim Absenden des Formulars unterbinden.

```
createNewItem() {
    var newItem = new entities.MediaItem("", "https://placeimg.com/100/100/city");
    this.showDialog("mediaItemDialog", {
        item: newItem,
        actionBindings: {
            submitForm: ((event) => {
                event.original.preventDefault();
                newItem.create(() => {
                    this.addToListview(newItem);
                });
                this.hideDialog();
            })
        }
    });
}
```

Erläuterungen:

- An ereignisbehandelnde Funktionen wird ein Reactive-spezifisches Event-Objekt übergeben, wobei das originale DOM Event als Wert von dessen `original` Attribut zugreifbar ist.
- Das Unterbinden der Formulardatenübermittlung erfolgt durch Aufruf von `preventDefault()` auf dem DOM Event.
- Mittels Aufruf von `hideDialog()` auf dem View Controller wird der Dialog wieder geschlossen.

Verwenden Sie den Dialog nun auch in der Umsetzung der `editItem` Funktion



```

editItem(item) {
    this.showDialog("mediaItemDialog", {
        item: item,
        actionBindings: {
            submitForm: ((event) => {
                event.original.preventDefault();
                item.update(() => {
                    this.updateInListview(item._id, item);
                });
                this.hideDialog();
            })
        }
    });
}

```

Erläuterungen:

- Vom Aufruf des Dialogs in `createNewItem()` unterscheidet sich dieser Aufruf zum einen dadurch, dass als Wert des `item` Attributs beim Aufruf des Dialogs das an die `updateItem()` Funktion übergebene Objekt weitergegeben wird. Nach Öffnen des Dialogs wird dessen Name als Wert des Texteingabefelds dargestellt.
- Wird der Wert des Eingabefelds durch den Nutzer geändert, dann wird der neue Wert durch das in Ractive.js vorgenommene ***bidirektionale Daten Binding*** als Wert des `name` Attributs von `item` gesetzt.
- Der zweite Unterschied zu `createNewItem()` besteht darin, dass als Wert der im Template mit `submitForm` bezeichneten Funktion, die im Falle eines `submit` Ereignisses auf dem Formular aufgerufen wird, eine Funktion übergeben wird, die die `update` Funktion auf dem `item` Objekt aufruft.

Führen Sie die Anwendung aus, rufen Sie den Dialog aus dem Aktionsmenü eines Listenelements auf und ändern Sie den Namen des Elements.

Fügen Sie in `editItem()` nun noch ein `actionBinding` für die ‘Löschen’ Aktion hinzu.

In der ereignisbehandelnden Funktion, die wir hierfür deklarieren, können wir die bereits existierende `deleteItem()` Funktion auf View Controller aufrufen.

```

this.showDialog("mediaItemDialog", {
    item: item,
    actionBindings: {
        submitForm: ((event) => {
            /*...*/
        }), /*!!!!*/
        deleteItem: ((event) => {

```

```

        this.deleteItem(item);
        this.hideDialog();
    })
}
);

```

Erläuterungen:

- Vergessen Sie nicht das Komma am Ende der Deklaration der unter `submitForm` angegebenen Funktion, um die Attribute von `actionBindings` voneinander zu separieren.
- Auch hier wird nach Aufruf von `deleteItem()` mittels `hideDialog()` der Dialog geschlossen.

Führen Sie die Anwendung aus, rufen Sie den Dialog aus dem Aktionsmenü eines Listenelements auf und löschen Sie das Element.

Abschließend werden wir die Realisierung des Dialogs nun noch an die beiden Anwendungsfälle Neuerstellung eines `MediaItem` vs. Editieren eines existierenden `MediaItem` anpassen.

Deaktivieren Sie die ‘Löschen’-Aktion bei Neuerstellung und passen Sie die Beschriftung des `submit`-Elements an den betreffenden Anwendungsfall an.

[Datei: app.html]

```

<main>
    <!-- ... -->
    <form id="itemEditForm" on-submit="submitForm">
        <!-- ... -->
        <div class="mwf-buttonbar">
            <button class="mwf-left-align_mwf-button"
                disabled="#unless
                    item.created}disabled{/unless}" type="button" on-click="deleteItem">Löschen
                </button>
            <input class="mwf-button-prio_mwf-right-align_mwf-
                button" type="submit" value="#if
                    item.created}Ändern{/else}Hinzufügen{/if}"/>
        </div>
    </form>
</main>

```

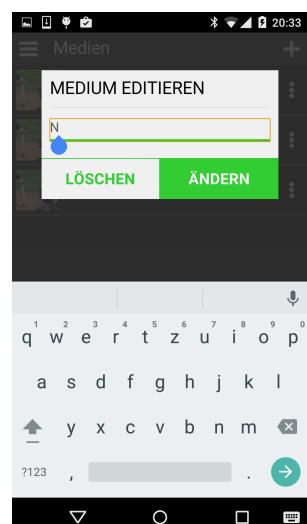
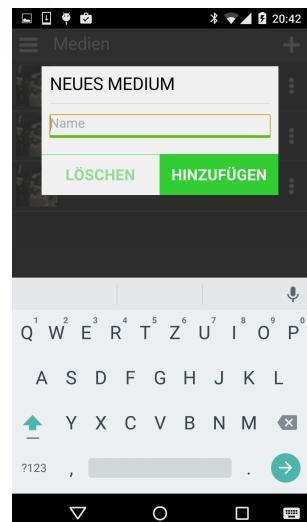
Erläuterungen:

- Mittels der Anweisungen `{#unless}` und `{#if}` können in Ractive.js Templates Bestandteile des darzustellenden Dokuments in Abhängigkeit von den dem Template übergebenen Daten eingefügt, entfernt bzw. in ihrer Realisierung angepasst werden.
- Im vorliegenden Fall sind die Wertzuweisung des `disabled` Attributs und die Beschriftung des `submit` Elements abhängig vom Wert des `created` Attributs des dem Template übergebenen `MediaItem` Objekts.
- Das `created` Attribut wird im `Entity`-Typ, d.h. im Obertyp von `MediaItem` definiert und liefert in Abhängigkeit vom Vorhandensein bzw. vom Wert des `_id` Attributs auf `MediaItem` entweder `true` oder `false`.
- Siehe zur Verwendung von ‘Conditional Sections’ in Ractive.js auch <https://ractive.js.org/tutorials/conditional-sections/>

Modifizieren Sie nun noch die Titelzeile des Dialogs in Abhängigkeit vom Anwendungsfall

```
<main>
  <h2>{{#if item.created}}Medium editieren{{else}}Neues
    Medium{{/if}}</h2>
  <!-- ... -->
</main>
```

Abschließend werden wir im folgenden Abschnitt des Tutorials noch eine Leseansicht für `MediaItem` Elemente umsetzen und eine alternative Lösung zur Aktualisierung von Ansichten nach Ausführung von CRUD Operationen implementieren.



5 Umsetzung der Leseansicht

Entsprechend den Anforderungen soll die Leseansicht für ein ausgewähltes `MediaItem` das mit diesem assoziierte Bild in Großansicht über die gesamte Bildschirmbreite darstellen. Außerdem soll es dem Nutzer möglich sein, das `MediaItem` zu löschen und danach zur Listenansicht zurückzukehren. Im folgenden wird die Leseansicht zunächst statisch aufgebaut und dann als `Active.js` Template für beliebige `MediaItem` Instanzen umgesetzt.

5.1 Gestaltung

Erstellen Sie ein neues `<div>` Element mit `mwf-view` Klasse und aktivieren Sie den Styling-Modus

[Datei: `app.html`]

```
<body data-mwf-application="MyApplication">
  <!-- TODO-REPEATED: add new views here -->

  <!-- ... -->
  <div class="mwf-view mwf-styling" id="mediaReadview">

    </div>
    <!-- ... -->
</div>
```

Übernehmen Sie den Aufbau aus Kopfzeile, scrollbarem Hauptbereich und Fußzeile aus der Listenansicht

```
<div class="mwf-view_mwf-styling">
  <header>

    </header>
    <main class="mwf-scrollbar">

      </main>
      <footer>

        </footer>
    </div>
```

Stellen Sie in Kopfzeile neben der ‘Hauptmenü’ Aktion den Titel des `MediaItem` dar und bieten Sie am rechten Rand die Aktion zum Löschen des Elements an

```

<header>
  <button class="mwf-imgbutton mwf-img-sandwich
    mwf-left-align"></button>
  <h1 class="mwf-left-align mwf-right-fill">Name</h1>
  <button class="mwf-imgbutton mwf-img-delete
    mwf-right-align"></button>
</header>

```

Fügen Sie in die Fußzeile links ein Element ein, das die Rückkehr zur vorangegangenen Ansicht ermöglicht.

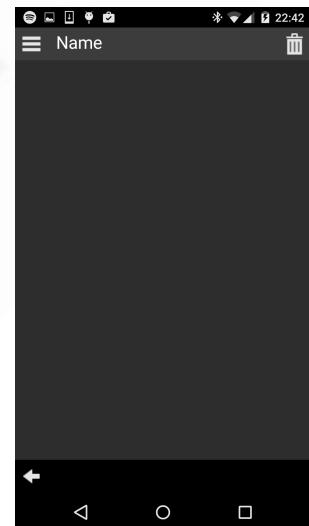
```

<footer>
  <button class="mwf-back mwf-imgbutton
    mwf-img-backward"></button>
</footer>

```

Erläuterungen:

- Die Markierung des Elements als `mwf-back` wird bei Instantierung des View Controllers der Ansicht von MWF erkannt und mit einer Funktion verknüpft, die die Rückkehr zur vorangegangenen Ansicht veranlasst.



Fügen Sie in den Hauptbereich jetzt ein `` Element ein und weisen Sie diesem eine beliebige Bildquelle zu.

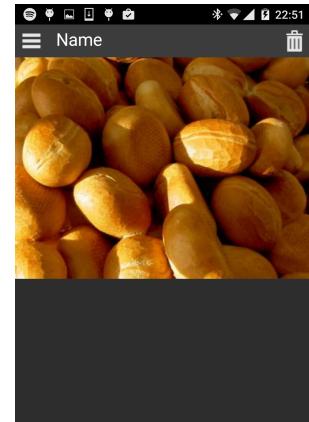
```

<main class="mwf-scrollview">
  
</main>

```

Weisen Sie dem `` Element in einer anwendungsspezifischen Styling-Regel 100% Breite zu.

[Datei: mystyle.css]



```
#mediaReadview img {
    width: 100%;
    max-width: 100%
}
```

Deaktivieren Sie den Styling-Modus und deklarieren Sie die Ansicht als Reactive.js Template.

[Datei: app.html]

```
<div class="mwf-view_mwf-styling" id="mediaReadview">
    <div class="mwf-template mwf-databind"
        data-mwf-templatename="mediaReadviewTemplate">
        <header>
            <!-- ... -->
            <h1 class="mwf-left-align_mwf-right-fill">
                {{item.name}}</h1>
            <button class="mwf-imgbutton_mwf-img-delete_mwf-right-
                align" on-click="deleteItem"></button>
        </header>
        <main class="mwf-scrollbar">
            
        </main>
        <!-- ... -->
    </div>
</div>
```

Erläuterungen:

- Das zusätzliche `<div>` Element, das die Ansicht als Template deklariert, ist für Fälle wie den vorliegenden erforderlich, bei dem tatsächlich die gesamte Ansicht als Template umgesetzt werden soll. Bei Anwendung des Templates zur Laufzeit wird dieses Container-Element entfernt, sodass die Struktur der Ansicht dann wieder der vorstehend statisch entwickelten Struktur entspricht.
- Innerhalb des Templates werden dann die bereits oben verwendeten Data Binding Expressions für die Attribute `name` und `src` eines `MediaItem` sowie die Assoziation eines `click` Ereignisses auf dem ‘Löschen’-Bedienlement mit einer dem entsprechend benannten Funktion vorgesehen, deren Implementierung durch den View Controller mit dem Template assoziiert werden muss.

5.2 Steuerung

Erstellen Sie einen neuen View Controller für die Steuerung der Leseansicht.

Kopieren Sie `ViewControllerTemplate`, benennen Sie die Datei `ReadviewViewController` und ersetzen Sie intern alle Vorkommen von `ViewControllerTemplate` durch `ReadviewViewController`.

[Datei: `ReadviewViewController.js`]

```
define(["mwf", "entities"], function(mwf, entities) {

    class ReadviewViewController extends mwf.ViewController {
        /*...*/
    }

    // and return the view controller function
    return ReadviewViewController;
});
```

Deklarieren Sie den neuen View Controller als Modul und laden Sie es beim Start der Anwendung.

[Datei: `Main.js`]

```
/*...*/
requirejs.config({
    /*...*/

    // TODO-REPEATED: add all new modules here, e.g.
    // Viewcontroller implementations
    paths: {
        /*...*/
        /* application libraries: view controllers */
        MyInitialViewController: 'js/controller/
            MyInitialViewController',
        ListviewViewController: 'js/controller/
            ListviewViewController',
        ReadviewViewController:
            'js/controller/ReadviewViewController'
    }
});

// TODO-REPEATED: add new ViewControllers to the dependency
// array
requirejs(["mwf", "GenericDialogTemplateViewController", "
    MyApplication", "MyInitialViewController", "
    ListviewViewController", "ReadviewViewController"],
```

```

        function(mwf) {
            mwf.onloadApplication();
        });
    }

```

Weisen Sie den neuen View Controller der zuvor erstellten Ansicht zu, markieren Sie diese als `mwf-view-initial` und entfernen Sie letztere Markierung von der Listenansicht.

[Datei: app.html]

```

<!-- ... -->
<div class="mwf-view mwf-view-initial" data-mwf-viewcontroller=
    "ListviewViewController" id="mediaOverview">
    <!-- ... -->
</div>
<div class="mwf-view mwf-view-initial" id="mediaReadview"
    data-mwf-viewcontroller="ReadviewViewController">
    <!-- ... -->
</div>
<!-- ... -->

```

Erläuterungen:

- Die Markierung `mwf-view-initial` bewirkt, dass die damit ausgezeichnete Ansicht einer Anwendung unmittelbar nach Starten der Anwendung dargestellt wird.
- Vorausgesetzt, die Daten, die eine Ansicht von außen übergeben bekommt, lassen sich in vergleichbarer Weise als Testdaten innerhalb der Ansicht aufbauen, kann damit das Laufzeitverhalten einer Ansicht auf sinnvolle und effiziente Weise getestet werden.

Befüllen Sie das Template im View Controller mit einem hardcodierten `MediaItem`.

Falls Sie ein Template in der hier gezeigten Form als Beschreibung einer kompletten Ansicht verwenden, müssen Sie dieses ‘von Hand’ in der `oncreate` Funktion des für die Ansicht zuständigen View Controllers befüllen.

[Datei: ReadviewViewController.js]

```

/*...*/
class ReadviewViewController extends mwf.ViewController {
    /*...*/
    constructor() {
        super();

        this.viewProxy = null;
    }
}

```



```

oncreate(callback) {
    // TODO: do databinding, set listeners, initialise the
    // view
    var mediaItem = new
        entities.MediaItem("m", "https://placeimg.com/300/400/music");
    this.viewProxy =
        this.bindElement("mediaReadviewTemplate", {item:
            mediaItem}, this.root).viewProxy;

    // call the superclass once creation is done
    super.oncreate(callback);
}
}

```

Erläuterungen:

- Die Befüllung eines Templates kann durch Aufruf von `bindElement()` auf View Controller veranlasst werden.
- Übergeben werden der Funktion der Name des Templates, die Daten, die für die Befüllung verwendet werden sollen, sowie ein Wurzelement, als dessen Kind-Element das gefüllte Template realisiert werden soll.
- Der Rückgabewert von `bindElement()` enthält ein `viewProxy` Objekt, das für darauf folgende Zugriffe auf das Template sowie, u.a., auch für die Aktualisierung der dargestellten Daten, verwendet werden kann.
- Wichtig ist, dass nach Befüllung des Templates die `oncreate` Funktion des durch MWF bereit gestellten Supertyps des View Controllers aufgerufen wird. Dort werden u.a. generische Event Handler für die Bedienelemente des gefüllten Templates gesetzt.

Binden Sie die im Template vorgesehene Funktion `deleteItem` an eine konkrete Implementierung.

Hierfür können Sie das `viewProxy` Objekt verwenden.

[Datei: `ReadviewViewController.js`]

```

/*...*/
oncreate(callback) {
    // TODO: do databinding, set listeners, initialise the view
    var mediaItem = new entities.MediaItem("m", "https://placeimg
        .com/300/400/music");
    this.viewProxy = this.bindElement("mediaReadviewTemplate", {
        item: mediaItem}, this.root).viewProxy;
    this.viewProxy.bindAction("deleteItem", (() => {
        mediaItem.delete() => {

```

```

        this.previousView();
    })
});
/* ... */
}
/* ... */

```

Erläuterungen:

- Die Funktion `bindAction()` auf dem `viewProxy` Objekt assoziiert eine Funktion mit einem Funktionsbezeichner, der innerhalb des Templates verwendet wird, in unserem Fall mit einem `on-click` Handler.
- Der Aufruf von `previousView()` bewirkt den Übergang in die Vorgängeransicht einer Ansicht, falls verfügbar.

Die Entwicklung der Leseansicht ist nun fast abgeschlossen. Wir werden jetzt die Leseansicht aus der Listenansicht heraus aufrufen.

Rufen Sie die Leseansicht aus der Listenansicht bei Auswahl eines Listenelements auf.

Hierfür modifizieren wir die Implementierung der bereits zu Beginn umgesetzten Funktion `onListItemSelected`.

[Datei: `ListviewViewController.js`]

```

/* ... */
class ListviewViewController extends mwf.ViewController {
/* ... */
onListItemSelected(listitem,listview) {
// TODO: implement how selection of listitem shall be
// handled
alert("Element " + listitem.name + " wurde ausgewählt!");
this.nextView("mediaReadview", {item: listitem});
}
}

```

Erläuterungen:

- Der Aufruf von `nextView()` auf einer View Controller Implementierung bewirkt einen Übergang in die im ersten Argument bezeichnete Folgeansicht, wobei diese das im zweiten Argument angegebene Objekt als Argument übergeben bekommt.
- Zur Identifikation einer Folgeansicht wird die in `app.html` vergebene `id` der Ansicht verwendet.
- Vor Übergang in die Folgeansicht wird auf der aktuellen Ansicht die `onpause()` Lebenszyklusmethode aufgerufen.

Greifen Sie in der Leseansicht auf die übergebenen Argumente zu.

Der Zugriff erfolgt üblicherweise in der `oncreate()` Lebenszyklusmethode.

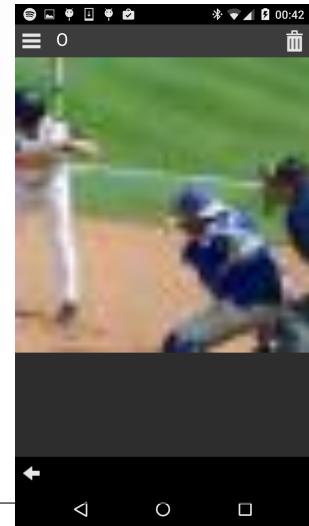
[Datei: ReadviewViewController.js]

```
/*...*/  
class ReadviewViewController extends mwf.ViewController {  
    oncreate(callback) {  
        // TODO: do databinding, set listeners, initialise the  
        // view  
        var mediaItem = this.args.item; //new entities.MediaItem("m","https://placeimg.com/300/400/music");  
        /*...*/  
    }  
}
```

Weisen Sie die Eigenschaft `mwf-view-initial` nun wieder der Listenansicht zu.

[Datei: app.html]

```
<!-- ... -->  
<div class="mwf-view_mwf-view-initial" data-mwf-viewcontroller=  
    "ListviewViewController" id="mediaOverview">  
    <!-- ... -->  
</div>  
<div class="mwf-view_mwf-view-initial" id="mediaReadview" data-  
    mwf viewcontroller="ReadviewViewController">  
    <!-- ... -->  
</div>  
<!-- ... -->
```



Sie sollten jetzt bei Auswahl eines Elements der Listenansicht dessen Leseansicht angezeigt bekommen. Löschen des Elements sollte zurück zur Listenansicht führen. Wie wir dort das Entfernen des Elements aus der Liste veranlassen können, wird weiter unten gezeigt.

Vereinfachen Sie den Übergang zur Leseansicht aus der Listenansicht.

Deklarieren Sie dafür die Leseansicht als ‘Zielansicht’ auf dem Template für die Umsetzung der einzelnen Listenansichten. [Datei: app.html]

```
<!-- ... -->  
<div class="mwf-view_mwf-view-initial" data-mwf-viewcontroller=  
    "ListviewViewController" id="mediaOverview">  
    <!-- ... -->  
    <main class="mwf-scrollview">  
        <ul class="mwf-listview" data-mwf-listitem-view="myapp-  
            listitem" data-mwf-listitem-menu="mediaItemMenu">
```

```

<li class="mwf-listitem_mwf-li-title-subtitle_mwf-
template_mwf-databind" data-mwf-templateName="

myapp-listitem"
data-mwf-targetview="mediaReadview"

```

Erläuterungen:

- Wenn das Attribut `data-mwf-targetview` auf einem Listenelement gesetzt ist, führt MWF bei Auswahl des Elements die oben von uns selbst verwendete Funktion `nextView()` aus und übergibt den Wert des Attributs, d.h. den Identifikator der Zielansicht, sowie ein Argument-Objekt mit Struktur `{item: <selectedItem>}`, wobei `<selectedItem>` das Objekt ist, das dem ausgewählten Listenelement zugrunde liegt.

Löschen Sie nun die Funktion `onListItemSelected` im View Controller für `mediaOverview` und führen Sie den Übergang zwischen Listenansicht und Leseansicht erneut aus. Es sollten keine Änderungen gegenüber dem vorangegangenen Zustand feststellbar sein. Nutzen Sie dann die ‘zurück’-Aktion, um in die Listenansicht zurückzukehren. Öffnen Sie nun die Leseansicht erneut und führen Sie die ‘Löschen’-Aktion aus. Entspricht die Listenansicht bei Rückkehr Ihren Erwartungen?

Die Ausführung von CRUD Operationen auf den von einer Anwendung verwendeten Daten erfordert üblicherweise eine Anpassung der Ansichten der Anwendung, anhand derer der Nutzer der erfolgreichen Abschluss der betreffenden Operation signalisiert wird. Wird beispielsweise in unserer Anwendung ein neues `MediaItem` erstellt oder ein bestehendes modifiziert oder gelöscht, dann sollte die jeweilige Änderung anhand der Darstellung der `MediaItem` Objekte in der Listenansicht nachvollzogen werden können. Bisher wurde dies dadurch erreicht, dass jedem Aufruf einer CRUD Operation eine Callback-Funktion übergeben wurde, die nach erfolgreicher Ausführung der CRUD Operation eine entsprechende Ansichtsänderung veranlasste, z.B. in der nachfolgenden Umsetzung von `deleteItem()` in `ListviewViewController`:

[Datei: `ListviewViewController.js`]

```

deleteItem(item) {
    item.delete(( ) => {
        this.removeFromListview(item._id);
    });
}

```

```
}
```

Die `delete()` Operation hatten wir auch in der vorangegangenen Umsetzung der Leseansicht für `MediaItem` verwendet. Dort jedoch sollte nach erfolgreicher Ausführung lediglich die Rückkehr aus der Leseansicht in die Listenansicht veranlasst werden. Bisher wird in letzterer jedoch noch das eigentlich gelöschte `MediaElement` angezeigt. Da MWF keine direkte Interaktion zwischen View Controllern erlaubt, ist es nicht möglich, die Entfernung des gelöschten Elements aus der Listenansicht z.B. mittels eines Aufrufs der `deleteItem()` Funktion auf `ListviewViewController` durch `ReadviewViewController` zu veranlassen. MWF sieht hingegen die Möglichkeit vor, dass aus einer Folgeansicht ‘Rückgabewerte’ an die vorangegangene Ansicht übergeben werden können – so kann die Tatsache, dass im Zuge der Bedienung der Leseansicht das dargestellte `MediaItem` Objekt gelöscht wurde, durch Übergabe eines Rückgabewerts an die `previousView()` Funktion wie folgt signalisiert werden:

[Datei: `ReadviewViewController.js`]

```
/*...*/
this.viewProxy.bindAction("deleteItem", () => {
    mediaItem.delete(() => {
        this.previousView({deletedItem:mediaItem});
    })
});
```

Auf die Übergabe des Rückgabewerts kann `ListviewViewController` seinerseits durch Implementierung einer Funktion `onReturnFromSubview()` reagieren, der die ID der gerade verlassenen Folgeansicht, deren Rückgabewert sowie ein ggf. ebenfalls von dieser gesetzter ‘Rückgabestatus’ übergeben wird – dieser Mechanismus ist wie einige andere Details von MWF durch eine vergleichbare Lösung in Android inspiriert:

[Datei: `ListviewViewController.js`]

```
onReturnFromSubview(subviewid, returnValue, returnStatus, callback
    ) {
    if (subviewid == "mediaReadview" && returnValue &&
        returnValue.deletedItem) {
        this.removeFromListview(returnValue.deletedItem._id);
    }
    callback();
}
```

Durch Abgleich der `subviewid` der Folgeansicht und Überprüfen des Rückgabewerts kann, wie hier gezeigt, auf eine in der Folgeansicht veranlasste Zustandsänderung bezüglich der von der Anwendung verwendeten Daten reagiert werden. Auf diese Weise können Sie die bisher noch ausstehende Funktionalität der Entfernung eines `MediaItem` aus der Listenansicht nach Ausführung der

‘Löschen’-Aktion in der Leseansicht umsetzen. Beachten Sie, dass der Aufruf der `callback`-Funktion erforderlich ist, um die Rückkehr in die Listenansicht abzuschließen. Soll hingegen die Ansicht, in die eine Rückkehr erfolgt, ‘übersprungen’ werden und unmittelbar deren Vorgängeransicht aufgerufen werden, kann dies seinerseits durch Aufruf von `previousView()` in `onReturnFromSubview()` erfolgen. Für diese Fälle muss dann jedoch der Aufruf von `callback` unterbunden werden.

Nach dem vorstehend beschriebenen Beispiel der Kommunikation zwischen Leseansicht und Listenansicht mittels `previousView()` und `onReturnFromSubview()` können Sie alle vergleichbaren Anforderungen des gesamten Übungsprogramms, auch über die Anforderungen von MWF1-4 hinaus, implementieren. Eine fortgeschrittene Alternative zur Implementierung ansichtsübergreifender Datenübermittlung wird im nachfolgenden Kapitel beschrieben. Sollten Sie diese umsetzen, ist es empfehlenswert, sie durchgängig zu verwenden. **Beachten Sie diesbezüglich, dass Konflikte aus dem Nebeneinander beider Varianten entstehen können und dass die Live-Demos zu MWF u.U. nur die einfachere Variante zeigen werden.**

6 Verwendung von Event-Notifikationen zur Reaktion auf CRUD Operationen

Vorstehend wurde beschrieben, wie ansichtsübergreifende Kommunikation – z.B. das Feedback, das die Leseansicht an die Listenansicht im Fall des Löschens von `MediaItem` Elementen übermitteln muss – unter Verwendung von `previousView()` und `onReturnFromSubview()` und der Übergabe geeigneter Daten umgesetzt werden kann. Bedenken Sie jedoch, dass bei einem weiteren Ausbau der Anwendung evtl. noch an anderen Stellen und in anderen Ansichten der Anwendung CRUD Operationen ausgeführt werden können, auf die die zu diesem Zeitpunkt gerade nicht im Vordergrund gezeigten Ansichten dann ebenfalls reagieren müssten. In diesem Fall würde auch die oben gezeigte Verwendung und Verarbeitung von Rückgabewerten die Gefahr potentiell unübersichtlicher Codeerweiterungen zumindest in sich bergen. Die Anforderung, dass Ansichten auf das Ergebnis bestimmter CRUD Operationen *reagieren* müssen – unabhängig von der Stelle, an der die CRUD Operation veranlasst wurde, sollte daher bei Einsatz eines Frameworks wie MWF nicht nur als reaktives Verhalten *benennbar*, sondern auch als solches *implementierbar* sein. Zu diesem Zweck stellt MWF einen **Benachrichtigungsmechanismus** zur Verfügung, der es den View Controllern erlaubt, für die für sie relevanten **Ereignisse** Listener-Funktionen zu registrieren, die durch den Benachrichtigungsmechanismus aufgerufen werden, sobald ein Ereignis auftritt. Vorbild hierfür ist die Verwendung von Event Listener Funktionen für DOM Ereignisse, die es ei-

ner Anwendung erlauben, auf die Bedienung einer Ansicht durch den Nutzer zu reagieren. Die Tatsache, dass die Listenansicht sich dafür interessiert, dass `MediaItem` Objekte gelöscht werden, könnte z.B. wie folgt durch Deklaration eines `Listeners` in `oncreate()` zum Ausdruck gebracht werden:

[Datei: `ListviewViewController.js`]

```
oncreate(callback) {
    /*...*/
    this.addListener(new
        mwf.EventMatcher("crud", "deleted", "MediaItem"), ((event)
    => {
        this.removeFromListview(event.data);
    })
);
/*...*/
}
```

Durch die Angabe eines `EventMatcher` Objekts können mittels dreier Attribute – `group`, `type` und `target` – Ereignisse beschrieben und durch Übergabe an `addListener()` mit einer ereignisbehandelnden Funktion assoziiert werden. Dieser Funktion wird bei Auftreten eines Ereignisses, auf das der `EventMatcher` zutrifft, ein `Event`-Objekt übergeben, das die Attribute des Ereignisses enthält und das zusätzlich über ein Attribut `data` verfügt, als dessen Wert weitere für das betreffende Ereignis relevante Daten gesetzt werden können. Im vorliegenden Fall wird beispielsweise angenommen, dass als `data` die `_id` des `MediaItem` Objekts gesetzt wird, dessen Gelöschtsein durch das Ereignis signalisiert wird.

Die Tatsache, dass in der Leseansicht ein `MediaItem` gelöscht wurde, könnte dann durch die folgende Anweisung zum Ausdruck gebracht werden und würde durch den Benachrichtigungsmechanismus an alle daran interessierten View Controller mitgeteilt:

[Datei: `ReadviewViewController.js`]

```
this.viewProxy.bindAction("deleteItem", () => {
    mediaItem.delete(() => {
        this.notifyListeners(new
            mwf.Event("crud", "deleted", "MediaItem", mediaItem._id));
        this.previousView();
    })
});
```

Wenn Sie *nur* den oben gezeigten Aufruf von `addListener()` in `ListviewViewController` übertragen, werden Sie jedoch feststellen, dass bereits damit die gewünschte Funktionalität erbracht wird, d.h. dass nach Ausführung der ‘Löschen’-Aktion in der Leseansicht das betreffende `MediaItem` bei Rückkehr in die Listenansicht nicht mehr dargestellt wird. Die oben gezeigte explizite Auslösung des Ereignisses in `ReadviewViewController` ist dafür also gar nicht erforderlich. Grund

dafür ist die Tatsache, dass der von uns verwendete Entity Manager von MWF nach Ausführung von CRUD Operationen auf Entity-Objekten die betreffenden Ereignisse bereits selbst auslösen kann. Voraussetzung dafür ist, dass dieses Verhalten bei Registrierung der Entity mittels des unten hervorgehobenen booleschen Parameters als gewünscht angezeigt wurde - dies haben wir oben bereits getan, daher sei die betreffende Codezeile hier nur noch einmal wiedergibt:

[Datei: MyApplication.js]

```
GenericCRUDImplLocal.initialiseDB("mwftutdb", 1, ["MyEntity", "MediaItem"], () => {
    /*...*/
    this.registerEntity("MediaItem", entities.MediaItem, true);
    /*...*/
});
```

Erwähnt sei außerdem, dass die durch einen View Controller gesetzten Listener-Funktionen für Events normalerweise nur bzw. erst dann aufgerufen werden, wenn sich die durch den View Controller gesteuerte Ansicht im Vordergrund befindet. Im ‘Vordergrund’ befindet sich ein View Controller auch dann, wenn er durch einen Dialog überlagert wird, erst bei Aufruf einer Folgeansicht mittels `nextView()` tritt die aufrufende Ansicht in den Hintergrund und wird pausiert. Tritt in diesem Zustand ein Ereignis auf, für das der View Controller einen Listener registriert hat – wie der `ListViewController` für den Fall der Ausführung von ‘Löschen’ in der Leseansicht – erfolgt die Ausführung der Listener-Funktionen erst bei Ausführung der `onresume()` Funktion des View Controllers, d.h. bei Wiedereintritt der Ansicht in den Vordergrund.

MWF lässt es auch zu, die verzögerte Ausführung ereignisbehandelnder Funktionen zu unterbinden – zu diesem Zweck erlaubt `notifyListener()` einen dritten optionalen booleschen Parameter, dessen `true` Wert die unmittelbare Ausführung der Funktion bei Auftreten des Ereignisses erzwingt. Damit lässt sich beispielsweise ein View Controller, dessen Existenzberechtigung durch ein Ereignis hinfällig ist, als ‘obsolete’ markieren. Wird aus einer Leseansicht wie der unseren beispielsweise eine Editier-Ansicht mit Löschfunktion als Folgeansicht geöffnet, dann könnte die erneute Darstellung der Leseansicht bei Löschen des dargestellten Objekts auf diese Weise unterbunden werden (das Codebeispiel ist nicht zur Übernahme in die Tutorial-Anwendung gedacht):

```
this.addListener(new mwf.EventMatcher("crud", "deleted", "MediaItem"), ((event) => {
    this.markAsObsolete();
}), true);
```

Auf Basis der vorstehenden Erläuterungen können Sie die bestehende Anwendung wie nachfolgend gezeigt modifizieren und das Tutorial zu MWF ab-

schließen.

Registrieren Sie in ListviewViewController drei Event Listener für die Ereignisse created, updated und deleted und nehmen Sie in den Listener-Funktionen die erforderlichen Änderungen der Listenansicht vor.

Platzieren Sie die folgenden Aufrufe in die `oncreate()` Funktion von `ListviewViewController` vor den Aufruf von `readAll()` zur Initialisierung der Listenansicht:

[Datei: `ListviewViewController.js`]

```
oncreate(callback) {
    /* */
    this.addListener(new mwf.EventMatcher("crud", "created", "
        MediaItem"), ((event) => {
        this.addToListview(event.data);
    }));
    this.addListener(new mwf.EventMatcher("crud", "updated", "
        MediaItem"), ((event) => {
        this.updateInListview(event.data._id, event.data);
    }));
    this.addListener(new mwf.EventMatcher("crud", "deleted", "
        MediaItem"), ((event) => {
        this.removeFromListview(event.data);
    }));
    /* */
}
```

Erläuterungen:

- Hervorgehoben sind hier nur die Unterschiede der drei hinzuzufügenden Aufrufe von `addListener()`.
- Als Wert von `event.data` wird für die Ereignisse `created` und `updated` das erstellte bzw. modifizierte `MediaItem` Objekt übergeben, für `deleted` der Wert des `_id` Attributs des gelöschten Objekts.

Entfernen Sie nun in ListviewViewController aus den bestehenden Funktionen `createNewItem()`, `deleteItem()` und `editItem()` die darin enthaltenen Aufrufe von `addToListview()`, `updateInListview()` und `removeFromListview()` oder kommentieren Sie diese aus.

Erläuterungen:

- Auch innerhalb von `ListviewViewController` sind diese Funktionsaufrufe nicht mehr erforderlich, da alle CRUD Operationen bezüglich `MediaItem` unabhängig vom Ort des Aufrufs die im vorangegangenen Schritt registrierten Event Listener auslösen.

Wenn Sie nun in der gegenüber dem vorangegangenen Schritt unveränderten Leseansicht die ‘Löschen’-Operation ausführen, sollte das betreffende `MediaItem` Element nach Rückkehr in die Listenansicht nicht mehr angezeigt werden.

