O'REILLY®

2. Auflage

Moderne Webanwendungen mit ASP.NET MVC & JavaScript

ASP.NET MVC IM ZUSAMMENSPIEL MIT WEB APIS UND JAVASCRIPT-FRAMEWORKS

Manfred Steyer
Holger Schwichtenberg

Inhalt

KAPITEL 4

AngularJS

Das Erstellen JavaScript-getriebener Anwendungen gestaltet sich anspruchsvoll: Der Entwickler muss sich um das Binden von Daten, um das Aufrufen von Services sowie um das Validieren von Eingaben kümmern. Der Quellcode, der dabei entsteht, soll darüber hinaus auch überschaubar, wartbar und testbar sein. All dies ist zwar mit JavaScript möglich, allerdings erfordert dies viel Disziplin seitens der Entwickler und geht mit der Erstellung großer Mengen ähnlicher Codestrecken einher. JavaScript-Frameworks versprechen hier Abhilfe. Eines dieser Frameworks ist AngularJS, welches aus der Feder von Google stammt. Es zeichnet sich dadurch aus, dass es sehr viele Aspekte moderner JavaScript-basierter Anwendungen unterstützt und dabei auch die Kriterien Wartbarkeit und Testbarkeit in den Vordergrund stellt.

AngularJS herunterladen und einbinden

AngularJS findet sich samt einer umfangreichen Dokumentation unter *http://angularjs.org/*. Darüber hinaus kann der Entwickler AngularJS auch über NuGet beziehen. Da es sich bei AngularJS um ein sehr umfangreiches Framework handelt, wurde es auf mehrere Dateien aufgeilt. Abhängig davon, welche Möglichkeiten der Entwickler nutzen möchte, muss er mehr oder weniger dieser Dateien einbinden. Die Basisfunktionalität findet sich in der Datei *angular.js* wieder:

```
<script src="~/Scripts/angular.js"></script>
```

Die nachfolgenden Abschnitte gehen davon aus, dass diese Datei eingebunden ist. Sind zusätzlich weitere Dateien einzubinden, welche sich im Lieferumfang von AngularJS befinden, weisen die Abschnitte explizit darauf hin.

MVC, MVP und MVVM mit AngularJS

AngularJS basiert, so wie ASP.NET MVC, auf dem Architekturmuster Model-View-Controller. Häufig wird AngularJS jedoch als MV*-Framework bezeichnet. Damit soll hervorgehoben werden, dass es auch die Umsetzung anderer Muster unterstützt. Dazu zählt in erster Linie das Muster Model-View-Presenter (MVP), welches im Umfeld von WPF und Silverlight auch als Model-View-ViewModel bezeichnet wird. Dabei wird das Model auf ViewModels (bzw. bei MVP auf Presenter) abgebildet.

Diese repräesentieren das Model für eine bestimmte View und stellen die Präsentationslogik zur Verfügung.

Als Präsentationslogik wird hierbei zum Beispiel das Anstoßen von serverseitigen Routinen unter Verwendung der erfassten Daten sowie das Aktualisieren der GUI verstanden. View-Models haben jedoch auch die Aufgabe, Modelle für die Anzeige in bestimmten Views anzupassen. Beispielsweise könnten sie sich um die Bildung von Zwischensummen oder um das Verdichten von Modellen für eine kompaktere Darstellung kümmern.

Der Einsatz von View-Models erlaubt es, die Präsentationslogik von der UI, welche bei Webanwendungen mittels HTML realisiert wird, zu trennen. Dies erhöht die Wartbarkeit und die Testbarkeit der Präsentationslogik, welche entkoppelt vorliegt. Zur Verknüpfung der View-Models mit der UI kommt die deklarative Datenbindung zum Einsatz. Im Rahmen dessen gibt der Entwickler an, welche Eigenschaften des Presenters bzw. View-Models an welchen Stellen in der UI angezeigt werden sollen sowie welche Ereignisbehandlungsroutinen zum Aufruf welcher Funktion des View-Models führen sollen. Ein erstes Beispiel, welches diese Ideen mit AngularJS veranschaulicht, findet sich im nächsten Abschnitt.

Erste Schritte mit AngularJS

Zur Einführung in die Welt von AngularJS beschreibt dieser Abschnitt die Umsetzung einer einfachen Anwendung. Diese erlaubt die Suche nach Flügen sowie das Auswählen eines Flugs zur Erstellung einer Buchung.

Definition eines Moduls und Controllers

AngularJS strukturiert Anwendungen anhand von Modulen. Ähnlich, wie Namespaces unter .NET, kapselt ein Modul wiederverwendbare Programmteile und Konfigurationsinformationen. Zu diesen Programmteilen zählen Controller im Sinne des MVC-Musters. Bei Controllern handelt es sich unter AngularJS um Funktionen, welche Models (bzw. View-Models) bereitstellen. Diese Models werden anschließend von einer View visualisiert.

Beispiel 4-1 zeigt, wie der Entwickler ein Modul mit einem Controller bereitstellen kann. Dazu nutzt er die Funktion `module` des globalen Objekts `angular`, welches die von AngularJS bereitgestellten Konstrukte beinhaltet. Damit diese Funktion ein Modul einrichtet, übergibt der Entwickler den Namen des neuen Moduls sowie ein Array mit jenen Modulen, die in das neue Modul zu importieren sind. Durch die Möglichkeit, andere Module zu importieren, können Entwickler immer wiederkehrende Aufgaben in eigene Module auslagern. Da im betrachteten Beispiel keine Module (explizit) importiert werden sollen, übergibt das betrachtete Beispiel lediglich ein leeres Array. Übergibt der Entwickler neben dem Namen des Moduls keine weiteren Argumente, erzeugt `module` kein neues Modul, sondern gibt ein bestehendes Modul mit dem angegebenen Namen zurück, sofern ein solches existiert.

Genau genommen wird in jedes neue Modul das Modul `ng` importiert, welches die von AngularJS bereitgestellten Funktionalitäten enthält. Dies gilt auch dann, wenn – wie im hier gezeigten Beispiel – das Modul `ng` nicht explizit angegeben wird.

Die Funktion `module` gibt ein Objekt zurück, welches das jeweilige Modul beschreibt. Das betrachtete Beispiel nutzt die Funktion `controller` dieses Objekts, um einen neuen Controller einzurichten. Der

erste Parameter ist der Name des Moduls. Beim zweiten Parameter handelt es sich um eine Funktion, welche zum Ermitteln von Views zur Ausführung gebracht wird. Diese Funktion kann mit einer Action-Methode bei ASP.NET MVC verglichen werden.

Objekte, die von der Funktion benötigt werden, werden mithilfe von Parametern übergeben. Diese Objekte werden auch als Abhängigkeiten (engl. Dependencies) bezeichnet. Diese Abhängigkeiten werden beim Einsatz des Controllers nicht vom Entwickler, sondern von AngularJS übergeben. Hierbei ist auch von Dependency-Injection die Rede. Diesem Ausdruck zufolge injiziert AngularJS die Abhängigkeiten. Damit dies möglich ist, muss der Entwickler für diese Parameter wohlbekannte Namen verwenden. Im betrachteten Beispiel werden die Namen `$scope`, `$http` und `$q` verwendet. Der Parameter `$scope` zeigt auf ein Objekt, welches Variablen beinhaltet, welche in einem bestimmten Teil der Anwendung gültig sind. Die Aufgabe eines Controllers liegt im Bereitstellen von Informationen über dieses Objekt. Dabei handelt es sich um das Model im Sinne von MVC. Das betrachtete Beispiel nutzt die dynamische Natur von JavaScript, um im Objekt `$scope` eine neue Eigenschaft `vm` einzurichten. Diese verweist auf das zu verwendende View-Model. Dabei handelt es sich um ein `FlugBuchenVM`-Objekt.

Der Parameter `$http` bezeichnet einen Service, der im Lieferumfang von AngularJS enthalten ist, und auf einfache Weise den Zugriff auf Ressourcen via HTTP erlaubt. Im Kontext von AngularJS werden Services als wiederverwendbare Objekte bezeichnet, welche über Dependency Injection zu beziehen sind. Der dritte Parameter namens `$q` ist ein Service, welcher zur Erzeugung von Promises herangezogen werden kann und, wie der Name vermuten lässt, von der populären JavaScript-Bibliothek *Q* (siehe Kapitel 3) inspiriert wurde.

Beispiel 4-1: Deklaration eines Moduls mit einem einfachen Controller

```
var app = angular.module("Flug", []);

app.controller("FlugBuchenCtrl", function ($scope, $http, $q) {
    $scope.vm = new FlugBuchenVM($scope, $http, $q);
});
```

Da AngularJS über die Namen der verwendeten Parameter auf die zu injizierenden Services schließt, sind Probleme beim Einsatz von Minification (siehe Kapitel 1) vorprogrammiert. Der Grund dafür ist, dass im Zuge einer Minification für gewöhnlich die Namen von Variablen und Parametern durch kürzere Bezeichner ersetzt werden. Dies hat zur Folge, dass die für AngularJS benötigten Informationen verloren gehen. Um diese Informationen trotz Minification zu erhalten, kann der Entwickler diese in Form von Strings im Quellcode deponieren. Das Beispiel in Beispiel 4-2 veranschaulicht dies. Es übergibt mit dem zweiten Parameter von `controller` anstelle einer Funktion ein Array, welches aus den Namen der zu injizierenden Services besteht und an der letzten Stelle die vom Controller zu verwendende Funktion beinhaltet. Der erste Eintrag dieses Arrays beinhaltet den Namen des Services, welcher in den ersten Parameter der Funktion zu injizieren ist, der zweite Eintrag den Namen des Services, welcher in den zweiten Parameter der Funktion zu injizieren ist usw.

Beispiel 4-2: Angabe der Objektnamen über Strings

```
app.controller("FlugBuchenCtrl",
               ["$scope", "$http", "$q", function ($scope, $http, $q) {
        $scope.vm = new FlugBuchenVM($scope, $http, $q);
}]);
```

Deklaration der View-Models

Beispiel 4-3 und Beispiel 4-4 zeigen die vom betrachteten Beispiel verwendeten View-Models. Das ViewModel in Beispiel 4-3 repräsentiert einen Flug. Er wird initialisiert durch ein JavaScript-Objekt, das von einem HTTP-Service zur Verfügung gestellt wurde. Falls die Eigenschaft `Datum` nicht bereits als `Date` vorliegt, wird es unter Verwendung der Funktion `moment` aus der freien Bibliothek *moment.js*, welche Funktionen zum Arbeiten mit Datums-Werten bereitstellt, in ein `Date`-Objekt umgewandelt.

Das Kapitel 3 beschreibt, wie das JSON-Objekt, welches auch AngularJS zum Parsen von JSON-Dokumenten nutzt, angepasst werden kann, sodass es für Datumswerte `Date`-Objekte einrichtet.

Beispiel 4-3: View-Model für Flüge

```
function FlugVM(flug) {
    this.Id = flug.Id;
    this.Abflugort = flug.Abflugort;
    this.Zielort = flug.Zielort;

    if (typeof flug.Datum == "string") {
        this.Datum = moment(flug.Datum).toDate();
    }
    else {
        this.Datum = flug.Datum;
    }
}
```

Das View-Model in Beispiel 4-4 repräsentiert die gesamte Anwendung. Es nimmt den aktuellen Scope sowie den HTTP-Service und den Q-Service von AngularJS entgegen und hinterlegt es für die spätere Verwendung in entsprechenden Eigenschaften. Daneben bietet es eine Eigenschaft `fluege` an, die auf ein Array mit den abgerufenen Flügen verweist. Die Eigenschaft `selectedFlug` verweist auf jenen Flug, den der Benutzer ausgewählt hat, und `message` beinhaltet Informationen, wie Fehler- oder Statusmeldungen, die dem Benutzer anzuzeigen sind. Die Funktion `loadFluege` simuliert das Laden von Flügen, indem es zwei hartcodierte Flüge zum Array `fluege` hinzufügt. In einer weiter unten gezeigten Version dieses View-Models wird sich die Funktion `loadFluege` hingegen unter Verwendung der Eigenschaften `flugNummerFilter`, `flugVonFilter` und `flugNachFilter` zum Abrufen von Flügen an einen HTTP-Service wenden.

Die Funktion `selectFlug` nimmt den Index eines Flugs aus dem Array entgegen und platziert diesen Flug in der Eigenschaft `selectedFlug`.

Beispiel 4-4: View-Model für die gesamte Anwendung

```
function FlugBuchenVM(scope, http, q) {
    var that = this;

    this.fluege = new Array();

    this.scope = scope;
    this.q = q;
    this.http = http;

    this.selectedFlug = null;
```

```
        this.message = "";

        this.flugNummerFilter = "";
        this.flugVonFilter = "";
        this.flugNachFilter = "";

        this.loadFluege = function () {

            that.fluege.push(new FlugVM({
                                    Id: 1,
                                    Abflugort: "Graz",
                                    Zielort: "Essen",
                                    Datum: new Date().toISOString() }));

            that.fluege.push(new FlugVM({
                                    Id: 2,
                                    Abflugort: "Essen",
                                    Zielort: "Graz",
                                    Datum: new Date().toISOString() }));
        }

        this.selectFlug = function (idx) {
            var f = this.fluege[idx];
            this.selectedFlug = f;
        };

    }
```

Datenbindung verwenden

Um ein Model bzw. View-Model, welches von einem Controller über seinen Scope bereitgestellt wurde, an eine View zu binden, muss diese, wie in Beispiel 4-5 gezeigt, mit dem Controller verknüpft werden. Dazu verwendet der Entwickler das von AngularJS verwendete Attribut `ng-app` und `ng-controller`, wobei `ng-app` den Namen des Moduls und `ng-controller` den Namen des Controllers beinhaltet.

Diese von AngularJS bereitgestellten Attribute, welche auch mit auszuführenden JavaScript-Routinen verknüpft sind, werden als Direktiven bezeichnet. Innerhalb des Elements, welches mit `ng-controller` versehen wird, kann der Entwickler nun auf den vom Controller bestückten Scope zugreifen. Beispiel 4-6 veranschaulicht dies.

Beispiel 4-5: View mit Controller verknüpfen

```
<div ng-app="flug">

    <div ng-controller="FlugBuchenCtrl">

        […]

    </div>

</div>
```

Die Direktive `ng-show` legt fest, ob ein Element angezeigt werden soll. Wird der damit referenzierte Wert als `true` ausgewertet, kommt das Element zur Anzeige, auf das `ng-show` angewandt wurde; ansonsten nicht. In Beispiel 4-6 verweist `ng-show` auf die Eigenschaft `vm.message`. Das führt dazu, dass

das hier betrachtete div-Element nur angezeigt wird, wenn das hinter vm stehende View-Model über die Eigenschaft message eine Nachricht für den Benutzer enthält.

Der Bindungs-Ausdruck

```
{{ vm.message }}
```

bewirkt, dass diese Nachricht ebendort ausgegeben wird. Bei den darauffolgenden über das Element Input beschriebenen Textfeldern kommt die Direktive ng-model zum Einsatz, um die Eigenschaften flugNummerFilter, flugVonFilter und flugNachFilter des View-Models an diese Textfelder zu binden. Dabei handelt es sich um eine bidirektionale Datenbindung, d.h., Änderungen an den Eigenschaften werden ans Textfeld weitergegeben und umgekehrt. Die Direktive ng-click des darauffolgenden Buttons verknüpft den Click-Handler dieses Buttons mit der Methode loadFluege des View-Models. Dabei ist zu beachten, dass ng-click nicht nur den Namen der Funktion, sondern den gesamten Funktionsaufruf repräsentiert. Da es sich bei loadFluege um eine Funktion handelt, die keine Argumente erwartet, werden dem Funktionsnamen zwei runde Klammern nachgestellt. Eventuelle Parameter könnten, wie gewohnt, innerhalb dieser Klammern angeführt werden.

Neben ng-click bietet AngularJS weitere Direktiven, die es erlauben, Funktionen an JavaScript-Ereignisse zu binden. Darunter befindet sich zum Beispiel ng-change (Änderung des Werts eines Steuerelements), ng-blur (Steuerelement hat Focus verloren), ng-focus (Steuerelement hat Fokus erhalten) oder ng-checked (Checkbox wurde aktiviert bzw. deaktiviert). Einen Überblick über sämtliche Direktiven bietet die AngularJS-Referenz unter *http://docs.angularjs.org/api/*.

Die Tabelle in der zweiten Hälfte von Beispiel 4-6 verwendet die bereits beschriebene Direktive ng-show. Sie legt fest, dass die Tabelle nur angezeigt wird, wenn das Array fluege mindestens einen Eintrag enthält. Anstatt des Größer-Symbols (>), welches in HTML ein reserviertes Zeichen darstellt, kommt die HTML-Entität > (gt steht für »greater than«) zum Einsatz.

Die Direktive ng-repeat wiederholt das tr-Element der Tabelle für jeden Eintrag im Array fluege des View-Models. Der darin hinterlegte Ausdruck legt fest, dass der jeweils behandelte Flug in der Variablen f abzulegen ist, sowie dass der Index, den dieser Flug im Array einnimmt, über die Variable $index in Erfahrung gebracht werden kann.

Zusätzlich bewirkt die Direktive ng-class, dass das tr-Element mit der CSS-Klasse selected zu formatieren ist, wenn die ID des gerade behandelten Flugobjekts der ID des Flugobjekts in der Eigenschaft selectedFlug entspricht. Dies hat zur Folge, dass die Anwendung das markierte Objekt hervorgehoben darstellt.

Unter Verwendung von Datenbindungsausdrücken bindet danach die Anwendung die Eigenschaften Id, Datum, Abflugort und Zielort des jeweiligen Flugs an den Inhalt der einzelnen Zellen. Die Direktive ng-click bindet die Methode selectFlug des View-Models an das Click-Ereignis eines Links mit der Beschriftung Auswählen. Dabei wird festgelegt, dass der jeweilige Wert der Variablen $index an selectFlug zu übergeben ist. Damit der Link in jedem Browser angeklickt werden kann, ist ihm ein href-Attribut zu spendieren. Dieses hat im betrachteten Fall den Wert javascript:void(0). Dies bewirkt, dass keine Aktion ausgeführt wird, sodass bei einem Klick lediglich die hinter dem Click-Handler stehende Funktion zur Ausführung kommt.

Beispiel 4-6: Einfache View

```
<div>

    <div class="step-header">
        <h2>Flug auswählen</h2>
    </div>

    <div ng-show="vm.message" class="message">
        {{ vm.message }}
    </div>

    <div>Flugnummer</div>
    <div><input ng-model="vm.flugNummerFilter" /></div>

    <div>Von</div>
    <div><input ng-model="vm.flugVonFilter" /></div>

    <div>Nach</div>
    <div><input ng-model="vm.flugNachFilter" /></div>

    <div><input type="button" value="Suchen" ng-click="vm.loadFluege()" /></div>

    <div>

        <table ng-show="vm.fluege.length &gt; 0">
            <tr>
                <th>Id</th>
                <th>Abflugort</th>
                <th>Zielort</th>
                <th>Freie Plätze</th>
            </tr>
            <tr ng-repeat="f in vm.fluege track by $index"
                ng-class="{ selected: f.Id == vm.selectedFlug.Id }">

                <td>{{f.Id}}</td>
                <td>{{f.Datum | date:'shortDate'}}</td>
                <td>{{f.Abflugort}}</td>
                <td>{{f.Zielort}}</td>
                <td><a href="javascript:void(0)"
                       ng-click="vm.selectFlug($index)">Auswählen</a></td>
            </tr>
        </table>

    </div>

</div>
```

Der Vollständigkeit halber wird an dieser Stelle erwähnt, dass AngularJS auch Funktionen, die nicht explizit als Controller definiert wurden, als Controller heranziehen kann. Um auf solche Controller verweisen zu können, verwendet der Entwickler die Direktive `ng-app` ohne Angabe eines Modulnamens. Zur Lösung des zuvor diskutierten Problems, das beim Einsatz von Minification auftritt, kann der Entwickler der Controller-Funktion über die Eigenschaft `$inject` ein Array mit den Namen der zu injizierenden Services zuweisen.

Da durch diese Vorgehensweise das Modul-System von AngularJS umgangen wird, ist hiervon abzuraten.

```
<script src="~/Scripts/angular.js"></script>
<script>

    function SimpleCtrl($scope, $log) {
        $scope.info = {};
        $scope.info.message = "Hallo Welt!";
        $log.info("Hallo Welt!");
    }

    SimpleCtrl.$inject = ["$scope", "$log"];

</script>

<div ng-app>
    <div ng-controller="SimpleCtrl">
        {{info.message}}
    </div>
</div>
```

AngularJS näher betrachtet

Nachdem der Einsatz von AngularJS anhand eines einführenden Beispiels veranschaulicht wurde, geht dieser Abschnitt auf einige Konzepte von AngularJS genauer ein. Dabei handelt es sich um Direktiven, Datenbindung, Scopes und Dependency-Injection.

Direktiven

Bis jetzt wurden Direktiven ausschließlich in Form von Attributen genutzt. Dies ist jedoch nicht die einzige Art, Direktiven einzusetzen. Daneben können Direktiven auch in Form von HTML-Elementen, HTML-Kommentaren und als Klassen, welche über das Attribut `class` angegeben werden, genutzt werden. Dabei ist jedoch zu beachten, dass nicht zwangsläufig jede Form von jeder Direktive unterstützt wird. Beispiel 4-7 beinhaltet ein Beispiel, welches aus der Dokumentation von AngularJS entnommen wurde und die einzelnen Varianten veranschaulicht. Es zeigt auch, dass beim Einsatz einer Direktive als HTML-Element das direkte Übergeben eines Werts an die Direktive nicht unterstützt wird. In allen anderen Fällen wird der Wert `exp` übergeben.

Beispiel 4-7: Arten der Anwendung von Direktiven

```
<my-dir></my-dir>
<span my-dir="exp"></span>
<!-- directive: my-dir exp -->
<span class="my-dir: exp;"></span>
```

Am häufigsten findet man in der Praxis Direktiven, die als Attribute und Elemente genutzt werden, wobei die Nutzung als Element bei älteren Browsern, wie Internet Explorer 8, zu Problemen führt.

Wie Beispiel 4-8 veranschaulicht, existieren mehrere Möglichkeiten, eine Direktive als Attribut zu verwenden. Neben der bis dato verwendeten Variante, welche sich auf den häufig verwendeten Präfix `ng-` stützt, kann unter anderem auch das XHTML-Kompatible Präfix `ng:` sowie das HTML 5-kompatible

Präfix `data-` verwendet werden. Dies kann dann wichtig sein, wenn eine erfolgreiche formale Prüfung gegen XHTML oder HTML 5 angestrebt wird.

Beispiel 4-8: Möglichkeiten, Direktiven mithilfe von Attributen einzusetzen

```
<input ng-model="name">
<input ng:model="name">
<input ng_model="name">
<input data-ng-model="name">
<input x-ng-model="name">
```

Datenbindung

Zur Realisierung einer bidirektionalen Datenbindung verwendet AngularJS ein Konzept, das sich Dirty-Checking nennt. Das bedeutet, dass AngularJS bei bestimmten Ereignissen prüft, ob sich ein gebundener Wert geändert hat und dessen Anzeige in der GUI ggf. aktualisiert. Bei diesen Ereignissen kann es sich zum Beispiel um den Abschluss der Abarbeitung eines `Click`-Handlers handeln, der mit `ng-click` registriert wurde.

Obwohl dieses Verfahren auf den ersten Blick nicht sonderlich effizient erscheint, bestätigen zahlreiche Benachmarks diese Designentscheidung, während andere Frameworks, die bei jeder Änderung einer gebundenen Variablen ein Ereignis auslösen, schlechter abschneiden. Im Gegensatz zu diesen Frameworks muss AngularJS lediglich bei bestimmten Ereignissen aktiv werden und kann dann mehrere Datenbindungen auf einmal aktualisieren.

Das Dirty-Checking funktioniert bei AngularJS erstaunlich gut, sofern der Entwickler die Mechanismen von AngularJS verwendet. Beispielsweise stößt der Service `$http`, welcher den Zugriff auf HTTP-Services gestattet, nach dem Empfang von Daten das Dirty-Checking an. Somit wird die Anzeige von Daten, die durch den Service-Aufruf verändert wurden, aktualisiert.

Lediglich wenn Daten ohne Zutun von AngularJS aktualisiert werden, muss der Entwickler diesen Umstand explizit kommunizieren. Beispiele dafür sind das Eintreten von Ereignissen, mit denen die Geo-Location-API neue Koordinaten bekannt gibt oder mit denen die IndexedDB anzeigt, dass Daten asynchron geladen bzw. geschrieben wurden. Um in diesen Fällen AngularJS über die eventuelle Änderung von gebundenen Daten zu informieren, verwendet der Entwickler die Funktion `$digest`, welche über das `Scope`-Objekt angeboten wird:

```
$scope.$digest();
```

Alternativ dazu kann der Entwickler auch die jeweilige Routine, welche eventuell gebundene Daten aktualisiert, an `$apply` übergeben:

```
$scope.$apply(function() { /* Daten aktualisieren */ });
```

Diese Funktion hat den Vorteil, dass `$digest` auf jeden Fall aufgerufen wird – auch dann, wenn die übergebene Funktion eine Ausnahme auslöst.

Möchte sich der Entwickler direkt über eine Datenänderung, die im Zuge des Dirty-Checkings entdeckt wurde, informieren lassen, kann er einen Callback mit der Funktion `$watch` beim Scope registrieren:

```
scope.$watch('preis', function(newValue, oldValue) {  /* … */ });
```

Als ersten Parameter erwartet die Funktion `$watch` einen String mit dem Namen der zu überwachenden Eigenschaft. Bei dieser Eigenschaft muss es sich um eine Eigenschaft des Scopes handeln. Als zweiten Parameter erwartet sie den Callback, welcher bei einer Änderung der referenzierten Variablen ihren neuen sowie ihren ursprünglichen Wert übergeben bekommt.

Scopes

Direktiven können entweder den bereits vorhandenen Scope nutzen oder einen neuen Scope erstellen. Die im letzten Beispiel verwendete Direktive `ng-model` verwendet zum Beispiel den bereits existierenden Scope, welcher durch den Controller erstellt wird. In diesem Scope sucht `ng-model` nach der zu bindenden Eigenschaft.

Die Direktive `ng-repeat` erzeugt hingegen einen eigenen Scope pro Array-Eintrag, für den sie das jeweilige Element wiederholt. Im letzten Beispiel erhält dieser Scope die Eigenschaften `f` und `$index`, welche den aktuell bearbeiteten Flug sowie dessen Position innerhalb des Arrays repräsentieren. Darüber hinaus ist dieser Scope über die Prototypenkette (siehe Kapitel 3), welche in JavaScript zur Realisierung von Vererbung genutzt wird, mit dem übergeordneten Scope verbunden, den der Controller bereitstellt. An der Spitze dieser Kette steht ein sogenannter `$rootScope`, welcher für die gesamte Anwendung gilt.

Abbildung 4-1 veranschaulicht dies. Dadurch, dass die Scopes über die Prototypenkette verbunden sind, kann der Entwickler auch innerhalb des Scopes `ng-repeat` – sprich innerhalb des Elements, welches das Attribut `ng-repeat` aufweist – auf die Variable `vm`, welche sich im übergeordneten Scope befindet, lesend zugreifen. Würde jedoch die Direktive `ng-repeat` oder eine darin geschachtelte Direktive, welche den Scope von `ng-repeat` verwendet, auf die Variable `vm` schreibend zugreifen, würde sie eine Eigenschaft `vm` im Scope von `ng-repeat` einrichten. Diese Eigenschaft würde die gleichnamige Eigenschaft im Scope von `ng-controller` überschatten.

Anders gestaltet sich dieser Umstand jedoch, wenn der Entwickler im Scope von `ng-repeat` auf `vm.flugNummerFilter` zugreift. Unabhängig davon, ob der Zugriff lesend oder schreibend erfolgt, sucht JavaScript zunächst nach der Eigenschaft `vm` und wird im Scope von `ng-controller` fündig. Anschließend greift JavaScript bei dem gefundenen Objekt lesend oder schreibend auf die Eigenschaft `flugNummerFilter` zu. Das ist auch der Grund, warum es sich beim Einsatz von AngularJS eingebürgert hat, nicht die zu bindenden Eigenschaften direkt im Scope abzulegen, sondern stattdessen dort ein Objekt mit den zu bindenden Eigenschaften zu deponieren.

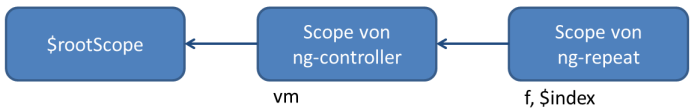

Abbildung 4-1: Scopes

Das Beispiel in Beispiel 4-9 veranschaulicht dies. Der Scope des Controllers erhält eine Eigenschaft `name` sowie eine Eigenschaft `person`, welche auf ein Objekt mit einer weiteren Eigenschaft `name` verweist. Darüber hinaus bekommt dieser Scope vom Controller ein Array mit Projektbezeichnungen spendiert. Die Direktive `ng-repeat` wiederholt ein `div`-Element pro Projekt. Neben dem Projektnamen beinhaltet dieses `div`-Elelement zwei Textfelder – das eine ist an `name` gebunden, das andere an `person.name`. Nach dem Start der Anwendung werden die erwarteten Namen in den einzelnen Textfeldern

angezeigt. Zwar befindet sich weder die Eigenschaft `name` noch die Eigenschaft `person.name` im Scope von `ng-repeat`, doch JavaScript sucht, nachdem es in diesem Scope nicht fündig wurde, im übergeordneten Scope, welcher der Scope von `ng-controller` ist, und wird dort fündig.

Ändert der Benutzer jedoch den Wert des ersten Textfelds im `div`-Element mit der Direktive `ng-repeat`, führt dies dazu, dass AngularJS im Scope, den `ng-repeat` für das jeweilige Projekt erzeugt hat, eine Eigenschaft `name` mit dem neuen Wert einrichtet. Somit wirkt sich diese Änderung nicht auf die anderen Stellen aus, welche ebenfalls mit der Eigenschaft `name` verknüpft sind.

Ändert der Benutzer hingegen den Namen im zweiten Textfeld, sucht AngularJS nach einer Eigenschaft `person` und wird im Scope des Controllers fündig. Daraufhin aktualisiert AngularJS die Eigenschaft `name` dieses Objekts und die Änderung wirkt sich auf alle Stellen aus, welche mit dieser Eigenschaft verknüpft sind.

Beispiel 4-9: Demonstration der Auswirkung der Verkettung von Scopes über Prototypen

```
<script src="~/Scripts/angular.js"></script>

<script>

    var app = angular.module("ScopeHorrorApp", []);

    app.controller("ScopeHorrorCtrl", function ($scope) {

        $scope.name = "Max Muster";

        $scope.person = {};
        $scope.person.name = "Susi Sorglos";

        $scope.projekte = ["Projekt-A", "Projekt-B", "Projekt-C"];

    });
</script>

<div ng-app="ScopeHorrorApp">

    <div ng-controller="ScopeHorrorCtrl">

        <input ng-model="name" /> <br />
        <input ng-model="person.name" />

        <div ng-repeat="p in projekte"
             style="border: 2px solid black; margin: 5px;">
            Projekt: {{ p }} <br />
            Projektleiter 1: <input ng-model="name" /> <br />
            Projektleiter 2: <input ng-model="person.name" />
        </div>

    </div>

</div>
```

Jedes `Scope`-Objekt bietet mit den Eigenschaften `$parent` und `$root` Zugriff auf den übergeordneten Scope sowie auf den Root-Scope.

Neben der Möglichkeit, einen neuen Scope zu schaffen, der mit dem übergeordneten Scope über die Prototypenkette in Beziehung steht, verfügen Direktiven auch über die Option, neue sogenannte isolierte Scopes zu schaffen. Dabei handelt es sich um Scopes, über die nicht auf übergeordnete Scopes oder nur auf bestimmte Eigenschaften von übergeordneten Scopes zugegriffen werden kann. Weitere Informationen hierzu finden Sie später in diesem Kapitel im Abschnitt »Benutzerdefinierte Direktiven«.

Dependency-Injection

Wie im letzten Abschnitt beschrieben, kommt bei AngularJS die Dependency-Injection zur Auflösung von Abhängigkeiten zum Einsatz. Hierdurch erhöht AngularJS die Testbarkeit von Anwendungen. Der Grund dafür liegt darin, dass der Entwickler im Rahmen von Unit-Tests die einzelnen Abhängigkeiten durch Test-Dummies, auch Mocks genannt, ersetzen kann. Diese Mocks verhalten sich nach außen hin wie die eigentlichen Abhängigkeiten, liefern jedoch spezielle für den Test benötigte Werte zurück. Wird beispielsweise der Service `$http` durch einen solchen Mock ausgetauscht, kann ein Controller, der sich im Produktivbetrieb auf HTTP-Services abstützt, ohne tatsächlichen Zugriff auf die jeweilige Webressource getestet werden. Dies macht die Ausführung des Tests schneller und erlaubt ein isoliertes Testen, zumal nun beim Testen des Controllers nicht der dahinterliegende HTTP-Service, sondern lediglich der jeweilige Mock mitgetestet wird.

Ein weiterer Vorteil ergibt sich, wenn der HTTP-Service auf eine Datenbank zugreift. Ohne Mock würde ein Unit-Test, der eigentlich nur den Controller testen soll, bei jeder Ausführung über den Service auf die Datenbank zugreifen. Das bedeutet auch, dass der Entwickler vor der Testausführung sicherstellen muss, dass sich bestimmte Testdaten in dieser Datenbank befinden. Durch den Einsatz eines Mocks, welches den Service- und somit auch den Datenbankzugriff für den Test simuliert, wird dieses Problem umgangen.

Die Tatsache, dass dieser Abschnitt die Vorteile von isolierten Unit-Tests hervorhebt, bedeutet nicht, dass man auf schichtenübergreifende Tests verzichten soll. Diese Tests, welche auch als End-2-End-Tests oder Integrationstests bezeichnet werden, tragen genauso wie Unit-Tests ihren Teil zu einem stabilen System bei. Aufgrund der Komplexität, die bei solchen Tests entsteht, beschränkt man sich bei Integrationstests in der Regel auf ausgewählte Szenarien, welche zeigen, dass die einzelnen Schichten bei den wichtigsten Use-Cases korrekt zusammenspielen.

Zur Realisierung solcher Tests stehen verschiedene Möglichkeiten zur Verfügung. Beispielsweise können hierzu die von Visual Studio 2013 ab der Premium Edition unterstützten Coded UI-Tests verwendet werden. Eine freie Lösung bietet hingegen das populäre Projekt Selenium (*http://docs.seleniumhq.org/*). Darauf aufbauend arbeitet die AngularJS-Gemeinde, welche sich nicht auf die Microsoft-Welt beschränkt, an einem Projekt namens Protractor (*https://github.com/angular/protractor*).

Um Tests unabhängig vom Browser zu gestalten, bietet AngularJS einige Services, die die Funktionalität von standardmäßig vorhandenen JavaScript-Objekten kapseln und somit gegen Mocks ausgetauscht werden können. So kann zum Beispiel das Anzeigen einer Alert-Box im Testfall unterdrückt oder Timeouts können verkürzt werden.

Beispielsweise kapseln die Services `$window` und `$document` die gleichnamigen JavaScript-Objekte `window` und `document`. Das Objekt `location`, welches unter anderem dem Entwickler die Möglichkeit bie-

tet, auf eine andere Seite zu wechseln und in Browsern über `window.location` zur Verfügung steht, wird durch den Service `$location` gekapselt. Die Möglichkeit zur Definition von Timeouts, welche von der Funktion `window.setTimeout` geboten wird, kapselt `AngularJS` über den Service `$timeout`.

Informationen über die vielen von AngularJS angebotenen Services finden sich unter *http://docs.angularjs.org/api*.

HTTP-Services via AngularJS konsumieren

Zum Zugriff auf HTTP-Services bietet AngularJS den Service `$http` an. Dieser Service funktioniert ähnlich wie die AJAX-Methoden in jQuery. Darüber hinaus besitzt er die angenehme Eigenschaft, nach dem asynchronen Erhalt von Daten die davon betroffenen Datenbindungen zu aktualisieren.

Das bereits bekannte Beispiel in Beispiel 4-10 veranlasst AngularJS dazu, den `$http`-Service in den Controller zu injizieren. Der Controller reicht diesen Parameter an das View-Model weiter, welches sich in Beispiel 4-11 befindet.

Beispiel 4-10: Service $http in Controller injizieren lassen

```
app.controller("FlugBuchenCtrl", function ($scope, $http, $q) {
    $scope.vm = new FlugBuchenVM($scope, $http, $q);
});
```

Die Funktion `get` des `$http`-Services veranlasst in Beispiel 4-11 den Zugriff auf einen HTTP-Service über das Verb GET. Als ersten Parameter nimmt sie den URL des Services entgegen und über den zweiten Parameter ein Objekt, welches den Aufruf genauer beschreibt. Die Eigenschaft `params` dieses Objekts verweist auf ein weiteres Objekt, dessen Eigenschaften als URL-Parameter übergeben werden. Um die notwendige URL-Kodierung kümmert sich AngularJS.

Das Ergebnis von GET ist ein Promise (siehe Kapitel 3), das die angestoßene asynchrone Anfrage repräsentiert. Die mit `then` registrierte Funktion nimmt das Ergebnis des Service-Aufrufs entgegen. Dabei handelt es sich um ein Objekt, welches über die Eigenschaft `data` die vom HTTP-Service zurückgegebenen Nutzdaten beinhaltet. Handelt es sich dabei um JSON-basierte Daten, beinhaltet `data` bereits ein JavaScript-Objekt, welches diesen Daten entspricht. AngularJS kümmert sich in diesem Fall also um das Parsen. Hierzu kommt das Browser-Objekt `JSON` zum Einsatz, weswegen der Entwickler das in Kapitel 3 gezeigte Verfahren verwenden kann, um den Parse-Vorgang anzupassen, sodass zum Beispiel Datumswerte als `Date`-Objekte dargestellt werden.

Neben der Eigenschaft `data` weist jenes Objekt, welches an die mit `then` registrierte Funktion übergeben wird, eine Eigenschaft `status` sowie eine Funktion `headers` auf. Die Eigenschaft `status` beinhaltet den HTTP-Status-Code der Antwort, zum Beispiel 200 für OK. Mit `headers` können die Kopfzeilen der Antwort-Nachricht entnommen werden. Der Aufruf

```
result.headers("content-type")
```

gibt zum Beispiel den Wert des Headers `content-type` zurück, der Aufschluss über das Format der zurückgegebenen Nutzdaten gibt.

Da die hier betrachtete Funktion `loadFluege` durch die Verwendung der asynchronen Methode `get` selbst einen asynchronen Charakter erhält, liefert sie dem Aufrufer ein Promise retour. Dieses Promise wird mit

dem AngularJS-Service $q, der der in Kapitel 3 behandelten gleichnamigen Bibliothek nachempfunden ist, erzeugt und genauso wie der $http-Service in den Controller injiziert (siehe Beispiel 4-10).

Beispiel 4-11: GET-Anfrage mit $http-Service

```
function FlugBuchenVM(scope, http, q) {
    var that = this;

    this.fluege = new Array();

    this.scope = scope;
    this.q = q;
    this.http = http;

    […]

    this.loadFluege = function () {
        var that = this;
        var deferred = this.q.defer();
        var params = {};

        if (that.flugNummerFilter) {
            params = {
                flugNummer: that.flugNummerFilter
            };
        } else {
            params = {
                abflugOrt: that.flugVonFilter,
                zielOrt: that.flugNachFilter
            };
        }

        this.http.get("/api/flug", { params: params }).then(function (result) {
            that.processFluege(result.data);
            deferred.resolve(result.data);
        }).catch(function (reason) {
            that.message = "Fehler: " + reason;
            deferred.reject(reason);
        });
    }

    this.processFluege = function (fluege) {
        this.fluege = new Array();

        if (!angular.isArray(fluege)) {
            fluege = [fluege];
        }

        for (var i = 0; i < fluege.length; i++) {
            var f = fluege[i];
            this.fluege.push(new FlugVM(f));
        }
    }
}
```

Ein weiteres Beispiel für den Einsatz des $http-Services findet sich in Beispiel 4-12. Hierbei handelt es sich im Gegensatz zum letzten Beispiel um eine POST-Anfrage, welche eine Flugbuchung erzeugt. Die

hier gezeigte Funktion buchen geht davon aus, dass sich der zu buchende Flug in der Eigenschaft selectedFlug und der betroffene Passagier in der Eigenschaft selectedPassagier befindet. Der erste Parameter repräsentiert hier den URL des HTTP-Services; der zweite Parameter ein in Form von JSON zu übersendendes JavaScript-Objekt. Als dritten Parameter könnte der Entwickler ein Objekt übergeben, welches die Anfrage genauer beschreibt und analog zu Beispiel 4-11 URL-Parameter und Header für den Aufruf beisteuert.

Beispiel 4-12: POST-Anfrage mit $http-Service

```
this.buchen = function () {
    var that = this;
    var deferred = this.q.defer();

    if (!this.selectedFlug || !this.selectedPassagier) {
        this.message = "Bitte wählen Sie einen Flug und einen Passagier aus!";
        return;
    }

    var buchung = {
        FlugID: that.selectedFlug.Id,
        PassagierID: that.selectedPassagier.Id
    };

    that.http.post("/api/buchung", buchung).then(function (result) {
        that.message = "Gebucht!";
        deferred.resolve(result.data);
    }).catch(function (reason) {
        that.message = "Der Flug konnte nicht gebucht werden: " + reason;
        deferred.reject(reason);
    });
};
```

Neben den pro Anfrage angegebenen HTTP-Parametern übersendet der $http-Service standardmäßig bei jeder Anfrage den Header

```
Accept: application/json, text/plain, * / *
```

Zusätzlich hängt er bei jeder POST- und PUT-Anfrage den Header Content-Type: application/json an. Das zeigt, dass AngularJS davon ausgeht, dass mit JSON gearbeitet wird. Möchte man diese Parameter ändern oder weitere Parameter definieren, die standardmäßig in die Nachrichten aufzunehmen sind, kann der Entwickler dies wie folgt bewerkstelligen:

```
$http.defaults.headers.common.Authentication = 'Basic bWF4OmdlaGVpbQ=='
```

Da in diesem Beispiel die Eigenschaft common verwendet wird, wird der angegebene Authentifizierungsheader, der Base64-kodiert Benutzername und Passwort enthält, unabhängig vom verwendeten Verb übersendet. Mehr Informationen zu diesem Header finden Sie in Kapitel 8, wo das Thema Sicherheit umfangreich behandelt wird. Möchte der Entwickler hingegen, dass bestimmte Header nur beim Einsatz bestimmter Verben übersendet werden, gibt er anstatt common das jeweilige Verb an, z.B. get, post, put oder delete.

Der Service-Zugriff über andere Verben gestaltet sich analog zu den hier gezeigten Beispielen, weshalb dafür an dieser Stelle auf die Online-Dokumentation von AngularJS (*http://docs.angularjs.org/api*) verwiesen wird.

Angular-Services bereitstellen und konsumieren

Services stellen in der von AngularJS bereitgestellten Infrastruktur wiederverwendbare Komponenten dar. Um einen eigenen Service bereitzustellen, kann der Entwickler, wie in Beispiel 4-13 gezeigt, die Funktion `factory` des jeweiligen Moduls heranziehen. Der erste Parameter repräsentiert den Namen des Services. Beim zweiten Parameter handelt es sich, analog zur Deklaration von Controllern, um ein Array. Der letzte Eintrag dieses Arrays ist eine Funktion, welche die gewünschte Serviceinstanz erzeugt und zurückgibt. Die anderen Einträge des Arrays beinhalten die Namen der Services, die in diese Funktion zu injizieren sind.

Anstatt des Arrays könnte auch nur die Funktion übergeben werden. In diesem Fall schließt AngularJS von den Namen der Parameter auf die zu injizierenden Services. Wie bereits im Abschnitt »Definition eines Moduls und Controllers« beschrieben, kommt es bei dieser etwas kürzeren Schreibweise beim Einsatz von Minification zu Problemen, da hierbei in der Regel die ursprünglichen Parameternamen durch kürzere ersetzt werden. Möchte der Entwickler also Minification anwenden, muss er, wie in Beispiel 4-13 gezeigt, auf das Array zurückgreifen.

Beispiel 4-13: Bereitstellen eines Services

```
var app = angular.module("Flug", []);

[…]

app.factory("flugService", ["$http", "$q", function ($http, $q) {
    return {

        load: function (params) {

            var deferred = $q.defer();

            $http.get("/api/flug", { params: params }).then(function (result) {
                deferred.resolve(result.data);
            }).catch(function (reason) {
                deferred.reject(reason);
            });

            return deferred.promise;
        }
    }

}]);
```

Beispiel 4-14 zeigt, dass ein benutzerdefinierter Service analog zu den von AngularJS bereitgestellten Services in Controller injiziert werden kann. Das hier gezeigte Beispiel injiziert neben dem in diesem Abschnitt gezeigten `flugService` einen analog aufgebauten `passagierService`.

Beispiel 4-14: Injizieren von benutzerdefinierten Services

```
app.controller("FlugBuchenCtrl", ["$scope", "$http", "$q", "flugService", "passagierService", function
($scope, $http, $q, flugService, passagierService) {
    $scope.vm = new FlugBuchenVM($scope, $http, $q, flugService, passagierService);
}]);
```

Der Vollständigkeit halber zeigt Beispiel 4-15, wie der vom Controller an das View-Model weitergereichte `flugService` verwendet werden kann.

Beispiel 4-15: Nutzung eines benutzerdefinierten Services

```
function FlugBuchenVM(scope, http, q, flugService, passagierService) {
    this.passagiere = new Array();
    this.state = "Passagier";
    this.buchungen = new Array();
    this.fluege = new Array();
    this.scope = scope;
    this.q = q;
    this.http = http;
    this.flugService = flugService;
    this.passagierService = passagierService;

    this.loadFluege = function () {
        var that = this;

        var deferred = this.q.defer();

        var params = {};

        if (that.flugNummerFilter) {
            params = {
                flugNummer: that.flugNummerFilter
            };
        } else {
            params = {
                abflugOrt: that.flugVonFilter,
                zielOrt: that.flugNachFilter
            };
        }

        flugService.loadFluege(params).then(function (result) {

            that.processFluege(result);
            deferred.resolve(result);

        }).catch(function (reason) {
            that.message = "Fehler: " + reason;
            deferred.reject(reason);
        });

    };

    […]
}
```

Filter in AngularJS

Mit Filter kann der Entwickler Daten im Zuge der Datenbindung bearbeiten. Dazu gehört unter anderem das Formatieren, Filtern oder Sortieren von darzustellenden Informationen. Dieser Abschnitt zeigt, wie Sie Filter, die im Lieferumfang von AngularJS enthalten sind, verwenden, sowie, wie Sie AngularJS um eigene Filter erweitern können.

Filter verwenden

Um einen Filter auf eine zu bindende Eigenschaft anzuwenden, hängt der Entwickler, wie in Beispiel 4-16 veranschaulicht, eine Pipe gefolgt vom Namen des Filters an.

Beispiel 4-16: Mit dem Filter date ein Datum formatieren

```
<td>{{p.Geburtsdatum | date:'shortDate' }}</td>
```

Falls der Filter, wie im betrachteten Fall, Parameter erwartet, werden diese getrennt durch jeweils einen Doppelpunkt angehängt. Ein Filter mit drei Parametern würde somit wie folgt verwendet werden:

```
{{ eigenschaft | filter:param1:param2:param3 }}
```

Das Ergebnis eines Filters kann an als Eingabe für einen weiteren Filter verwendet werden. Dazu hängt der Entwickler an den Ausdruck erneut eine Pipe, gefolgt vom Filteraufruf:

```
{{ eigenschaft | filter1:param1:param2:param3 | filter2:param }}
```

Wie der Name des Filters in Beispiel 4-16 vermuten lässt, formatiert er ein Datum. Der Parameter gibt Aufschluss über das Format des Datums. Der hier eingesetzte Wert `shortDate` verwendet eine kompakte Darstellung ohne Ausgabe der Uhrzeit. Um der Lokalisierung von Datumsformaten gerecht zu werden, kann AngularJS dahingehend parametrisiert werden, dass eine Datei mit Formatierungsanweisungen eingebunden wird. Im Lieferumfang von AngularJS befinden sich im Ordner `i18n` zahlreiche Dateien, welche die Formate einzelner Kulturen beinhalten. Der Begriff Kultur ist in diesem Umfeld als Kombination von Sprache und Region zu verstehen. Beispiele dafür sind Deutsch/Deutschland (de-DE), Deutsch/Österreich (de-AT) und Englisch/Großbritannien (en-GB).

Bindet der Entwickler zum Beispiel die Datei `i18n\angular-locale_de-de.js` ein, bewirkt der in Beispiel 4-16 gezeigte Aufruf des Filters `date` mit dem Parameter `shortDate` die Verwendung des in Deutschland üblichen Datumsformats `Tag.Monat.Jahr` (z.B. 5.4.2063). Durch das dynamische Einbinden von Ressourcedateien kann der Entwickler somit die Darstellung von Datumswerten anpassen.

Neben Formaten für Datumswerte beinhalten die genannten Dateien auch die in der jeweiligen Kultur üblichen Formate für die Darstellung von Zahlen. Zum Formatieren von Zahlen bietet AngularJS analog zum Filter `Date` einen Filter `number`. Ein optionaler Parameter gibt Aufschluss über die Anzahl der anzuzeigenden Dezimalstellen. Ein Aufruf von

```
{{ eigenschaft | number:2 }}
```

würde beispielsweise eine Zahl, die sich in der Eigenschaft `eigenschaft` befindet, wie in der jeweiligen Kultur üblich, mit zwei Nachkommastellen darstellen (z.B. 14.763,95 bei de-DE).

Die hier betrachteten Möglichkeiten zur Internationalisierung beschränken sich in der betrachteten AngularJS-Version 1.2 auf die Ausgabe von Daten. Für das Erfassen von Daten im jeweiligen Format der gewählten Kultur existiert derzeit noch keine direkte Unterstützung. Aus diesem Grund wird weiter unten gezeigt, wie benutzerdefinierte Direktiven, welche Eingaben mit dem Framework Globalize (siehe Kapitel 3) validieren, entwickelt werden können. Für eine durchgängige Nutzung von Globalize zeigt dieses Kapitel auch, wie benutzerdefinierte Filter, welche Globalize kapseln, bereitgestellt werden können.

Ein weiterer Filter, der an dieser Stelle vorgestellt wird und unter anderem beim Troubleshooting nützlich sein kann, ist der Filter `json`: Er gibt einen JSON-String zurück, welcher das formatierte Objekt beschreibt:

```
{{ eigenschaft | json }}
```

Neben Filtern, die der Entwickler auf einzelne Werte anwenden kann, bietet AngularJS auch Filter, die für Arrays bestimmt sind. Ein Beispiel dafür bietet Beispiel 4-17. Es filtert und sortiert die Passagierobjekte in der gebundenen Eigenschaft `vm.passagiere` und beschränkt anschließend die Ausgabe auf die ersten zehn.

Das Objekt, welches das betrachtete Beispiel an den ersten Parameter übergibt, bestimmt, welche Eigenschaften der Passagierobjekte nach welchen Kriterien zu filtern sind. Das verwendete Objekt bewirkt, dass die Namen der Passagiere gegen den Wert `'W'` zu prüfen sind. Der zweite Parameter von `filter` informiert über die Art dieser Prüfung. `False` bedeutet, dass lediglich zu prüfen ist, ob der definierte Wert (hier `'W'`) innerhalb des Namens vorkommt. `True` würde bedeuten, dass der Name genau dem angegebenen Wert zu entsprechen hat. Daneben kann der Entwickler auch eigene Funktionen bereitstellen, die die hier eingesetzte Prüfung beeinflussen.

Der Filter `orderBy` nimmt ein Array mit jenen Eigenschaftsnamen entgegen, nach denen der Entwickler sortieren möchte. Standardmäßig wird absteigend sortiert. Stellt der Entwickler dem Namen einer Eigenschaft ein Minus voran, sortiert `orderBy` hingegen absteigend. Möchte der Entwickler nur nach einem einzigen Wert sortieren, kann er an `orderBy` auch anstatt eines Arrays einen String mit dem Namen der gewünschten Eigenschaft übergeben.

Der Filter `limitTo` bewirkt, dass nach dem Filtern und Sortieren lediglich die ersten zehn Passagiere herangezogen werden.

Beispiel 4-17: Filter auf Arrays anwenden

```
<tr ng-repeat="p in vm.passagiere | filter:{Name:'W'}:false |
orderBy:['-Vorname', 'PassagierStatus'] | limitTo:10 track by $index"
ng-class="{ selected: p.Selected }">
    <td>{{p.Vorname}}</td>
    <td>{{p.Name}}</td>
    <td>{{p.Geburtsdatum | date:'shortDate' }}</td>
    <td>{{p.PassagierStatus}}</td>
    <td><a href="#" ng-click="vm.select($index)">Auswählen</a></td>
</tr>
```

All diese Filter bieten zahlreiche Parameter, mit denen der Entwickler ihr Verhalten anpassen kann. Informationen darüber, sowie Informationen über sämtliche im Lieferumfang von AngularJS enthaltenen Filter, finden sich in der Online-Dokumentation unter *http://docs.angularjs.org/api*.

Benutzerdefinierte Filter bereitstellen

Um eigene Filter zu definieren, verwendet der Entwickler die Funktion `filter`, welche AngularJS auf Modulebene bereitstellt. Beispiel 4-18 demonstriert dies anhand eines benutzerdefinierten Filters `globalize`, der an die Bibliothek `Globalize` (siehe Kapitel 3) delegiert. Der erste Parameter von `filter` gibt den Namen des neuen Filters an. Beim zweiten Parameter handelt es sich um eine Funktion, welche eine Funktion zurückgibt, die den gewünschten Filter repräsentiert. Diese Funktion hat mindestens einen Parameter. Dabei handelt es sich um den zu formatierenden Wert. Alle weiteren Parameter ent-

sprechen jenen Parametern, die beim Anwenden des Filters getrennt durch Doppelpunkte angehängt werden.

Beispiel 4-18: Benutzerdefinierten Filter definieren

```
var app = angular.module("Flug", []);
[…]
app.filter('globalize', function () {
    return function (input, format) {
        return Globalize.format(input, format);
    }
});
```

Vor dem Einsatz dieses Filters sollte der Entwickler, wie in Kapitel 3 gezeigt, die gewünschte Kultur setzen:

```
Globalize.culture("de-AT");
```

Das nachfolgende Beispiel zeigt, wie der Entwickler den in Beispiel 4-18 definierten Filter anwenden kann:

```
{{ vm.datum | globalize }}
{{ vm.maxPassagiere | globalize:'N4' }}
```

Mit Formularen arbeiten

AngularJS bietet einige Direktiven, um Daten und Funktionen an Steuerelemente, wie Formularfelder, Kontrollkästchen (engl. Checkbox) oder Schaltflächen, zu binden. Darüber hinaus bietet AngularJS eine Unterstützung für die Validierung und Formatierung von Eingaben. Dieser Abschnitt geht darauf ein und zeigt, wie der Entwickler eigene Validierungslogiken für AngularJS implementieren kann.

Objekte an Formularfelder binden

Um zu demonstrieren, auf welche Weise Daten mit AngularJS an die einzelnen von HTML angebotenen Steuerelemente gebunden werden können, verwenden die nachfolgenden Abschnitte den Controller in Beispiel 4-19. Dieser spendiert dem Scope eine Eigenschaft `vm`, welche auf ein Objekt verweist, das als View-Model fungiert und über eine Eigenschaft `flug` ein Flugobjekt anbietet. Die Eigenschaft `Airlines` dieses View-Models beinhaltet darüber hinaus Vorschlagswerte für die Auswahl einer Airline bei der Verwaltung eines Flugs.

Beispiel 4-19: Controller zur Demonstration des Einsatzes von Formularen

```
var app = angular.module("flug", []);

app.controller("editFlugCtrl", function ($scope) {

    $scope.vm = {};

    $scope.vm.airlines = [
        { id: "LH", name: "Lufthansa", allianz: "Star Alliance" },
        { id: "AUA", name: "Austrian", allianz: "Star Alliance" },
        { id: "S", name: "Swiss", allianz: "Star Alliance" },
        { id: "NIKI", name: "Fly Niki", allianz: "oneworld" },
        { id: "AB", name: "Air Berlin", allianz: "oneworld" },
```

```
    ];

    $scope.vm.flug = {
        flugNummer: "LH4711",
        datum: new Date(),
        preis: 400.90,
        maxPassagiere: 300,
        verspaetet: false,
        airline: "LH"
    };

    $scope.vm.flug.save = function () {
         // Hier könnte nun gespeichert werden!
    }

});
```

Beispiel 4-20 zeigt, wie der Entwickler ein Textfeld sowie eine Checkbox an einen Wert des Models binden kann. In beiden Fällen verweist das Beispiel über die Direktive `ng-model` auf die zu bindende Eigenschaft. Wird der Wert, den es an die Checkbox bindet, als `true` ausgewertet, aktiviert AngularJS die Checkbox.

Beispiel 4-20: Daten an eine Checkbox binden

```
<div ng-app="flug">

    <form ng-controller="editFlugCtrl" name="form" id="form">

        <div>FlugNummer</div>
        <div><input ng-model="vm.flugNummer" name="flugNummer" /></div>


        <div>Verspätet</div>
        <div>
            <input type="checkbox" ng-model="vm.flug.verspaetet">
        </div>

        […]

    </form>
</div>
```

Ein Beispiel für das Binden von Optionsfeldern (engl. Radio Button) findet sich in Beispiel 4-21. Auch dieses Beispiel verweist über die Direktive `ng-model` auf die zu bindende Eigenschaft. Dabei fällt auf, dass alle Optionsfelder denselben Namen aufweisen. Das ist notwendig, damit der Browser den Zusammenhang zwischen diesen Optionsfeldern erkennt und nur die Auswahl einer einzigen Option zulässt. Wird ein Optionsfeld aktiviert, schreibt AngularJS jenen Wert, den der Entwickler mit dem Attribut `value` angegeben hat, in die gebundene Eigenschaft.

Beispiel 4-21: Daten an Optionsfelder binden

```
<div>
    Airline
</div>
<div>
    <input type="radio" ng-model="vm.flug.airline" name="airline" value="LH" /> Lufthansa <br/>
    <input type="radio" ng-model="vm.flug.airline" name="airline" value="AUA" /> Austrian <br />
```

```
        <input type="radio" ng-model="vm.flug.airline" name="airline" value="S" /> Swiss <br />

        <div>
            Selected: {{ vm.flug.airline }}
        </div>

    </div>
```

Möchte der Entwickler die Optionsfelder dynamisch anhand einer Liste von Vorschlagswerten generieren, kann er ein Array mit Vorschlagswerten mit der Direktive `ng-repeat` iterieren. Beispiel 4-22 demonstriert dies.

Beispiel 4-22: Daten an dynamisch generierte Optionsfelder binden

```
<div>
    Airline
</div>
<div>

    <div ng-repeat="a in vm.airlines">
        <input type="radio" ng-model="vm.flug.airline" name="airline" value="{{a.id}}" /> {{ a.name }}
    </div>

    <div>
        Selected: {{ vm.flug.airline }}
    </div>
</div>
```

Für das Generieren eines Dropdownfeldes bietet AngularJS die Direktive `ng-options` an. Diese Direktive erwartet einen Ausdruck, der einer simplen vorgegebenen Grammatik folgt. Beispiel 4-23 demonstriert dies, indem es ein Dropdownfeld generiert, das pro Eintrag im Array `vm.airlines` eine Option anbietet; als Beschriftung kommt die Eigenschaft `name` zum Einsatz, als Wert die Eigenschaft `id`.

Beispiel 4-23: Daten an ein Dropdownfeld binden

```
<div>
    Airline
</div>
<div>

    <select ng-model="vm.flug.airline" ng-options="a.id as a.name for a in vm.airlines"></select>

    <div>
        Selected: {{ vm.flug.airline }}
    </div>
</div>
```

Erweitert der Entwickler den Ausdruck in `ng-options` um eine `group-by`-Klausel, generiert AngularJS ein Dropdownfeld, welches die Einträge nach der in dieser Klausel angegebenen Eigenschaft gruppiert. Auf diese Weise gruppiert das Beispiel in Beispiel 4-24 die Vorschlagswerte nach der Eigenschaft `allianz`.

Beispiel 4-24: Dropdownfeld mit Gruppen

```
<div>
    Airline
</div>
<div>
```

```
    <select ng-model="vm.flug.airline" ng-options="a.id as a.name group by a.allianz for a in vm.
airlines"></select>

    <div>
        Selected: {{ vm.flug.airline }}
    </div>
</div>
```

Form-Controller

Für jedes Formular innerhalb einer AngularJS-Anwendung erstellt AngularJS im aktuellen Scope einen sogenannten Form-Controller. Die Eigenschaft des Scopes, welche auf den Form-Controller verweist, hat jenen Namen, den der Entwickler über das Attribut `name` dem Formular spendiert. Unter anderem weist der Form-Controller eine Eigenschaft `$dirty` auf. Diese zeigt an, ob der Bentuzer ein Feld des Formulars geändert hat. Genau das Gegenteil davon zeigt die Eigenschaft `$pristine` an, welche ebenfalls jeder Form-Controller anbietet. Darüber hinaus hat der Form-Controller für jedes Steuerelement des jeweiligen Formulars eine Eigenschaft, die denselben Namen wie das Steuerelement trägt. Diese Eigenschaften verweisen auf Objekte, die wiederum über eine Eigenschaft `$dirty` verfügen. Diese zeigt an, ob das Formularfeld geändert wurde. Daneben zeigt auch hier `$pristine` das Gegenteil davon an. Das Beispiel in Beispiel 4-25 demonstriert dies, indem es die genannten Eigenschaften sowohl auf Formular- als auch auf Steuerelementebene anzeigt.

Beispiel 4-25: Nutzung des Form-Controllers

```
<div ng-app="flug">

    <form novalidate ng-controller="editFlugCtrl" name="form" id="form">

        <div>FlugNummer</div>
        <div><input ng-model="vm.flugNummer" name="flugNummer"/></div>

        <div>
            <b>Dirty (form):</b> {{ form.$dirty }}
        </div>
        <div>
            <b>Pristine (form):</b> {{ form.$pristine }}
        </div>

        <div>
            <b>Dirty (flugNummer):</b> {{ form.flugNummer.$dirty }}
        </div>
        <div>
            <b>Pristine (flugNummer):</b> {{ form.flugNummer.$pristine }}
        </div>


        […]

    </form>
</div>
```

Neben den hier genannten Eigenschaften weist der Form-Controller für das Formular sowie für die einzelnen Steuerelemente auch Eigenschaften auf, die darüber informieren, ob das Formular bzw. das

Steuerelement erfolgreich validiert werden konnte bzw. welche Validierungsfehler aufgetreten sind. Diese Eigenschaften werden im nächsten Abschnitt besprochen.

Eingaben validieren

Zum Validieren von Eingaben bietet AngularJS einige vordefinierte Validierungslogiken. Diese werden unter anderem durch das Setzen des Attributs `type` bei den verwendeten Input-Elementen aktiviert. Die hier betrachtete Version unterstützt die folgenden Werte für `type`: `text`, `number`, `url`, `email`.

Darüber hinaus kann der Entwickler mit der Eigenschaft `required` angeben, ob es sich bei einem Feld um ein Pflichtfeld handelt und mit den Direktiven `ng-minlength` und `ng-maxlength` kann er die minimale und maximale Länge des eingegebenen Texts definieren. Daneben steht noch eine Direktive `ng-pattern` zur Verfügung, welche die Eingabe anhand eines regulären Ausdrucks validiert. Beispiel 4-26 demonstriert den Einsatz dieser Möglichkeiten. Es verwendet das Attribut `novalidate`, welches eine eventuelle Validierung, die der Browser auf eigene Faust durchführt – zum Beispiel unter Berücksichtigung des Attributs `type` –, unterbindet. Dieses Attribut wirkt sich jedoch nicht auf die Validierung durch AngularJS aus. Beachtenswert ist hier, dass der an `ng-pattern` übergebene reguläre Ausdruck, wie bei JavaScript üblich, zwischen zwei Schrägstrichen zu platzieren ist.

Das betrachtete Beispiel prüft auch unter Verwendung des Form-Controllers, ob das Steuerelement erfolgreich validiert werden konnte. Dazu greift es auf die Eigenschaft `form.flugNummer.$invalid` zu. Eine Fehlermeldung gibt es jedoch nur dann, wenn der Benutzer zusätzlich den Wert im Eingabefeld geändert hat. Dieser Umstand wird über die Eigenschaft `form.flugNummer.$dirty` geprüft. Weisen diese beiden Eigenschaften den Wert `true` auf, gibt das Beispiel eine Fehlermeldung aus.

Neben der Eigenschaft `$invalid` existiert auch eine Eigenschaft `$valid`, welche genau das Gegenteil davon ausdrückt. Beide Eigenschaften existieren auch direkt im Form-Controller. Dort geben sie darüber Auskunft, ob sämtliche Felder im Formular erfolgreich validiert werden konnten.

Darüber hinaus wird ebenfalls geprüft, welche Validierungslogik fehlgeschlagen ist. Dazu wird auf die Eigenschaft `form.flugNummer.$error` zugegriffen. Diese verweist auf ein Objekt, welche pro Validierungslogik eine Eigenschaft aufweist. Diese nennen sich im betrachteten Fall passend zu den verwendeten Attributen bzw. Direktiven `required`, `maxlength`, `minlength` und `pattern`. Weisen diese Eigenschaften den Wert `true` auf, konnte AngularJS den erfassten Wert mit der assoziierten Validierungslogik nicht erfolgreich validieren.

Beispiel 4-26: Textfelder validieren

```
<div ng-app="flug">

    <form ng-controller="editFlugCtrl" name="form" novalidate>

        <div>FlugNummer</div>
        <div><input ng-model="vm.flugNummer" name="flugNummer"
                    required ng-minlength="6" ng-maxlength="10" ng-pattern="/\w+\d+/" /></div>

        <div ng-show="form.flugNummer.$dirty && form.flugNummer.$invalid">
            Flugnummer ist nicht gültig!
        </div>
        <div ng-show="form.flugNummer.$dirty && form.flugNummer.$error.required">
            Pflichtfeld!
        </div>
```

```
            <div ng-show="form.flugNummer.$dirty && form.flugNummer.$error.maxlength">
                Maximal 10 Zeichen!
            </div>
            <div ng-show="form.flugNummer.$dirty && form.flugNummer.$error.minlength">
                Mindestens 6 Zeichen!
            </div>
            <div ng-show="form.flugNummer.$dirty && form.flugNummer.$error.pattern">
                Mindestens ein Buchstabe gefolgt von meindestens einer Ziffer, z.B. LH4711
            </div>
            <div>
                form.flugNummer.$error: {{ form.flugNummer.$error | json }}
            </div>

            […]

        </form>
    </div>
```

Für Input-Felder des Typs `number` kann der Entwickler darüber hinaus eine untere sowie eine obere Schranke definieren. Das Beispiel in Beispiel 4-27 demonstriert dies, indem es dem `Input`-Element eine Eigenschaft `min` sowie eine Eigenschaft `max` spendiert. Die Handhabung dieser Attribute entspricht jener der zuvor betrachteten Direktiven. Das Ergebnis der Validierung durch die damit verbundenen Logiken findet sich im betrachteten Fall in den Eigenschaften `form.preis.$error.min` bzw. `form.preis.$error.max`.

Beispiel 4-27: Zahlen validieren

```
<div>Preis</div>
<div><input ng-model="vm.preis" name="preis" type="number" min="100" max="2000" /></div>
<div ng-show="form.preis.$dirty && form.preis.$invalid">
    Preis muss zwischen 0 und 2000 liegen!
</div>
<div>
    form.preis.error: {{ form.preis.$error | json }}
</div>
```

Fehlerhafte Eingaben mit CSS hervorheben

Damit Eingabefelder in Hinblick auf die Validität ihrer Inhalte formatiert werden können, weist ihnen AngularJS abhängig von ihrem Zustand vordefinierte CSS-Klassen zu. Indem der Entwickler Formatierungsanweisungen für diese Klassen einrichtet, kann er zum Beispiel unterschiedliche Hintergrundfarben für korrekt und nicht korrekt validierte Textfelder anbieten.

Felder, deren Inhalte korrekt validiert werden konnten, erhalten die Klasse `ng-valid`. Felder, bei denen das nicht der Fall ist, erhalten die Klasse `ng-invalid`. Die Klasse `ng-dirty` wird hingegen vergeben, wenn der Inhalt des Felds geändert wurde.

Zur Veranschaulichung der damit verbundenen Möglichkeiten definiert Beispiel 4-28 zwei Formatierungen mit CSS. Die erste gilt für Input-Elemente, die sowohl die Klassen `ng-invalid` als auch `ng-dirty` aufweisen. Die zweite gilt hingegen für Input-Elemente, die sowohl die Klassen `ng-valid` als auch `ng-dirty` aufweisen. Auf diese Weise erhalten Felder, deren Inhalte verändert und nicht korrekt validiert wurden, die Hintergrundfarbe rot. Felder, deren Inhalte verändert und korrekt validiert wurden, werden hingegen mit einer grünen Hintergrundfarbe angezeigt.

Beispiel 4-28: CSS-Klassen zur Formatierung von Steuerelementen

```
input.ng-invalid.ng-dirty {
    background-color: red;
}

input.ng-valid.ng-dirty {
    background-color: green;
}
```

Auf Validierungsergebnisse programmatisch zugreifen

Um zur Laufzeit zu ermitteln, ob ein Formular veränderte Werte beinhaltet bzw. korrekt validiert werden konnte, greift der Entwickler über den Scope auf den mit dem Formular assoziierten Form-Controller zu. Beispiel 4-29 demonstriert dies.

Beispiel 4-29: Programmatisch prüfen, ob Eingaben valide sind

```
$scope.vm.flug.save = function () {
    if (!$scope.form.$dirty) {
        alert("Mensch, Du hast ja gar nix geändert!");
        return;
    }
    if ($scope.form.$invalid) {
        alert("Validierungsfehler!");
        return;
    }

    // Hier könnte nun gespeichert werden!
}
```

Auf dieselbe Weise kann der Entwickler auch herausfinden, welche Validierungslogik welches Feld nicht korrekt validieren konnte. Die Funktion in Beispiel 4-30 demonstriert dies. Sie nimmt den Namen des Formulars und somit auch den Namen des damit assoziierten Form-Controllers entgegen. Übergibt der Aufrufer keinen Formularnamen, geht sie vom Formularnamen `form` aus. Über diesen Namen adressiert sie den Form-Controller im Scope und iteriert dessen Eigenschaften. Dabei geht die betrachtete Funktion davon aus, dass sämtliche Eigenschaften, deren Namen nicht mit `$` beginnen, für Eingabefelder stehen. Weist deren Eigenschaft `$invalid` darauf hin, dass ein Validierungsfehler aufgetreten ist, durchläuft die Funktion die Eigenschaften des Objekts hinter der Eigenschaft `$error`. Diese Eigenschaften weisen die Namen der Validierungslogiken auf (z.B. `required`, `pattern`, `min`, `max`) und haben den Wert `true`, wenn die jeweilige Validierungslogik den erfassten Wert nicht korrekt validieren konnte.

Somit werden die Namen der Felder mit Validierungsfehlern sowie pro Feld die Bezeichner der fehlgeschlagenen Validierungslogiken ausgegeben. In einer Implementierung für den Produktiveinsatz könnte der Entwickler zum Beispiel Letztere auf sprechende Beschreibungen der einzelnen Validierungslogiken umschlüsseln.

Beispiel 4-30: Programmatisch auf Validierungsergebnisse zugreifen

```
$scope.vm.flug.validationSummary = function (formName) {
    var result = "";

    if (!formName) formName = "form"; // Standardwert: form
```

```
        var form = $scope[formName];
        for (var ctrlName in form) {
            if (ctrlName.substr(0, 1) == "$") continue;
            var ctrl = form[ctrlName];

            if (ctrl.$invalid) {
                result += ctrl.$name + ": ";
                for (var error in ctrl.$error) {
                    result += error + " ";
                }
                result += "\n";
            }
        }
        return result;
    };
```

Das letzte Beispiel hatte den Nachteil, dass der Entwickler den Namen des Formulars hart codieren muss. Um dies zu umgehen, zeigt Beispiel 4-31, wie sämtliche Form-Controller im aktuellen Scope gefunden werden können. Da die Konstruktorfunktion der Form-Controller nicht außerhalb von AngularJS verfügbar ist, muss an dieser Stelle geprüft werden, ob Objekte existieren, die »so aussehen wie Form-Controller«.

Beispiel 4-31: Programmatisch sämtliche Form-Controller in Erfahrung bringen

```
    $scope.vm.flug.getForms = function () {

        var result = [];
        for (var key in $scope) {
            var scopeMember = $scope[key];

            if (scopeMember != null
                    && typeof scopeMember.$valid == "boolean"
                    && typeof scopeMember.$dirty == "boolean"
                    && typeof scopeMember.$error == "object") {
                result.unshift(scopeMember.$name);
            }
        }
        return result;
    }
```

Benutzerdefinierte Validierungslogiken

Um benutzerdefinierte Validierungslogiken bereitzustellen, ist der Entwickler angehalten, eigene Direktiven zu entwickeln. Während der Abschnitt »Benutzerdefinierte Direktiven« ausführlich auf die Erstellung benutzerdefinierter Direktiven eingeht, fokussiert sich dieser Abschnitt auf die Erstellung benutzerdefinierter Direktiven für das Validieren von Eingaben.

Hierzu verwendet der Entwickler, wie Beispiel 4-32 demonstriert, die Funktion `directive`, welche AngularJS auf Modul-Ebene anbietet. Der erste Parameter ist der Name der Direktive; der zweite Parameter erwartet eine Funktion, die ein Objekt zurückgibt, das die Direktive beschreibt. Die Eigenschaft `require` dieses Objekts legt fest, dass die hier definierte Direktive nur für Felder, die auch die Direktive `ng-model` nutzen, verwendet werden darf. Dies macht sinn, denn `ng-model` kümmert sich schließlich um die Datenbindung. Die Eigenschaft `link` verweist auf eine Funktion, welche die Direktive für ein

Feld aktiviert. Über ihre Parameter erhält diese Funktion von AngularJS den aktuellen Scope (`scope`), ein jQuery-Objekt, das das auf die Direktive angewandte HTML-Element repräsentiert (`elm`), ein Objekt, welches die Attribute dieses Elements beinhaltet (`attrs`), sowie ein Objekt, welches das Steuerelement im Form-Controller repräsentiert.

Die Direktive in Beispiel 4-32 prüft unter Verwendung von `Globalize` (*https://github.com/jquery/globalize*), ob die Eingabe des Benutzers einer Zahl entspricht, die entsprechend der aktuell gewählten Kultur formatiert wurde. Um das gewünschte Format zu ermitteln, greift `link` auf den Wert des Attributs `gnumber` zu. Dabei ist zu beachten, dass dieses Attribut denselben Namen wie die Direktive hat und somit jenen Wert aufweist, der der Direktive zugewiesen wurde.

Anschließend richtet `link` für das Eingabefeld einen Parser sowie einen Formatter ein. Bei beiden Konzepten handelt es sich um Funktionen. Ein Parser ist eine Funktion, die die Eingabe des Benutzers parst. Dabei kann sie dem Steuerelement mitteilen, ob die Eingabe korrekt validiert werden konnte, indem sie dessen Funktion `$setValidity` verwendet. Der erste Parameter von `$setValidity` ist der Name der Validierungslogik; der zweite Parameter zeigt an, ob ein Validierungsfehler aufgetreten ist. Jenen Wert, den der Parser zurückgibt, schreibt AngularJS zurück ins Model. Auf diese Weise kann die `link`-Funktion veranlassen, dass Benutzereingaben nicht nur in Form von Strings, sondern in Form anderer Typen in das Model geschrieben werden. Besonders nützlich ist dies bei der Behandlung von Datumswerten, wo das Zurückschreiben eins `Date`-Objekts wünschenswert ist.

Der Formatter kümmert sich um die Formatierung des Werts, bevor dieser an das Eingabefeld gebunden wird. Er formatiert den Wert unter Verwendung von Globalize und des an die Direktive übergebenen Formats.

Bei Betrachtung des vorliegenden Quellcodes fällt auf, dass ein Steuerelement mehrere Parser und Formatter besitzen kann, zumal es sich bei den Eigenschaften `$parsers` und `$formatters` um Arrays handelt. Dies ist notwendig, zumal pro Steuerelement mehrere Direktiven mit jeweils einer eigenen Validierungslogik verwendet werden können.

Beispiel 4-32: Benutzerdefinierte Validierunslogiken bereitstellen

```
var app = angular.module("Flug", []);
[…]
app.directive('gnumber', function () {
return {
    require: 'ngModel',
    link: function (scope, elm, attrs, ctrl) {

        var format = attrs.gnumber;

        ctrl.$parsers.unshift(function (viewValue) {

            var number = Globalize.parseFloat(viewValue);
            if (!isNaN(number)) {
                ctrl.$setValidity('gnumber', true);
                return number;
            }
            else {
                ctrl.$setValidity('gnumber', false);
                return undefined;
            }
```

```
        });

        ctrl.$formatters.unshift(function (value) {
            var formatted = Globalize.format(value, format);
            return formatted;
        });

    }
};
});
```

Beispiel 4-33 demonstriert, wie die im letzten Beispiel beschriebene Direktive eingesetzt werden kann. Dazu erhält das zu validierende Input-Element ein Attribut, welches dem Namen der Direktive entspricht. Der Wert dieses Attributs wird auf das Format gesetzt, das Globalize verwenden soll. Um herauszufinden, ob die Validierung erfolgreich war, greift das Beispiel auf die Eigenschaft `form.flugNummer.$error.gnumber` zu, wobei `gnumber` jener Bezeichner ist, den die Direktive an `$setValidity` übergibt.

Beispiel 4-33: Benutzerdefinierte Validierungslogik anwenden

```
<div>Preis</div>
<div><input ng-model="vm.preis" gnumber="N2" /></div>
<div ng-show="form.flugNummer.$dirty && form.flugNummer.$error.gnumber">
   Muss eine Zahl sein!
</div>
```

In Beispiel 4-34 und Beispiel 4-35 findet man zwei weitere Direktiven, die analog zum gerade betrachteten Beispiel prüfen, ob der Benutzer einen korrekten Integer-Wert bzw. ein korrektes Datum erfasst hat. Dazu nutzen diese Direktiven ebenfalls `Globalize`, um herauszufinden, ob die Eingaben den Formatierungen der jeweiligen Kultur entsprechen. Letztere Direktive filtert aus der Eingabe des Benutzers das Zeichen mit dem Code 8206. Dieses Zeichen wird von IE 11 eingefügt, um eine Information über die Leserichtung der Eingabe (von links nach rechts oder umgekehrt) zu hinterlegen und führt zu einem Problem bei der Validierung mit Globalize.

Beispiel 4-34: Benutzerdefinierte Validierungslogik zum Validieren von Integer-Werten

```
app.directive('integer', function () {
return {
    require: 'ngModel',
    link: function (scope, elm, attrs, ctrl) {

        var format = attrs.integer;

        ctrl.$parsers.unshift(function (viewValue) {

            var integer = Globalize.parseInt(viewValue);
            if (!isNaN(integer)) {
                ctrl.$setValidity('integer', true);
                return integer;
            }
            else {
                ctrl.$setValidity('integer', false);
                return undefined;
            }

        });
```

```
        ctrl.$formatters.unshift(function (value) {
            debugger;
            var formatted = Globalize.format(value, format);
            return formatted;
        });

    }
};
});
```

Beispiel 4-35: Benutzerdefinierte Validierungslogik zum Validieren von Datumswerten

```
app.directive('gdate', function () {

return {
    require: 'ngModel',
    link: function (scope, elm, attrs, ctrl) {

        var fmt = attrs.gdate;

        ctrl.$parsers.unshift(function (viewValue) {

            var d = Globalize.parseDate(viewValue);

            if (d) {
                ctrl.$setValidity('gdate', true);
                return d;
            }
            else {
                ctrl.$setValidity('gdate', false);
                return undefined;
            }

        });

        ctrl.$formatters.unshift(function (value) {
            var formatted = Globalize.format(value, { date: fmt });
            var parts = formatted.split(' ');

            var date = parts[0];
            var result = "";
            for (var i = 0; i < date.length; i++) {

                if (date.charCodeAt(i) != "8206") {
                    result += date.charAt(i);
                }

            }

            return result;
        });

    }
};
```

Steuerelementbibliotheken für AngularJS

Mittlerweile stehen für AngularJS sowohl kommerzielle als auch freie Steuerelementbibliotheken zur Verfügung. Als Beispiel für eine freie Bibliothek sei an dieser Stelle AngularUI (*http://angular-ui.github.io/*) genannt. Diese Bibliothek beinhaltet unter anderem neben einer umfangreichen Grid-Implementierung auch eine Variante von Bootstrap, welche auf AngularJS portiert wurde.

Ein Beispiel für eine kommerzielle Bibliothek stellt Kendo UI von Telerik dar (*http://www.telerik.com/kendo-ui*). Diese Bibliothek bietet ebenfalls eine ausgereifte Grid-Implementierung, jedoch auch andere Steuerelemente, wie TreeViews, Slider, Kalender oder Menüs.

Logische Seiten und Routing

JavaScript-getriebene Webanwendungen realisieren in der Regel über eine physische Seite mehrere logische Seiten, sodass der Benutzer ohne merklichen Ladevorgang zwischen einzelnen Seiten wechseln kann. Zur Realisierung dieses Verfahrens existieren in AngularJS zwei Möglichkeiten: Zum einen kann der Entwickler verschiedene Teile einer Seite bei Bedarf ein- und ausblenden und zum anderen bietet AngularJS ein Modul namens `ngRoute`, welches diese Aufgabe abstrahiert und auch Deep-Linking unterstützt. Deep-Linking bedeutet, dass der Benutzer über die *Zurück*-Schaltfläche nicht nur zu physischen Seiten, sondern auch zu logischen Seiten zurückkehren sowie für logische Seiten Bookmarks erstellen kann.

Dieser Abschnitt geht auf diese beiden Bordmittel ein und beschreibt zusätzlich das freie externe Modul UI-Router, welches als Alternative zu `ngRoute` einige weitere Möglichkeiten anbietet.

Logische Seiten mit ng-switch

Die einfachste Möglichkeit, mehrere logische Seiten innerhalb einer physischen zu implementieren, besteht im Ein- und Ausblenden verschiedener Seitenteile. Dies kann zum Beispiel über die in den letzten Abschnitten mehrmals eingesetzte Direktive `ng-show` erfolgen. Etwas strukturierter kann sich der Entwickler dieser Aufgabe hingegen mit der Direktive `ng-switch` nähern. Wie ihr Name schon vermuten lässt, funktioniert diese Direktive ähnlich wie die aus C# bekannte Kontrollstruktur `switch`: Der Entwickler gibt eine Variable an und abhängig von dieser Variablen wird einer von mehreren Bereichen angezeigt.Beispiel 4-36 und Beispiel 4-37 demonstrieren den Einsatz von `ng-switch`. Beispiel 4-36 zeigt ein View-Model, welches eine Eigenschaft `state` aufweist. Diese Eigenschaft zeigt an, welche logische Seite angezeigt werden soll und kann über `SetState` verändert werden.

Beispiel 4-36: Für den Einsatz von ng-switch erweitertes View-Model

```
function FlugBuchenVM(scope, http, q, stateParams, state) {
    this.state = "Passagier";

    […]

    this.SetState = function (state) {
        this.state = state;
    };

    […]
}
```

Die dazu passende View findet sich in Beispiel 4-37. Im oberen Bereich bietet sie einige Links zum Navigieren zwischen den logischen Seiten an. Dazu modifiziert sie über `SetState` die zuvor betrachtete Eigenschaft `state`. Im Bereich darunter befinden sich die logischen Seiten. Die Direktive `ng-switch` kümmert sich um das Ein- aus Ausblenden der einzelnen logischen Seiten unter Berücksichtigung des Werts der Eigenschaft `state`. Für jeden Wert, den `state` annehmen kann, existiert ein `div`-Element mit einer `ng-switch-when`-Direktive, welche den jeweiligen Wert aufweist. AngularJS blendet, wenn `state` den jeweiligen Wert einer `ng-switch-when`-Direktive aufweist, das damit assoziierte Element ein. Die anderen `div`-Elemente werden ausgeblendet. Hat `state` einen Wert, der durch keine `ng-switch-when`-Direktive repräsentiert wird, blendet AngularJS das Element mit der `ng-switch-default`-Direktive ein.

Beispiel 4-37: Logische Seiten mit ng-switch realisieren

```
<div ng-app="flug">
    <div ng-controller="FlugBuchenCtrl">

        <div class="step">
            <a href="#" ng-click="vm.SetState('Passagier')">Passagier</a> |
            <a href="#" ng-click="vm.SetState('Flug')">Flug</a> |
            <a href="#" ng-click="vm.SetState('Buchen')">Buchen</a> |
            <a href="#" ng-click="vm.SetState('MeineBuchungen')">Meine Buchungen</a>
        </div>

        <div ng-switch="vm.state">

            <div class="step" ng-switch-when="Passagier">
                [...]
            </div>

            <div class="step" ng-switch-when="Flug">
                [...]
            </div>

            <div class="step" ng-switch-when="Buchen">
                [...]
            </div>

            <div class="step" ng-switch-default>
                [...]
            </div>

        </div>

    </div>
</div>
```

Da die hier betrachtete Lösung einzelne Seitenteile ein- und ausblendet, kann der Benutzer nicht unter Verwendung der *Zurück*-Schaltfläche zu einer zuvor besuchten logischen Seiten zurückkehren oder ein Bookmark für eine logische Seite erstellen. Dieser Unschönheit nimmt sich das Modul `ngRoute` an.

Routing mit dem Modul ngRoute

Das Modul `ngRoute`, welches sich im Lieferumfang von AngularJS befindet, erlaubt das Definieren sogenannter Routen. Über Routen legt der Entwickler fest, welche logische Seite innerhalb einer physischen Seite anzuzeigen ist. Auf den ersten Blick sieht eine Route wie ein Teil eines URLs aus. Allerdings wird die Route im Hash-Fragment jenes URLs platziert, der zur übergeordneten physischen Seite führt. Nachfolgend finden sich drei Beispiele für Routen einer physischen Seite:

- *http://.../Flug/RoutingSample#/passagiere*
- *http://.../Flug/RoutingSample#/passagiere/info*
- *http://.../Flug/RoutingSample#/passagiere/1*

Abhängig von der Route im Hash-Fragment blendet AngularJS eine von mehreren logischen Seiten innerhalb der adressierten physischen Seiten ein. Da die Route ein Teil des URLs ist, erzeugt der Browser dafür einen Eintrag in der Browser-History. Das führt dazu, dass der Benutzer mit der Zurück-Schaltfläche zu bereits besuchten logischen Seiten zurückkehren kann und ein Bookmark für eine logische Seite erstellen kann. Wie das letzte der zuvor betrachteten Beispiele andeutet, kann eine Route auch Parameter aufweisen. Im betrachteten Fall gibt dieser Parameter an, welcher Passagier anzuzeigen ist.

Um auf einen über das Routing bereitgestellten Parameter zugreifen zu können, lässt der Entwickler in den Controller den Service `$routeParams` injizieren. Dabei handelt es sich um ein einfaches JavaScript-Objekt, welches pro Routing-Parameter eine Eigenschaft aufweist. Der Controller `DetailCtrl` im betrachteten Listing veranschaulicht dies anhand des Routing-Parameters `id`, der in der Route `/passagiere/:id` angegeben wurde.

Zur Nutzung des Moduls `ngRoute` bindet der Entwickler das Skript `angular-route.js` ein und referenziert es bei der Erstellung des Moduls für die Anwendung. Beispiel 4-38 demonstriert dies, sowie, wie der Entwickler unter Verwendung der Funktion `config`, welche AngularJS auf Modulebene bereitstellt, Routen konfigurieren kann. Die Funktion `config` nimmt eine Funktion mit dem nötigen Konfigurationscode entgegen. Diese bekommt, wie unter AngularJS üblich, Abhängigkeiten injiziert. Um die diskutierten Probleme mit der Minification zu verhindern, akzeptiert `config` auch, wie ebenfalls unter AngularJS üblich, ein Array mit den Namen der Abhängigkeiten und einer Funktion. Auf diese Weise lässt das Beispiel in Beispiel 4-38 den von `ngRoute` bereitgestellten `$routeProvider` injizieren.

Mit der Funktion `when` von `$routeProvider` richtet das hier betrachtete Beispiel die oben gezeigten Routen ein. Der erste an `when` übergebene Parameter ist die Route. Routen können Platzhalter beinhalten, welche mit einem Doppelpunkt eingeleitet werden. Dies ist zum Beispiel bei der Route `/passagiere/:id` der Fall, die die ID des anzuzeigenden Passagiers beinhaltet.

Der zweite an `when` übergebene Parameter ist ein Objekt, welches Informationen zur Route beinhaltet. Über die Eigenschaft `templateUrl` verweist es auf eine Datei, welche die logische Seite enthält, die mit der definierten Route zu assoziieren ist. Eine Alternative zu dieser Eigenschaft bietet die Eigenschaft `template`, welche direkt das Markup der jeweiligen logischen Seiten entgegennimmt. Die Eigenschaft `controller` beinhaltet hingegen den Namen des Controllers, der für diese logische Seite verantwortlich ist. Diese werden im Beispiel weiter unten definiert. Eine Verwendung derselben Controller-Instanz für mehrere Routen ist leider nicht möglich. Allerdings kann diese Einschränkung durch das im nächsten Abschnitt beschriebene externe Modul UI-Router umgangen werden. Mit `otherwise` legt das hier betrachtete Beispiel fest, dass AngularJS zur Route `/passagiere` wechseln soll, sofern der Benutzer eine nicht existierende Route ansteuert.

Beispiel 4-38: Definition von Routen

```
<script src="~/Scripts/jquery-1.10.2.js"></script>
<script src="~/Scripts/angular.js"></script>
<script src="~/Scripts/angular-route.js"></script>

<script>

    var app = angular.module("routing", ['ngRoute']);

    app.config(['$routeProvider', function ($routeProvider) {
        $routeProvider.
            when('/passagiere, {
                templateUrl: '/partials/routing.list.html',
                controller: 'ListCtrl'
            })
            .when('/passagiere/info', {
                templateUrl: '/partials/routing.info.html',
                controller: 'InfoCtrl'
            })
            .when('/passagiere/:id', {
                templateUrl: '/partials/routing.details.html',
                controller: 'DetailCtrl'
            })
            .otherwise({
                redirectTo: '/passagiere'
            });
    }]);

    app.controller("ListCtrl", ["$scope", function ($scope) {
        $scope.passagiere = [
            { id: 1, vorname: "Max", nachname: "Muster", passagierStatus: "B" },
            { id: 2, vorname: "Susi", nachname: "Sorglos", passagierStatus: "A" },
            { id: 3, vorname: "Reiner", nachname: "Zufall", passagierStatus: "B" },
            { id: 4, vorname: "Hans-Peter", nachname: "Grahsl", passagierStatus: "A" }
        ];
    }]);

    app.controller("DetailCtrl", ["$scope", "$routeParams", function ($scope, $routeParams) {
        $scope.id = $routeParams.id;

    }]);

    app.controller("InfoCtrl", ["$scope", function ($scope) {
        $scope.info = "Hallo Welt!";
    }]);
</script>
```

Beispiel 4-39 zeigt die View, welche von der physischen Seite ausgeliefert wird. Um zwischen zwei der drei definierten Routen zu wechseln, weist dieses Beispiel zwei Links auf. Mit der Direktive `ng-view` markiert es ein `div`-Element, in dem die jeweils adressierte logische Seite anzuzeigen ist.

Beispiel 4-39: Seite mit Platzhalter ng-view

```
<body ng-app="routing">
    <h1>Routing-Sample</h1>
```

```
        <a href="#/routing">List</a>
        <a href="#/routing/info">Info</a>

        <div ng-view></div>

    </body>
```

Ein Beispiel für die Datei, welche eine logische Seite zur Anzeige mehrerer Passagiere beinhaltet, findet sich in Beispiel 4-40. Pro Passagier weist diese logische Seite einen Link mit der Beschriftung `Details` auf. Dieser Link führt zur Route jener logischen Seite, die die Details des jeweiligen Passagiers präsentiert.

Beispiel 4-40: View zur Anzeige eines Passagiers

```
    <table>
        <tr>
            <th>Id</th>
            <th>Vorname</th>
            <th>Nachname</th>
            <th>PassagierStatus</th>
            <th></th>
        </tr>
        <tr ng-repeat="p in passagiere">
            <td>{{p.id}}</td>
            <td>{{p.vorname}}</td>
            <td>{{p.name}}</td>
            <td>{{p.passagierStatus}}</td>
            <td><a href="#/routing/{{p.id}}">Details</a></td>
        </tr>

    </table>
```

Während `ngRoute` einige Möglichkeiten zur Realisierung logischer Seiten bietet, ist es dennoch mit einigen Einschränkungen versehen. Beispielsweise können Controller nicht View-übergreifend eingesetzt werden. Views können auch nicht geschachtelt werden, sodass innerhalb einer View eine weitere View zur Anzeige kommt. Darüber hinaus kann die physische Seite auch nur einen einzigen Platzhalter aufweisen, in den logische Seiten eingebettet werden. Wenn auch derzeit keine Bordmittel zur Lösung dieser Einschränkungen existieren, schaffen Module von Drittanbietern hier Abhilfe. Beim Modul UI-Router, auf das der nächste Abschnitt eingeht, handelt es sich um ein solches Modul.

Routing mit dem externen Modul UI-Router

Das externe Modul UI-Router (*https://github.com/angular-ui/ui-router*) gibt dem Entwickler die Möglichkeit, Views zu schachteln, mehrere Views als logische Seiten in eine physische Seite einzubinden sowie ein und denselben Controller für mehrere logische Seiten zu verwenden. Es ist Teil des Projekts AngularUI, welches Module zur Gestaltung ansprechender Benutzeroberflächen mit AngularJS bereitstellt.

Verschachtelte Views

Um UI-Router zu nutzen, bindet der Entwickler die JavaScript-Datei `angular-ui-router.min.js` ein und importiert das Modul `ui.router`. Um UI-Router zu konfigurieren, lässt der Entwickler die Services `$stateProvider` und `$urlRouterProvider` in die Funktion `config` injizieren. Beispiel 4-41 definiert mit

der Funktion `otherwise` des Services `$urlRouterProvider`, welche Route zu verwenden ist, wenn der Benutzer eine Route aufruft, die nicht definiert wurde. Anschließend definiert es mit der Funktion `state` des Services `$stateProvider` verschiedene Zustände, die die Anwendung einnehmen kann. Jeder Zustand wird mit einer logischen Seite assoziiert, die sich aus einer oder mehreren Views zusammensetzen kann.

Der erste an `state` übergebene Parameter ist der Name des jeweiligen Zustands. Der zweite Parameter stellt ein Objekt dar, welches den Zustand beschreibt. Dessen Eigenschaft `url` spiegelt die Route des Zustands wieder, `templateUrl` verweist auf die Datei, welche die View des Zustands beinhaltet und `controller` beinhaltet den Namen des für den jeweiligen Zustand verantwortlichen Controllers. Alternativ zur einer `templateUrl` kann auch mit der Eigenschaft `template` direkt das Markup der gewünschten View als String angegeben werden.

Wird ein Zustand mit der booleschen Eigenschaft `abstract` als abstrakter Zustand ausgewiesen, geht UI-Router davon aus, dass die damit assoziierte View Platzhalter für weitere Views beinhaltet. Ein Beispiel dafür ist der Zustand `flugbuchung` in Beispiel 4-41. Welche View in diesen Platzhalter eingefügt wird, bestimmt der Benutzer, indem er den URL eines untergeordneten Zustands ansteuert. Untergeordnete Zustände (auch verschachtelte Zustände) erkennt man daran, dass Ihr Name dem des übergeordneten Zustands gefolgt von einem Punkt und einem eigenen Namen entspricht. Demnach handelt es sich im betrachteten Beispiel bei `flugbuchung.passagier`, `flugbuchung.flug`, `flugbuchung.buchen` und `flugbuchung.meineBuchungen` um Zustände, die dem Zustand `flugbuchung` untergeordnet sind. Der URL, der zu diesen untergeordneten Zuständen führt, entspricht der Kombination aus dem URL des übergeordneten Zustands und dem des eigenen Zustands. Beispielsweise würde im betrachteten Fall die Verwendung des URLs `/flugbuchung/passagier` innerhalb des Hash-Fragments zur Aktivierung des Zustands `flugbuchung.passagier` führen.

Bei Betrachtung der Konfiguration in Beispiel 4-41 fällt darüber hinaus auf, dass nur die abstrakte View `flugbuchung` einen Controller zugewiesen bekommen hat. Dieser Controller kommt somit auch für sämtliche untergeordnete Zustände zum Einsatz.

Beispiel 4-41: Zustände mit UI-Router definieren

```
var app = angular.module("Flug", ['ui.router']);

app.config(function ($stateProvider, $urlRouterProvider) {
    $urlRouterProvider.otherwise("/flugbuchung/passagier");

    $stateProvider.state('flugbuchung', {
        abstract: true,
        url: '/flugbuchung',
        templateUrl: '/partials/flugbuchung.html',
        controller: "FlugBuchenCtrl"
    }).state('flugbuchung.passagier', {
        url: '/passagier',
        templateUrl: '/partials/flugbuchung.passagier.html'
    }).state('flugbuchung.flug', {
        url: '/flug',
        templateUrl: '/partials/flugbuchung.flug.html'
    }).state('flugbuchung.buchen', {
        url: '/buchen',
        templateUrl: '/partials/flugbuchung.buchen.html'
    }).state('flugbuchung.meineBuchungen', {
        url: '/meineBuchungen',
```

```
        templateUrl: '/partials/flugbuchung.meineBuchungen.html'
    });
});
```

Beispiel 4-42 zeigt das Markup der hier beschriebenen AngularJS-Anwendung. Im Element, das mit der Direktive `ui-view` versehen wurde, platziert UI-Router die View des jeweiligen Zustands.

Beispiel 4-42: Platzhalter für logische Seite

```
<div>
    <div ui-view></div>
</div>
```

Die View des Zustands `flugbuchung` findet sich in Beispiel 4-43. Er weist zur Navigation zwischen den untergeordneten Zuständen Links auf, wobei jeder Link über die Direktive `ui-sref` auf den Namen jenes Zustands verweist, zu dem er führen soll. Die Stelle, an der die View des jeweils gewählten untergeordneten Zustands eingesetzt wird, ist mit der Direktive `ui-view` markiert.

Beispiel 4-43: Logische Seite

```
<div>

    <div class="step">
        <a ui-sref="flugbuchung.passagier">Passagier</a> |
        <a ui-sref="flugbuchung.flug">Flug</a> |
        <a ui-sref="flugbuchung.buchen">Buchen</a> |
        <a ui-sref="flugbuchung.meineBuchungen">Meine Buchungen</a>
    </div>

    <div ng-show="vm.message"
         style="color:red; padding-left: 20px; font-weight: bold">
        {{ vm.message }}
    </div>

    <div ui-view></div>

</div>
```

Mehrere Views pro Vorlage verwenden

Neben dem Verschachteln von Zuständen erlaubt UI-Router auch die Verwendung mehrerer Views pro Zustand. In diesem Fall erhält jede View einen Namen. Beispiel 4-44 veranschaulicht dies, indem es für jeden konkreten Zustand eine View `main` sowie eine weitere View `info` definiert.

Beispiel 4-44: Konfiguration von UI-Router zur Verwendung mehrerer Views pro Zustand

```
app.config(function ($stateProvider, $urlRouterProvider) {
    $urlRouterProvider.otherwise("/flugbuchung/passagier");

    $stateProvider.state('flugbuchung', {
        abstract: true,
        url: '/flugbuchung',
        templateUrl: '/partials/flugbuchung.html',
        controller: "FlugBuchenCtrl"
    }).state('flugbuchung.passagier', {
        url: '/passagier',
        views: {
            "main": { templateUrl: '/partials/flugbuchung.passagier.html' },
```

```
            "info": { templateUrl: '/partials/flugbuchung.passagier.info.html' }
        }
    }).state('flugbuchung.flug', {
        url: '/flug',
        views: {
            "main": { templateUrl: '/partials/flugbuchung.flug.html' },
            "info": { templateUrl: '/partials/flugbuchung.flug.info.html' }
        }
    }).state('flugbuchung.buchen', {
        url: '/buchen',
        views: {
            "main": { templateUrl: '/partials/flugbuchung.buchen.html' },
            "info": { templateUrl: '/partials/flugbuchung.empty.html' }
        }
    }).state('flugbuchung.meineBuchungen', {
        url: '/meineBuchungen',
        views: {
            "main": { templateUrl: '/partials/flugbuchung.meineBuchungen.html' },
            "info": { templateUrl: '/partials/flugbuchung.empty.html' }
        }
    });
});
```

Um innerhalb einer View auf mehrere definierte benannte Views zuzugreifen, übergibt der Entwickler den Namen der jeweiligen View an die Direktive `ui-view`, die Bereiche der Seite als Platzhalter ausweist (siehe Beispiel 4-45).

Beispiel 4-45: View, welche auf mehrere weitere Views verweist

```
<div>
    <div class="step">
        <a ui-sref="flugbuchung.passagier">Passagier</a> |
        <a ui-sref="flugbuchung.flug">Flug</a> |
        <a ui-sref="flugbuchung.buchen">Buchen</a> |
        <a ui-sref="flugbuchung.meineBuchungen">Meine Buchungen</a>
    </div>

    <div ng-show="vm.message" style="color:red; padding-left: 20px; font-weight: bold">
        {{ vm.message }}
    </div>

    <div ui-view="main"></div>

    <div class="step">
        <input ng-click="vm.goToNextState()" value="Weiter" type="button" />
    </div>
    <hr />

    <div ui-view="info"></div>

</div>
```

Parameter übergeben

Genau wie das Modul `ngRouter` (siehe den Abschnitt »Routing mit dem Modul ngRoute«) unterstützt auch UI-Router den Einsatz von Routing-Parameter. Dazu nimmt der Entwickler Platzhalter, die mit einem Doppelpunkt eingeleitet werden, in den URL der Route auf. Um auf die Werte dieser Parameter

zugreifen zu können, injiziert der Entwickler – analog zum Einsatz von ngRouter – den Service $stateParams in den zuständigen Controller (siehe Beispiel 4-46).

Beispiel 4-46: Verwendung von Routing-Parametern beim Einsatz von UI-Router

```
$stateProvider.state("passagierDetails", {
        url: "/passagierDetails/:pNummer",
        templateUrl: '/partials/passagierDetails.html',
        controller: function ($scope, $stateParams) {
            $scope.detail = {};
            $scope.detail.pNummer = $stateParams.pNummer;
            $scope.detail.name = "Max Muster";
        }
    });
```

Um Routing-Parameter an einen Zustand zu übergeben, sieht der Entwickler einen Link mit dem URL des jeweiligen Zustands vor. Dabei ersetzt er die Platzhalter im URL durch die gewünschten Werte. Alternativ dazu kann er auch die Direktive ui-sref wie folgt verwenden:

```
ui-sref="flugbuchung.meineBuchungen({pNummer: 4711})"
```

Auf Zustände programmatisch zugreifen

Zum programmatischen Zugriff auf Zustände kann der Entwickler den Service $state heranziehen. Über dessen Eigenschaft current.name kann er den Namen des aktuellen Zustands in Erfahrung bringen, über dessen Funktion go kann er den Zustand wechseln. Die Beispiele in Beispiel 4-47 und Beispiel 4-48 veranschaulichen dies.

Beispiel 4-47: Den Service $state in einen Controller injizieren

```
app.controller("FlugBuchenCtrl", ["$scope", "$http", "$q", "$stateParams", "$state", function ($scope,
$http, $q, $stateParams, $state) {

        $scope.vm = new FlugBuchenVM($scope, $http, $q, $stateParams, $state);
}]);
```

Beispiel 4-48: Den Service $state verwenden

```
function FlugBuchenVM(scope, http, q, stateParams, state) {
    […]
    this.stateParams = stateParams;
    this.state = state;

    this.goToNextState = function () {

         var current = this.state.current.name;

         switch(current) {
            case "flugbuchung.passagier":
                this.state.go("flugbuchung.flug");
            break;

            case "flugbuchung.flug":
                this.state.go("flugbuchung.buchen");
            break;
        }
    }
}
```

AngularJS-Anwendungen testen

Da bei modernen Webanwendungen immer mehr Logik mittels JavaScript implementiert wird, liegt der Wunsch nahe, diese Codebereiche automatisiert zu testen. Für das Testen von AngularJS bietet sich das Framework Jasmine an. Es handelt sich dabei um ein Framework zur Implementierung von Unit-Tests, welches die Ideen des Behavior-Driven-Designs (BDD) unterstützt. Das bedeutet, dass mit jedem Testfall ein Verhaltensmerkmal der Anwendung beschrieben werden kann. Allerdings ist Jasmine nicht von AngularJS abhängig, sondern kann auch zum Testen anderer JavaScript-basierter Codestrecken herangezogen werden.

Dieser Abschnitt geht zunächst auf Jasmine ein und zeigt anschließend, wie es zum Testen von verschiedenen AngularJS-Konstrukten, wie Controllern, Filtern oder Direktiven, eingesetzt werden kann. Darüber hinaus zeigt dieser Abschnitt, wie die Möglichkeiten bezüglich Dependency-Injection in AngularJS zur Steigerung der Testbarkeit beitragen.

Anatomie eines Jasmine-Tests

Testfälle befinden sich beim Einsatz von Jasmine innerhalb von Blöcken, die als Test-Suiten bezeichnet durch einen Aufruf der Methode `describe` definiert werden. Dazu nimmt `describe`, wie Beispiel 4-49 demonstriert, zwei Parameter entgegen. Beim ersten Parameter handelt es sich um eine Beschreibung des erwarteten Verhaltens oder eines Teils davon. Der zweite Parameter repräsentiert eine Funktion, die den Test beinhaltet. Innerhalb dieser Funktion können weitere Aufrufe von `describe` stattfinden, um das beschriebene Verhalten weiter zu konkretisieren. Auf diese Weise lässt sich eine Hierarchie mit Anforderungen aufbauen. Am Ende dieser Hierarchie befinden sich die einzelnen Testfälle, welche der Entwickler mit der Funktion `it` beschreibt. Genauso wie `describe` erwartet `it` eine Beschreibung zur Konkretisierung des erwarteten Verhaltens sowie eine Funktion. Letztere beinhaltet den Test. Abbildung 4-2 zeigt die Hierarchie, die auf diese Weise Beispiel 4-49 aufbaut und die Jasmine bei der Ausführung ausgibt. Es handelt sich dabei, den Ideen von BDD folgend, um eine ausführbare Spezifikation. Die Tatsache, dass die Blätter dieser Hierarchie mit grüner Schrift präsentiert werden, deutet darauf hin, dass die damit assoziierten Tests erfolgreich ausgeführt wurden.

```
Object under test
  when this
    and when that
      should do this
      should do that
```

Abbildung 4-2: Ausgabe eines Jasmine-Tests

Innerhalb der mit `it` definierten Testfälle führt der Entwickler Prüfungen durch, um herauszufinden, ob der Testfall die gewünschten Ergebnisse erzielt hat. Der betrachtete Testfall prüft zum Beispiel mit

```
expect(objectUnderTest.ok).toBe(true)
```

ob die Eigenschaft `objectUnderTest.ok` den Wert `true` aufweist. Funktionen wie `toBe` werden im Jasmine-Jargon als Matcher bezeichnet. Die Bedeutung eines Matchers kann unter Verwendung der Eigenschaft `not` umgekehrt werden:

```
expect(objectUnderTest.ok).not.toBe(true)
```

Neben `toBe` bietet Jasmine einige weitere Matcher an, deren jeweiliger Name Programm ist. Dazu zählen `toBeDefined`, `toBeNull`, `toBeNaN`, `toBeFalsy`, `toBeTruthy`, `toContain`, `toBeLessThan`, `toBeGreaterThan`

und `toThrow`. Daneben kann der Entwickler mit `toMatch` prüfen, ob ein Wert einem übergebenen regulären Ausdruck entspricht. Dem Umstand, dass Fließkommazahlen kleine Ungenauigkeiten aufweisen können, trägt der Matcher `toBeCloseTo` Rechnung. Er nimmt zwei Werte entgegen: Jenen Wert, der erwartet wird, sowie ein erlaubtes Delta. Informationen dazu findet man unter *https://github.com/pivotal/jasmine/wiki/Matchers*.

Mit `beforeEach` und `afterEach` registriert der Entwickler jeweils eine weitere Funktionen, die vor oder nach sämtlichen Testfällen der aktuellen sowie der untergeordneten Ebenen der beschriebenen Hierarchie ausgeführt werden.

Beispiel 4-49: Ein sehr einfacher Jasmine-Test

```
describe("Object under test", function () {

    beforeEach(function () {
    });

    afterEach(function () {
    });

    describe("when this", function () {

        beforeEach(function () {
        });

        afterEach(function () {
        });

        describe("and when that", function () {

            it ("should do this", function () {
                var objectUnderTest = { ok: true };
                expect(objectUnderTest.ok).toBe(true);
            });

            it("should do that", function () {
                var objectUnderTest = { notOk: false };
                expect(objectUnderTest.notOk).toBe(false);

            });
        });
    });
});
```

Der Jasmine-Test-Runner

Um Jasmine-Tests, wie jenen im letzten Abschnitt, auszuführen, benötigt der Entwickler einen Test-Runner. Der Test-Runner, der im Lieferumfang von Jasmine zu finden ist, besteht lediglich aus einer HTML-Seite, welche die CSS- und JavaScript-Dateien von Jasmine referenziert und ein wenig JavaScript-Code zur Ausführung der Tests beinhaltet (vgl. Beispiel 4-50). In diese Datei inkludiert der Entwickler die JavaScript-Dateien, welche seine Tests beinhalten sowie sämtliche Dateien, welche die (direkten und indirekten) Abhängigkeiten der Tests aufweisen.

Beispiel 4-50: Jasmine-Test-Runner

```
@{
    Layout = null;
}
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Jasmine Spec Runner</title>
    <link rel="shortcut icon" type="image/png" href="/Content/jasmine/jasmine_favicon.png">
    <link rel="stylesheet" type="text/css" href="/Content/jasmine/jasmine.css">
    <script type="text/javascript" src="/Scripts/jasmine/jasmine.js"></script>
    <script type="text/javascript" src="/Scripts/jasmine/jasmine-html.js"></script>
    <!-- include source files here... -->
    <script type="text/javascript" src="/Scripts/jasmine-samples/SpecHelper.js"></script>
    <script type="text/javascript" src="/Scripts/jasmine-samples/PlayerSpec.js"></script>
    <!-- include spec files here... -->
    <script type="text/javascript" src="/Scripts/jasmine-samples/Player.js"></script>
    <script type="text/javascript" src="/Scripts/jasmine-samples/Song.js"></script>

    ///
    <script src="~/Scripts/angular.js"></script>
    ///
    <script src="~/Scripts/moment.js"></script>
    <script src="~/Scripts/q.js"></script>

    <script src="~/Scripts/AngularApp.js"></script>
    <script src="~/Scripts/tests/AngularAppTests.js"></script>

    <script src="~/Scripts/angular-ui.js"></script>

    <script type="text/javascript">
        (function () {
            var jasmineEnv = jasmine.getEnv();
            jasmineEnv.updateInterval = 1000;

            var htmlReporter = new jasmine.HtmlReporter();

            jasmineEnv.addReporter(htmlReporter);

            jasmineEnv.specFilter = function (spec) {
                return htmlReporter.specFilter(spec);
            };

            var currentWindowOnload = window.onload;

            window.onload = function () {
                if (currentWindowOnload) {
                    currentWindowOnload();
                }
                execJasmine();
            };

            function execJasmine() {
                jasmineEnv.execute();
            }
```

```
        })();
    </script>
</head>
<body>
</body>
</html>
```

Das freie Projekt `Chutzpah` (*http://chutzpah.codeplex.com*) bietet einen Test-Runner, welcher Jasmine-Tests direkt in Visual Studio zur Ausführung bringt und auch beim Build durch Team Foundation Server genutzt werden kann. Damit Chutzpah herausfinden kann, welche zusätzlichen JavaScript-Dateien mit benötigten Abhängigkeiten für den Test zu laden sind, ist der Entwickler angehalten, diese über Kommentare zu referenzieren, die mit drei Schrägstrichen beginnen und XML-Elemente beinhalten. Diese Elemente weisen den Namen `reference` auf und verweisen über ein Attribut `path` auf die gewünschte Datei. Da auch TypeScript solche Kommentare verwendet, erlaubt `Chutzpah` auch die Verwendung des Namens `chutzpah_reference`.

```
/// <reference path="../angular.js" />
/// <reference path="../AngularApp.js" />
/// <chutzpah_reference path="../moment.js" />
/// <chutzpah_reference path="../jasmine/jasmine.js" />
/// <chutzpah_reference path="../q.js" />
describe("FlugBuchenVM", function () { […] }
```

Da der HTML-basierte Jasmine-Test-Runner mehr Informationen über fehlgeschlagene Testfälle liefert, hat es sich bewährt, Tests mit diesem Test-Runner zu testen, bevor `Chutzpah` verwendet wird.

Ein View-Model testen

Am einfachsten gestaltet sich das Testen von View-Models, gerade auch dann, wenn lediglich zu prüfen ist, ob die einzelnen Methoden die gewünschten Werte zurückgeben bzw. zu den gewünschten Zustandsänderungen führen. Etwas komplizierter wird es, wenn die View-Models, wie jene in diesem Kapitel, Services erhalten, die AngularJS dem Controller injiziert hat. Denn für den Testfall muss der Entwickler sich darum kümmern, an diese Abhängigkeiten oder an sogenannte Mocks (Attrappen), die diese Abhängigkeiten simulieren, zu kommen. Beispiel 4-51 bezieht zu diesem Zweck den Injector-Service von AngularJS. Dabei handelt es sich um jenen Service, der sich um das Auflösen von Abhängigkeiten kümmert. Um an diesen Service zu kommen, ruft der betrachtete Testfall die Funktion `angular.injector` auf und übergibt in einem Array die Namen der Module mit den gewünschten Abhängigkeiten. Der Modulname `ng` bezieht sich hier auf jenes Modul, das von AngularJS standardmäßig eingerichtet wird und das die Standard-Services bereitstellt. Die Funktion `get` des Injectors nimmt den Namen eines Services entgegen und gibt diesen zurück, sofern er existiert.

Auf diese Weise gelangt der hier betrachtete Test in der Methode `beforeEach` an die Services `$http`, `$q` und `$rootScope`. Mit der Funktion `$new` des Services `$rootScope` erstellt er einen neuen Scope. Anschließend instanziiert der Test eine `FlugBuchenVM`-Instanz, welche er in weiterer Folge testet.

Das hier betrachtete Beispiel spendiert dem View-Model auch einen `baseUrl`. Diesen Wert stellt es dem URL der einzelnen aufgerufenen HTTP-Services, wie `/api/flug`, voran, um einen absoluten URL hierfür zu bilden. Dies ist notwendig, wenn die Testfälle nicht innerhalb der zu testenden Webanwendung, sondern in einem lokalen Test-Runner ausgeführt werden. Ein Beispiel dafür ist der Test-Runner

Chutzpah, welcher in Visual Studio über den Erweiterungsmanager installiert werden kann und Jasmine-Tests direkt in Visual Studio zur Ausführung bringt. Innerhalb von `it` wird anschließend geprüft, ob der über `selectFlug` ausgewählte Flug auch über die Eigenschaft `selectedFlug` zur Verfügung steht.

Beispiel 4-51: Testen eines View-Models mit Jasmine

```
describe("FlugBuchenVM", function () {

    var vm;

    beforeEach(function () {

        var injector = angular.injector(['ng']);

        var http = injector.get("$http");
        var q = injector.get("$q");
        var rootScope = injector.get("$rootScope");
        var scope = rootScope.$new();

        vm = new FlugBuchenVM(scope, http, q);
        vm.baseUrl = "http://localhost:59978";

    });

    afterEach(function () {
    });

    it("should select 2nd flight", function () {

        var flug1 = new Flug.FlugVM({});
        var flug2 = new Flug.FlugVM({});
        var flug3 = new Flug.FlugVM({});

        vm.fluege.push(flug1);
        vm.fluege.push(flug2);
        vm.fluege.push(flug3);

        vm.selectFlug(1);

        expect(vm.selectedFlug).toBe(flug2);
    });

});
```

Asynchrone Tests

Für das Testen asynchroner Methoden mit Jasmine bildet der Entwickler eine Kette von Funktionen, die gemeinsam den gewünschten Testfall repräsentieren. Dazu nutzt er, wie Beispiel 4-52 veranschaulicht, die von Jasmine angebotenen Funktionen `runs` und `waitsFor`. Erstere fügt eine beliebige Funktion zur Kette hinzu; Letztere fügt eine Funktion hinzu, welche dazu führt, dass Jasmine den Testfall anhält, bis eine bestimmte Bedingung eingetreten ist. Jasmine ruft diese Funktion immer und immer wieder auf. Liefert sie `true`, fährt Jasmine mit der nächsten Funktion in der Kette fort. Ansonsten pausiert Jasmine und bringt kurz darauf die mit `waitsFor` definierte Funktion erneut zur Ausführung. Damit dies in keiner Endlosschleife mündet, kann der Entwickler `waitsFor`, wie im betrachteten Bei-

spiel gezeigt, ein Timeout in Sekunden sowie eine Information, die im Falle eines Timeouts auszugeben ist, übergeben.

Beispiel 4-52: Asynchrone Tests mit AngularJS

```
it("should load Passagiere", function () {

    var finished = false;
    var error = false;

    runs(function () {
        vm.passagierNrFilter = "1";
        vm.load().then(function () {
            finished = true;
        }).catch(function (ex) {
            finished = true;
            error = true;
        });
    });

    waitsFor(function () {
        return finished;
    }, "Zu lange auf Load gewartet!", 20000);

    runs(function () {
        expect(error).toBe(false);
        expect(vm.passagiere.length).toBe(1);
    });
});
```

Controller testen

Der Entwickler testet Controller mit Jasmine ähnlich wie View-Models. Allerdings muss er hierzu wissen, wie er an eine Instanz eines definierten Controllers gelangt. Dazu besorgt er sich vom Injector den Service `$controller`. Dabei handelt es sich lediglich um eine Funktion, welche den Controller zur Ausführung bringt. Sie nimmt den Namen des Controllers sowie ein Objekt mit den Parametern entgegen, die an den Controller zu übergeben sind. Die Eigenschaften dieses Objekts haben den Namen der einzelnen Parameter zu entsprechen. Auf diese Weise kann der Entwickler den Controller unter Verwendung eines eigenen Scopes zur Ausführung bringen. Danach kann er prüfen, ob der Scope die gewünschten Werte beinhaltet sowie eventuelle Funktionen bzw. View-Models dieses Scope-Objekts testen.

Beispiel 4-53: Controller mit AngularJS testen

```
describe("FlugBuchenVM", function () {

    var scope;

    beforeEach(function () {

        var injector = angular.injector(['ng', 'Flug', 'ui.router']);

        var http = injector.get("$http");
        var q = injector.get("$q");
        var rootScope = injector.get("$rootScope");
```

```
        scope = rootScope.$new();
        var controller = injector.get("$controller");

        controller("FlugBuchenCtrl", { $scope: scope });

        scope.vm.baseUrl = "http://localhost:59978";


    });

    afterEach(function () {
    });

    it("should load Passagiere", function () {

        var finished = false;
        var error = false;

        var vm = scope.vm;

        runs(function () {
            vm.passagierNrFilter = "1";
            vm.load().then(function () {
                finished = true;
            }).catch(function (ex) {
                finished = true;
                error = true;
            });
        });

        waitsFor(function () {
            return finished;
        }, "Zu lange auf Load gewartet!", 6000);

        runs(function () {
            expect(error).toBe(false);
            expect(vm.passagiere.length).toBe(1);
        });
    });
});
```

HTTP-Zugriffe für Tests mit angular-mock.js simulieren

Mit dem im Lieferumfang von AngularJS enthaltenen Skript *angular-mocks.js* bietet AngularJS dem Entwickler Unterstützung für das Erstellen von Tests mit Jasmine an. Dieses Skript stellt ein paar Hilfsfunktionen, welche den Einsatz des Injectors vereinfachen, zur Verfügung. Daneben bewirkt das Einbinden dieses Skripts auch, dass zentrale AngularJS-Services durch Mocks (Attrappen) ausgetauscht werden. Zu diesen Mocks zählt ein Mock für den Service `$httpBackend`, welcher der Service `$http` für den Zugriff auf HTTP-Services heranzieht. Um ein isoliertes Testen zu ermöglichen, greift dieser Mock nicht auf HTTP-Services zu, sondern retourniert stattdessen Objekte, die der Testfall im Vorfeld definiert hat.

Beispiel 4-54 demonstriert dies. Mit der von *angular-mock.js* bereitgestellten Funktion `module` definiert es, dass das Modul `Flug` getestet werden soll. Da diese Funktion innerhalb von `beforeEach` verwendet

wird, wird dieses Modul vor jedem Testfall initiiert. Das Modul ng muss der Entwickler beim Einsatz von angular-mock.js nicht explizit referenzieren. An einen zweiten Aufruf von beforeEach übergibt das betrachtete Beispiel einen Aufruf von inject. Dabei handelt es sich um eine weitere Funktion, die angular-mock.js bereitstellt. Inject erwartet eine Funktion, an die sie delegiert. Dabei injiziert inject die benötigten Abhängigkeiten in deren Parameter. Die Auflösung der Abhängigkeiten erfolgt, wie unter AngularJS üblich, über die Namen dieser Parameter. Auf diese Weise erhält Beispiel 4-54 die Services $rootScope, $controller, $http, $q und $httpBackend, wobei es sich bei Letzterem um einen Mock handelt, zumal angular-mock.js eingebunden wurde.

Der mit it definierte Testfall bereitet diesen Mock auf seinen Einsatz vor. Dazu verwendet er die Funktion expect, um anzuzeigen, dass davon auszugehen ist, dass ein an den hinterlegten URL gerichteter Servicezugriff via GET erfolgt. Hierbei ist auch von Expectation die Rede, zumal damit eine Erwartungshaltung ausgedrückt wird.

Die Funktion respond legt das Objekt fest, welches in diesem Fall vom Mock zurückzugeben ist. Dies erlaubt es, Logik, die sich auf diesen Servicezugriff stützt, ohne tatsächliches Durchführen dieses Servicezugriffs isoliert zu testen.

Damit sich der betrachtete Testfall wie gewünscht verhält, ist der Entwickler angehalten, nach dem Serviceaufruf die vom httpBackend-Mock angebotene Funktion flush aufzurufen. Erst dadurch wird die Simulation des asynchronen Servicezugriffs abgeschlossen und der definierte Wert retourniert.

Innerhalb von afterEach prüft das hier gezeigte Beispiel mit verifyNoOutstandingExpectation, ob sämtliche für den Mock definierte Erwartungen (engl. expectations) eingetreten sind. Mit verifyNoOutstandingRequest prüft es darüber hinaus, ob sämtliche Serviceaufrufe abgeschlossen wurden. Dies ist nicht der Fall, wenn der zuvor erwähnte Aufruf von flush nicht stattfindet.

Beispiel 4-54: HTTP-Zugriffe simulieren

```
<script src="~/Scripts/angular.js"></script>
<script src="~/Scripts/angular-mocks.js"></script>

[…]

describe("FlugBuchenVM", function () {

    var scope;
    var httpBackend

    beforeEach(module("Flug"));

    beforeEach(inject(function ($rootScope, $controller, $http, $q, $httpBackend) {
        scope = $rootScope.$new();

        var ctrl = $controller('FlugBuchenCtrl', {
            $scope: scope,
            $http: $http,
            $q: $q
        });

        httpBackend = $httpBackend;

        scope.vm.baseUrl = "http://localhost:59978";
    }));
```

```
    afterEach(function () {

        httpBackend.verifyNoOutstandingExpectation();
        httpBackend.verifyNoOutstandingRequest();

    });

    it("should load Passagiere", function () {

        var finished = false;
        var error = false;

        var vm = scope.vm;

        runs(function () {

            httpBackend
                .expect('GET', 'http://localhost:59978/api/passagier?pNummer=1')
                .respond([{ Id: 1, Vorname: "Max", Name: "Muster", PassagierStatus: "A" }]);

            vm.passagierNrFilter = "1";

            vm.load().then(function () {
                finished = true;
            }).catch(function (ex) {
                finished = true;
                error = true;
            });

            httpBackend.flush();
        });

        waitsFor(function () {
            return finished;
        }, "Zu lange auf Load gewartet!", 6000);

        runs(function () {
            expect(error).toBe(false);
            expect(vm.passagiere.length).toBe(1);
        });
    });
});
```

Angular-Services für Tests simulieren

Anstatt einen Mock für den Service `$httpBackend` zu verwenden, kann der Entwickler auch jenen Service simulieren, der über `$http` und somit indirekt über `$httpBackend` auf einen HTTP-Service zugreift. Dazu muss er lediglich ein Objekt, das nach außen hin so aussieht wie der eigentliche Service, an den Aufruf von `$controller` übergeben. Beispiel 4-55 demonstriert dies. Es definiert innerhalb von `beforeEach` einen Mock für den `passagierService` mit einer Funktion `load`. Diese Funktion gibt, wie das Gegenstück im Tatsächlichen, `passagierService`, ein Promise zurück. Dabei ist es wichtig, zu erkennen, dass das `deferrable`, mit dem das Promise erstellt wird, außerhalb von `beforeEach` deklariert wird und somit auch später im Zuge der Testfälle zur Verfügung steht. Dies ist notwendig, damit die Test-

fälle das Promise mit `resolve` oder `reject` über den Ausgang des asynchronen Aufrufs informieren können. Damit das Promise hierauf reagiert, ist der Entwickler angehalten, die Funktion `scope.$root.$digest()` zur Ausführung zu bringen. Diese veranlasst AngularJS u.a. dazu, ausstehende Ereignisse anzustoßen.

Damit der Aufruf der Methode `load` vom Testfall überwacht werden kann, richtet dieser hierfür mit der von Jasmine bereitgestellten Funktion `spyOn` einen sogenannten Spy ein. Dabei handelt es sich um einen Wrapper für Funktionen eines Mocks, der aufzeichnet, dass die ummantelte Funktion aufgerufen wird. Spys können an die ummantelte Funktion delegieren oder stattdessen einen eigenen Rückgabewert zurückliefern.

Durch den zusätzlichen Aufruf von `andCallThrough` bewirkt der Testfall, dass die tatsächliche Logik von `load` zur Ausführung kommt. Stattdessen könnte der Testfall auch unter Verwendung der Methode `andReturn` angeben, dass bei jedem Aufruf lediglich ein bestimmter Wert zurückzugeben ist:

```
andReturn(deferredPassagiere.promise)
```

Mit Matchern wie `toHaveBeenCalled` prüft der Testfall am Ende, ob `load` aufgerufen wurde. Die einzelnen hier gezeigten Varianten erlauben es auch, herauszufinden, wie häufig `load` zur Ausführung gebracht wurde und welche Parameter dabei zum Einsatz kamen.

Beispiel 4-55: Services simulieren

```
describe("FlugBuchenVM", function () {

    var scope;
    var httpBackend
    var passagierService;
    var deferredPassagiere;

    beforeEach(module("Flug"));

    beforeEach(inject(function ($rootScope, $controller, $http, $q, $timeout) {
        scope = $rootScope.$new();

        passagierService = {
            load: function (params) {
                deferredPassagiere = $q.defer();
                return deferredPassagiere.promise;
            }
        }

        var ctrl = $controller('FlugBuchenCtrl', {
            $scope: scope,
            $http: $http,
            $q: $q,
            flugService: null,
            passagierService: passagierService
        });

        scope.vm.baseUrl = "http://localhost:59978";
    }));

    afterEach(function () {
    });
```

```
    it("should load Passagiere", function () {

        var vm = scope.vm;

        spyOn(passagierService, "load").andCallThrough();

        vm.passagierNrFilter = "1";

        vm.load();

        deferredPassagiere.resolve([{
            Id: 1,
            Vorname: "Max",
            Name: "Muster",
            PassagierStatus: "A"
        }]);

        scope.$root.$digest(); // scope-lifecycle simulieren

        expect(vm.passagiere.length).toBe(1);

        var expectedArgument = { pNummer: '1' };
        expect(passagierService.load).toHaveBeenCalled();
        expect(passagierService.load.calls.length).toEqual(1);
        expect(passagierService.load).toHaveBeenCalledWith(expectedArgument);
        expect(passagierService.load.mostRecentCall.args[0]).toEqual(expectedArgument);
        expect(passagierService.load.mostRecentCall.args[0]).toEqual(expectedArgument);
        expect(passagierService.load.calls[0].args[0]).toEqual(expectedArgument);
    });

});
```

Der Spy-Mechanismus von Jasmine erlaubt auch das Erzeugen von Mocks, die lediglich aus Spy-Methoden (Spies) bestehen. Dazu nutzt der Entwickler die Funktion `createSpyObj`, an die er einen intern für das Objekt zu verwendenden Namen, der bei Fehlermeldungen zum Einsatz kommt, sowie ein Array mit den einzurichtenden Funktionen übergibt. Mit Funktionen, wie `andReturn`, kann der Entwickler für ein solches Spy-Objekt festlegen, welche Werte die einzelnen Methoden zurückzugeben haben:

```
passagierService = jasmine.createSpyObj("passagierServiceSpy", ["load"]);
passagierService.load.andReturn(deferredPassagiere.promise);
```

Filter testen

Um innerhalb eines Tests eine Referenz auf einen Filter zu erhalten, lässt der Entwickler den Service `$filter` injizieren. Hierbei handelt es sich um eine Funktion, die den Namen eines Filters entgegennimmt und jene Funktion, die den damit assoziierten Filter repräsentiert, zurückliefert. Beispiel 4-56 demonstriert dies.

Beispiel 4-56: Filter testen

```
describe("GlobalizeFilter", function () {

    var scope;
    var httpBackend
    var passagierService;
```

```
    var deferredPassagiere;

    var filter;

    beforeEach(module("Flug"));

    beforeEach(inject(function ($rootScope, $controller, $http, $q, $timeout, $filter) {
        scope = $rootScope.$new();

        filter = $filter;

        deferredPassagiere = $q.defer();
        passagierService = jasmine.createSpyObj("passagierServiceSpy", ["load"]);


        var ctrl = $controller('FlugBuchenCtrl', {
            $scope: scope,
            $http: $http,
            $q: $q,
            flugService: null,
            passagierService: passagierService
        });

        scope.vm.baseUrl = "http://localhost:59978";
    }));

    afterEach(function () {

    });

    it("should format 12.5 for curlture DE as 12,50", function () {

        Globalize.culture("de");

        var globalize = filter("globalize");
        var result = globalize(12.5, "n2");
        expect(result).toEqual("12,50");

    });

});
```

Direktiven testen

Zum Testen von Direktiven muss der Entwickler ein HTML-Fragment simulieren, welches sich auf die Direktive stützt. Dies geschieht über den Service `$compile`. Auch hierbei handelt es sich lediglich um eine Funktion. Diese Funktion nimmt einen String mit HTML entgegen und kompiliert ihn auf dieselbe Weise, wie AngularJS ganze Views kompiliert. Dabei aktiviert `$compile` die hinterlegten Direktiven. Das Ergebnis von `$compile` ist ein jQuery-Objekt, welches das jeweilige HTML-Fragment beschreibt. Damit tatsächlich sämtliche von jQuery bekannten Funktionen zur Verfügung stehen, ist das Skript für jQuery einzubinden.

Damit sich AngularJS um die Datenbindung kümmert und anstehende Ereignisbehandlungsroutinen ausführt, muss der Testfall die Funktion `$digest` vom `$rootScope` anstoßen. Innerhalb von »richtigen«

Views geschieht dies ohne Zutun des Entwicklers. Anschließend kann der Entwickler zum Testen über jQuery mit der Direktive interagieren. Da hierbei keine Browserereignisse ausgelöst werden, muss er sich manuell darum kümmern. Hierzu kann die jQuery-Funktion `trigger` herangezogen werden. Beispiel 4-57 verwendet diese Funktion zum Simulieren des Browserereignisses `input`, welches anzeigt, dass der Inhalt eines Textfelds geändert wurde. Anschließend führt das betrachtete Beispiel aus den genannten Gründen erneut `$digest` aus und prüft, ob sich die Direktive wie erwartet verhalten hat.

Beispiel 4-57: Direktiven mit Jasmine testen

```
describe("IntegerDirective", function () {

    var rootScope;
    var compile;

    beforeEach(module("Flug"));

    beforeEach(inject(function ($rootScope, $compile) {
        rootScope = $rootScope;
        compile = $compile;
    }));

    afterEach(function () {

    });

    it("should mark the value 'Keine Zahl!' as invalid value.", function () {

        var element = compile("<form name='form' id='form'><input number ng-model='aNumber'
id='aNumber' name='aNumber'></form>")(rootScope);

        rootScope.$digest();

        element.find("#aNumber").val("Keine Zahl!");
        element.find("#aNumber").trigger("input");

        rootScope.$digest();

        var invalid = rootScope.form.aNumber.$invalid;

        expect(invalid).toBe(true);

    });

});
```

Benutzerdefinierte Direktiven

Eine der Stärken von AngularJS ist die Möglichkeit, eigene Direktiven zu entwickeln. Damit kann der Entwickler das Vokabular von HTML erweitern und wiederverwendbare Komponenten schaffen. Während der Abschnitt »Benutzerdefinierte Validierungslogiken« dieses Thema bereits für die Erstellung einer benutzerdefinierten Validierungslogik aufgegriffen hat, geht dieser Abschnitt auf die Möglichkeiten benutzerdefinierter Direktiven jenseits der Validierung von Eingaben ein. Dazu zeigt er anhand eines durchgängigen Beispiels die vielen Möglichkeiten, die AngularJS in diesem Bereich bie-

tet. Hierbei handelt es sich, wie Abbildung 4-3 veranschaulicht, um eine Alternative zu Kontrollkästchen (engl. Checkbox).

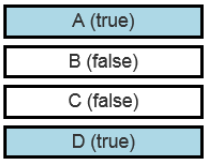

Abbildung 4-3: Benutzerdefinierte Direktive mit einer alternativen Darstellungsform für Kontrollkästchen

Eine erste (zu) einfache Direktive

Zur Erstellung einer eigenen Direktive verwendet der Entwickler die Funktion `directive`, wie Beispiel 4-58 veranschaulicht. Der erste Parameter repräsentiert den Namen der Direktive, der zweite verweist hingegen auf eine Funktion, welche die Direktive in Form eines Objekts zurückgibt. Für Namen wird in der Regel die Kamelschreibweise (engl. Camel Case) verwendet, wobei diese für Markup in eine Schreibweise umgewandelt wird, die auf Kleinbuchstaben basiert und zur Trennung der einzelnen Worte Bindestriche verwendet. Aus `coolCheckbox` wird somit im Markup `cool-checkbox`.

Die Eigenschaft `templateUrl` verweist auf eine Datei, welche jene HTML-Elemente enthält, mit denen die Direktive darzustellen ist. Wie später noch gezeigt wird, kann die Direktive diese Elemente unter Verwendung von jQuery auch abändern. Alternativ zu `templateUrl` kann der Entwickler auch die Eigenschaft `template` heranziehen, welche auf einen String mit den jeweiligen HTML-Elementen verweist.

Setzt der Entwickler die Eigenschaft `transclude` auf `true`, gibt er an, dass der Inhalt jenes Elements, auf das die Direktive angewendet wird, an einer bestimmten Stelle im Template auftachen soll. Diese Stelle zeichnet das Template mit der Direktive `ng-transclude` aus (siehe Beispiel 4-59). Somit würde beim betrachteten Beispiel der Einsatz von

```
<div cool-checkbox>Hallo Welt</div>
```

bewirken, dass der Text `Hallo Welt` innerhalb des Elements angezeigt würde, welches mit `ng-transclude` markiert ist.

Beim Einsatz von Direktiven in Form von Kommentaren ist zu beachten, dass diese keine Möglichkeit zur Transklusion bieten, zumal Kommentare im Gegensatz zu Elementen keinen Inhalt aufweisen, der transkludiert werden könnte.

Mit der Eigenschaft `restrict` gibt der Entwickler an, mit welchen HTML-Mitteln die Direktive im Markup repräsentiert werden darf. `restrict` erwartet einen String, welcher aus den Zeichen E (Element), C (Class, im Sinne des `class`-Attributs), M (Darstellung über Kommentare) und A (Attribute) besteht. Abhänig davon, welche dieser Zeichen im String vorkommen, darf die Direktive mit mehr oder weniger diesen Möglichkeiten, für die Beispiel 4-60 ein Beispiel bietet, dargestellt werden.

Setzt der Entwickler `replace` auf `true`, gibt er an, dass das Element, welches die Direktive repräsentiert, durch das Template der Direktive zu ersetzen ist. Setzt der Entwickler diese Eigenschaft hingegen auf `false`, stellt AnugularJS das Template der Direktive innerhalb des Elements dar, das die Direktive repräsentiert. In Fällen, in denen die Direktive über HTML-Kommentare dargestellt werden soll, ist es

notwendig, dass der Entwickler `replace` auf `true` setzt, zumal HTML-Kommentare keine untergeordneten HTML-Elemente besitzen.

Die Eigenschaft `controller` verweist auf eine Funktion, die als Controller der Direktive fungiert. Wie bei Controllern üblich, erhält auch dieser Controller die notwendigen Abhängigkeiten über die Dependency-Injection. Im betrachteten Fall erhält der Controller den aktuellen Scope, ein jQuery-Objekt, das das Template repräsentiert (`$element`), und ein Objekt, welches sämtliche Attribute (`$attrs`) jenes Elements, über das die Direktive im Markup dargestellt wird, beinhaltet. Stellt der Entwickler die Direktive beispielsweise über das Element

```
<div cool-checkbox class="B">Some Text</div>
```

dar, weist `$attrs` eine Eigenschaft `class` mit dem Wert `B` auf. Zusätzlich besäße `$attrs` auch eine Eigenschaft `coolCheckbox` mit dem Wert `undefined`. Zur Bildung des Namens solcher Eigenschaften wandelt AngularJS die schreibweise `cool-checkbox` in die unter JavaScript übliche Schreibweise `coolCheckbox` um. Beim Einsatz von

```
<div cool-checkbox="this-and-that" class="B">Some Text</div>
```

würde die Eigenschaft `coolCheckbox` hingegen den Wert `this-and-that` aufweisen.

Der in Beispiel 4-58 gezeigte Controller hinterlegt im Scope ein Model, welches über `isChecked` angibt, ob die dargestellte Checkbox aktiviert ist, und über `toggleState` einen Zustandswechsel der Checkbox bewirkt. Auf dieses Model bezieht sich das Template in Beispiel 4-59: Es wählt unter Berücksichtigung von `isChecked` eine CSS-Klasse aus und verweist über den `Click`-Handler auf `toggleState`.

Beispiel 4-58: Einfache Direktive

```
var app = angular.module("directivesSample", []);

app.directive("coolCheckbox", function () {

    return {
        //template: "<div class='coolCheckBoxOff'>Checkbox</div>"
        templateUrl: "/templates/coolCheckBoxTemplateWithNg.html",
        transclude: true,
        restrict: "ECMA",
        replace: true,
        controller: function ($scope, $element, $attrs) {
            // $scope.title = "Haaacked"; // wenn nichts anderes angegeben wird, bekommt die Direktive
den selben Scope, wie ihr Parent

            $scope.model = {
                isChecked: false,
                toggleState: function () {
                    this.isChecked = !this.isChecked;
                }
            }
        }
    };

});

app.controller("sampleCtrl", function ($scope) {
    $scope.title = "Directive-Sample";
});
```

Beispiel 4-59: Template für einfache Direktive

```
<div ng-class="(model.isChecked) ? 'coolCheckBoxOn' : 'coolCheckBoxOff' " ng-click="model.
toggleState()"><span ng-transclude></span> ({{model.isChecked}})</div>
```

Beispiel 4-60: Anwenden einer einfachen Direktive

```
<div ng-app="directivesSample">
    <div ng-controller="sampleCtrl">
        <h2>{{ title }}</h2>
        <div cool-checkbox>Some Text</div>
        <div class="cool-checkbox">Some Text</div>
        <cool-checkbox>Some Text</cool-checkbox>
        <!-- directive: cool-checkbox -->
    </div>
</div>
```

Da in diesem ersten Beispiel keine weiteren Angaben zum Scope gemacht wurden, entspricht der an den Controller übergebene Scope dem aktuellen Scope in der AngularJS-Anwendung. Somit teilt sich die Direktive den Scope mit der Anwendung und das kann zu Konflikten führen. Beispielsweise würden nun mehrere Instanzen der betrachteten Direktive auf dasselbe Model zugreifen. Das hat zur Folge, dass sich eine Änderung der Eigenschaft `isChecked` auf sämtliche Instanzen von `coolCheckbox` auswirkt.

Daneben ist `coolCheckbox` in der Lage, Eigenschaften, die an anderen Stellen in der Seite genutzt werden, zu überschreiben. Das betrachtete Beispiel deutet dies mit einem Kommentar an: Setzt `coolCheckbox` die Scope-Eigenschaft `title` auf einen anderen Wert, aktualisiert AngularJS den Titel der Seite, zumal dieser an die Eigenschaft `title` gebunden ist (Beispiel 4-60). Es liegt auf der Hand, dass solche Nebeneffekte verhindert werden müssen. Aus diesem Grund bietet AngularJS die Möglichkeit, einer Direktive einen eigenen Scope zu spendieren. Informationen darüber finden sich in den nachfolgenden Abschnitten »Eigener Scope für Direktive « und »Isolierte Scopes«.

Eigener Scope für Direktive

Die im letzten Abschnitt beschriebene Direktive verwendet den aktuellen Scope. Dies kann zu Nebeneffekten führen, wenn die Direktive Variablen dieses Scopes verwendet, die auch an anderen Stellen in der Anwendung zum Einsatz kommen. Um dieses Problem zu lösen, kann der Entwickler angeben, dass für die Direktive ein neuer Scope erzeugt werden soll. Wie im Abschnitt »Scopes« beschrieben, steht dieser Scope über die Prototypenkette in Beziehung mit dem aktuellen, bereits existierenden Scope, auch als Parent-Scope bezeichnet. Somit stehen auch sämtliche Elemente dieses Scopes zur Verfügung. Zuweisungen zu Eigenschaften des Scopes führen jedoch immer zu einer Änderung des eigenen Scopes.

Damit eine Direktive ihren eigenen Scope erhält, setzt der Entwickler die Eigenschaft `scope` auf `true`. Beispiel 4-61 veranschaulicht das.

Beispiel 4-61: Direktive mit eigenem Scope

```
var app = angular.module("directivesSample", []);

app.directive("coolCheckbox", function () {

    return {
        templateUrl: "/templates/coolCheckBoxTemplate.html",
        transclude: true,
        scope: true, //new Scope
```

```
            replace: true,
            controller: function ($scope) {
                $scope.model = {
                    isChecked: false,
                    toggleState: function () {
                        this.isChecked = !this.isChecked;
                    }
                }
            }

        };

    });
```

Einen eigenen Scope zu haben, verhindert zwar Konflikte mit anderen Direktiven, bringt jedoch ein weiteres Problem mit sich: Es besteht keine (einfache und schöne) Möglichkeit, eine Variable des eigenen Scopes an eine Variable des äußeren Scopes zu binden. Dies ist im betrachteten Fall jedoch vonnöten, um den Wert der Checkbox nach außen zu kommunizieren, sodass Funktionen des Controller-Scopes darauf Zugriff haben. Dieses Problem kann der Entwickler mit isolierten Scopes lösen, die im nächsten Abschnitt beschrieben werden.

Isolierte Scopes

Im Gegensatz zu neuen Scopes, die für eine Direktive erzeugt werden, stehen isolierte Scopes nicht über die Prototypenkette in Beziehung mit dem bereits existierenden Parent-Scope. Stattdessen kann der Entwickler angeben, dass bestimmte Eigenschaften des bestehenden Scopes an Eigenschaften des isolierten Scopes zu binden sind. Um einen isolierten Scope für eine Direktive zu erstellen, weist der Entwickler der Direktiveneigenschaft `scope` ein Objekt zu. Jede Eigenschaft dieses Objekts beschreibt eine Eigenschaft des isolierten Scopes, welche an einen existierenden Wert zu binden ist.

Der Wert dieser Eigenschaften ist ein String, der aus zwei Informationen besteht: einem Präfix, das die Art der Bindung beschreibt, und dem Namen eines HTML-Attributs, mit dem die zu bindende Eigenschaft des äußeren Scopes zu benennen ist. Da es sich hier um eine (Zwei-Wege-)Datenbindung handelt, werden Änderungen, die die Direktive an diesen Eigenschaften im isolierten Scope vornimmt, auch an die gebundenen Eigenschaften des äußeren Scopes übertragen.

Beispiel 4-62 veranschaulicht dies. Es legt fest, dass die Direktive einen isolierten Scope erhalten soll, indem es der Eigenschaft `scope` ein Objekt zuweist. Da dieses Objekt die Eigenschaften `isChecked` und `stateChanged` aufweist, erhält der isolierte Scope diese beiden Eigenschaften, welche mit Werten des äußeren Scopes verknüpft werden.

An `isChecked` weist das betrachtete Beispiel den Wert `=isChecked` zu. Das Präfix `=` bedeutet an dieser Stelle, dass `isChecked` mit dem Wert einer Variablen (und nicht mit einer Funktion) des äußeren Scopes zu verknüpfen ist. Der restliche String mit dem Inhalt `isChecked` gibt an, dass die Variable des äußeren Scopes über ein HTML-Attribut `is-checked` anzugeben ist. An dieser Stelle wird abermals die in JavaScript übliche Kamelschreibweise (`isChecked`) auf die für HTML-Attribute übliche Schreibweise (`is-checked`) abgebildet.

Die Eigenschaft `stateChanged` erhält den Wert `&onchange`. Das Präfix `&` gibt an, dass `stateChanged` mit einer Funktion des äußeren Scopes zu verknüpfen ist. Der restliche String mit dem Inhalt `onchange` gibt auch an dieser Stelle den Namen des zu verwendenden HTML-Attributs an.

Neben dem Präfix = und dem Präfix & kann der Entwickler auch auf das Präfix @ zurückgreifen. Damit gibt er an, dass das benannte Attribut den gewünschten Wert direkt beinhaltet. Es findet hier somit keine Verknüpfung mit einer Variablen des äußeren Scopes statt. Dies kann jedoch umgangen werden, indem der Entwickler im genannten Attribut einen Datenbindungsausdruck nach dem Schema

```
{{ variablenName }}
```

hinterlegt. In diesem Fall wird die beschriebene Eigenschaft des isolierten Scopes trotz Verwendung von @ mit der im Datenbindungsausdruck genannten Variable verknüpft.

Beispiel 4-62: Direktive mit isoliertem Scope

```
var app = angular.module("directivesSample", []);


app.directive("coolCheckbox", function () {

    return {
        //template: "<div class='coolCheckBoxOff'>Checkbox</div>"
        templateUrl: "/templates/coolCheckBoxTemplate2.html",
        transclude: true,
        restrict: "ECA",
        scope: {
            isChecked: "=isChecked",
            stateChanged: "&onchange"
        },
        replace: true,
        controller: function ($scope, $element, $attrs) {
            // $scope.title = "Haaacked"; // wenn nichts anderes angegeben wird, bekommt die Direktive
den selben Scope, wie ihr Parent



            $scope.toggleState = function () {
                $scope.isChecked = !$scope.isChecked;
                $scope.stateChanged();
            }
        }
    };
});
```

Wenn eine Eigenschaft im isolierten Scope denselben Namen, wie das entsprechende HTML-Attribut aufweist, kann der Name des HTML-Attributs auch weggelassen werden. Aus diesem Grund ist die Definition von

```
scope: {
    isChecked: "=isChecked",
    stateChanged: "&onchange"
},
```

im hier gezeigten Beispiel gleichbedeutend mit

```
scope: {
    isChecked: "=",
    stateChanged: "&onchange"
},
```

Um den Einsatz der gerade betrachteten Dirktive und ihres isolierten Scopes zu veranschaulichen, findet sich in Beispiel 4-63 ein einfacher Controller sowie in Beispiel 4-64 eine einfache AngularJS-Anwendung, die sich auf diesen Controller und die diskutierte Direktive stützt. Die Anwendung definiert zwei Instanzen von `coolCheckbox` und bindet an diese über das Attribut `is-checked` die Eigenschaften `isChecked1` und `isChecked2` aus dem Scope des Controllers. Mit `onchange` bindet sie hingegen die Funktion `changed` aus demselben Scope. Auf diese Weise werden nun die Eigenschaften `isChecked1` und `isChecked2` aus dem Scope des Controllers an die Eigenschaft `isChecked` aus dem isolierten Scope der `coolCheckbox`-Direktive gebunden. Änderungen, die die Direktive an der Eigenschaft `isChecked` vornimmt, wirken sich somit auch auf die Eigenschaften `isChecked1` und `isChecked2` des vom Controller verwendeten äußeren Scopes aus.

Beispiel 4-63: Controller zum Testen einer benutzerdefinierten Direktive

```
app.controller("sampleCtrl", function ($scope) {
    $scope.title = "Directive-Sample";
    $scope.isChecked1 = true;
    $scope.isChecked2 = false;

    $scope.changed = function () {
        window.console.log("Changed state");
    }

});
```

Beispiel 4-64: Einsatz einer benutzerdefinierten Direktive mit isoliertem Scope

```
<div ng-app="directivesSample">
    <div ng-controller="sampleCtrl">
        <h2>{{ title }}</h2>
        <div cool-checkbox is-checked="isChecked1" onchange="changed()">Some Text</div>
        <div cool-checkbox is-checked="isChecked2" onchange="changed()">Some Text 2</div>
        <p>
            isChecked1: {{ isChecked1 }}, isChecked2: {{ isChecked2 }}
        </p>
    </div>
</div>
```

Link-Phase

Die bis dato betrachteten Beispiele haben innerhalb der Templates der `coolCheckbox`-Direktive weitere Möglichkeiten von AngularJS verwendet, darunter die Direktiven `ngClick` und `ngClass`. Dies ist legitim und – sofern möglich – auch aufgrund der damit verbundenen Einfachheit zu bevorzugen. In Fällen, wo es zur Implementierung des gewünschten Templates jedoch keine geeigneten vorgefertigten Direktiven gibt, ist der Entwickler angehalten, selbst Hand anzulegen. Hierzu stellt er im Rahmen der Direktive eine Link-Funktion zur Verfügung. Diese hat die Aufgabe, das Template vor dem Einsatz für eine Instanz der Direktive anzupassen. Dabei kann der Entwickler auf jQuery zurückgreifen sowie Ereignisbehandlungsroutinen registrieren.

Beispiel 4-65 und Beispiel 4-66 demonstrieren den Einsatz von `link`. Beispiel 4-65 zeigt dazu eine Alternative zur im letzten Abschnitt betrachteten Template, welche im Gegensatz zu der hier verwendeten Template keinen Gebrauch von `ng-click` und `ng-class` macht. Als Ersatz für diese Direktiven kommt im betrachteten Beispiel die Funktion `link` in Beispiel 4-66 zum Einsatz. Sie erhält den aktuellen Scope,

ein jQuery-Element, dass die Template repräsentiert (`iElement`), und ein Objekt, welches Zugriff auf sämtliche Attribute jenes HTML-Elements, mit dem die Direktive repräsentiert wird, gewährt (`iAttrs`). Das Präfix `i` weist an dieser Stelle darauf hin, dass es hier lediglich um eine einzige Instanz der Direktive geht, oder anders ausgedrückt: Die Funktion `link` wird pro Instanz der Direktive einmal aufgerufen und die einzelnen Aufrufe sind unabhängig voneinander.

Unter Verwendung des Parameters `iAttrs` prüft `link` im betrachteten Beispiel, ob das Element, mit dem die Direktive dargestellt wird, ein Attribut `round` mit dem Wert `true` aufweist. Ist dem so, fügt sie dem Element, welches das Template repräsentiert, eine CSS-Anweisung hinzu, welche die Darstellung abgerundeter Ecken bewirkt.

Würde im hier präsentierten Beispiel die Eigenschaft `replace` nicht den Wert `true`, sondern den Wert `false` aufweisen, würde `iElement` nicht das Template-Element, sondern jenes Element repräsentieren, mit dem die Direktive dargestellt wird. Dies ist der Fall, weil bei `replace=true` das Template das ursprüngliche Element nicht ersetzt, sondern als dessen Kind-Element positioniert wird. In diesem Fall müsste der Entwickler, wie der Kommentar im betrachteten Listing andeutet, auf das erste Kind von `iElement` zugreifen, um jenes Element zu erhalten, welches die Template widerspiegelt.

Danach prüft `link`, ob ein Wert über das Attribut `coolCheckbox` übergeben wurde. Ist dem so, fügt es diesen als zusätzliche Beschriftung in die Template ein. Danach definiert es eine Funktion mit dem Namen `setCssClass`, welche abhängig vom Zustand der Checkbox der Template eine CSS-Klasse zuweist. Damit die Checkbox für ihren Initialzustand die korrekte CSS-Klasse erhält, ruft die Funktion `link` diese Funktion gleich nach ihrer Definition auf. Außerdem definiert sie für das Template einen `Click`-Handler, der den Zustand der Checkbox mit `toggleState` aktualisiert, `setCssClass` aufruft und anschließend die Funktion `$apply` des Scopes zur Ausführung bringt. Letzteres ist notwendig, um AngularJS anzuweisen, die Datenbindung zu aktualisieren.

Beispiel 4-65: Template für benutzerdefinierte Direktive

```
<div><span ng-transclude></span> ({{model.isChecked}})</div>
```

Beispiel 4-66: Einfache Link-Funktion

```
var app = angular.module("directivesSample", []);

app.directive("coolCheckbox", function () {

    return {
        //template: "<div class='coolCheckBoxOff'>Checkbox</div>"
        templateUrl: "/templates/coolCheckBoxTemplate.html",
        transclude: true,
        restrict: "ECMA",
        replace: true,
        controller: function ($scope) {
            $scope.model = {
                isChecked: false,
                toggleState: function () {
                    this.isChecked = !this.isChecked;
                }
            }
        },
        link: function (scope, iElement, iAttrs) {
```

```
            // var checkBoxElement = iElement.children().first();
            // wenn replace:false

            var checkBoxElement = iElement;

            if (iAttrs.round && iAttrs.round.toLowerCase() == "true") {
                checkBoxElement.css("border-radius", "5px");
            }

            if (iAttrs.coolCheckbox) {
                iElement.children().first().before($("<span>" + iAttrs.coolCheckbox + "</span>"));
            }

            var setCssClass = function () {

                if (scope.model.isChecked) {
                    checkBoxElement.removeClass("coolCheckBoxOff");
                    checkBoxElement.addClass("coolCheckBoxOn");
                }
                else {
                    checkBoxElement.removeClass("coolCheckBoxOn");
                    checkBoxElement.addClass("coolCheckBoxOff");
                }

            };

            setCssClass();

            checkBoxElement.on("click", function () {
                scope.model.toggleState();
                setCssClass();
                scope.$apply();
            });
        }
    };

});
```

Beispiel 4-67 demonstriert die Anwendung der gerade beschriebenen Direktive, indem es sieben Instanzen davon erstellt. Diese rendern jeweils eine Checkbox, welche den Text `Some Text 1 bis 7` enthalten. Dabei fällt auf, dass beim Einsatz der Direktive als Attribut, als Klasse, aber auch als Kommentar ein Parameter übergeben werden kann. Im betrachteten Beispiel handelt es sich dabei um einen String mit den Inhalt `Some`. Unabhängig von der Verwendung dieser Darstellungsformen wird der übergebene Wert innerhalb der Direktive als Attribut mit dem Namen der Direktive dargestellt, also `coolCheckbox`. Somit kann der Entwickler ihn in `link` über die Eigenschaft `iAttrs.coolCheckbox` abrufen.

Beim Einsatz der Direktive als Attribut übergibt der Entwickler den gewünschten Wert an das Attribut, mit dem die Direktive repräsentiert wird. Beim Einsatz der Direktive als Klasse stellt er dem Namen der Direktive einen Doppelpunkt sowie den gewünschten Wert nach. Bei Darstellung als HTML-Kommentar platziert er den gewünschten Wert hingegen unmittelbar nach dem Namen der Direktive.

Beispiel 4-67: Mögliche Darstellungsformen einer Direktive im Markup

```
<div ng-app="directivesSample">
    <div ng-controller="sampleCtrl">
        <h2>{{ title }}</h2>
```

```
        <div cool-checkbox>Some Text</div>
        <div cool-checkbox>Some Text 2</div>
        <div cool-checkbox="Some ">Text 3</div>
        <cool-checkbox>Some Text 4</cool-checkbox>
        <div class="cool-checkbox">Some Text 5</div>
        <div class="cool-checkbox: Some">Text 6</div>
        <!-- directive: cool-checkbox Some Text 7 -->
    </div>
</div>
```

Compile-Phase

Neben der Link-Phase existiert aus Gründen der Performance auch eine sogenannte Compile-Phase. Während AngularJS die Link-Phase für jede Instanz einer Direktive einmal ausführt, wird die Compile-Phase nur ein einziges Mal pro Verwendung der Direktive im Markup aus. Der Unterschied zwischen diesen beiden Phasen wird deutlich, wenn man sich den Einsatz einer Direktive innerhalb einer Repeat-Direktive vorstellt:

```
<div ng-repeat="item in items">
    <div cool-checkbox
         is-checked="item.checked"
         onchange="changed()">{{item.label}}</div>
</div>
```

Während die Direktive `cool-checkbox` hier nur einmal im Markup Verwendung findet, erzeugt AngularJS mittels `ng-repeat` mitunter mehrere Instanzen davon – eine pro Eintrag im hier nicht näher betrachteten Array `items`. Transformationen der Templates, die für sämtliche dieser Instanzen gleich sind, können somit zur Steigerung der Performance in die Compile-Phase ausgelagert werden. In diesem Fall arbeitet AngularJS in den Link-Phasen dieser Direktiven nicht mit der eigentlichen Template, sondern mit dem Ergebnis der Transformation aus der Compile-Phase.

Beispiel 4-66 veranschaulicht den Einsatz der Compile-Phase, indem es der Direktiveneigenschaft `compile` eine Funktion zuweist. Wie die Funktion `link` nimmt auch diese Funktion ein jQuery-Objekt entgegen, das die Template repräsentiert, sowie ein Objekt, welches die Attribute des Elements definiert, mit dem die Direktive dargestellt wird. Da die Funktion `compile` womöglich für mehrere Instanzen, die erst später erzeugt werden, zur Ausführung kommt, erhält sie keinen Scope. Allerdings nimmt sie über den dritten Parameter eine Funktion zur manuellen Durchführung einer Transklusion entgegen. Per Definition liefert die `compile`-Funktion ein Objekt mit den Eigenschaften `pre` und `post` zurück. Beide Eigenschaften verweisen auf auszuführende `link`-Funktionen, die nachfolgend als `preLink` und `postLink` bezeichnet werden. Für sämtliche erzeugte Instanzen bringt AngularJS zuerst `preLink` zur Ausführung. Erst danach wird für alle Instanzen `postLink` angestoßen. Eine eventuell im Rahmen der Direktive festgelegte `link`-Funktion kommt in diesem Fall nicht zur Ausführung.

Die Funktion `compile` in Beispiel 4-66 kümmert sich um das Abrunden der Ecken und gibt anschließend ein Objekt mit einer leeren `preLink`-Funktion und einer `postLink`-Funktion zurück. Letztere führt eine manuelle Transklusion aus. Dazu ruft sie die an `compile` übergebene Funktion `transcludeFn` auf. Diese erwartet den Parent-Scope sowie eine weitere Funktion, an die sie den zu transkludierenden Text übergibt, als Parameter. Die übergebene Funktion nimmt diesen Text entgegen und platziert ihn innerhalb der Template mit den Möglichkeiten von jQuery.

Da die betrachtete postLink-Funktion anstatt der link-Funktion in der Direktive ausgeführt wird, delegiert sie an diese link-Funktion, um eventuelle Überraschungen zu vermeiden.

Beispiel 4-68: Kommunikation zwischen Direktiven

```
var app = angular.module("directivesSample", []);

app.directive("coolCheckbox", function () {

    return {
        //template: "<div class='coolCheckBoxOff'>Checkbox</div>"
        templateUrl: "/templates/coolCheckBoxTemplate3.html",
        transclude: true,
        restrict: "ECA",
        scope: {
            isChecked: "=isChecked",
            stateChanged: "&onchange"
        },
        replace: true,
        controller: function ($scope, $element, $attrs, $transclude) {
            // $scope.title = "Haaacked"; // wenn nichts anderes angegeben wird, bekommt die Direktive
den selben Scope, wie ihr Parent

            // $transclude(); // Liefert den Inhalt des Elements

            $scope.toggleState = function () {
                $scope.isChecked = !$scope.isChecked;
                $scope.stateChanged();
            }

        },
        compile: function (tElement, tAttrs, transcludeFn) {

            var checkBoxElement = tElement; //.children().first();

            if (tAttrs.round && tAttrs.round.toLowerCase() == "true") {
                checkBoxElement.css("border-radius", "5px");
            }

            var this_link = this.link;

            return {
                pre: function preLink(scope, iElement, iAttrs, controller) {
                },
                post: function postLink(scope, iElement, iAttrs, controller) {

                    // Manuelle Transklusion
                    transcludeFn(scope.$parent, function (transElement) {
                        var checkBoxElement = iElement; //.children().first();
                        checkBoxElement.children().first().before(transElement);
                    });

                    this_link(scope, iElement, iAttrs);
                }
            }
```

```
        },
        link: function postLink(scope, iElement, iAttrs) {
            var setCssClass = function () {
                if (scope.isChecked) {
                    checkBoxElement.removeClass("coolCheckBoxOff");
                    checkBoxElement.addClass("coolCheckBoxOn");
                }
                else {
                    checkBoxElement.removeClass("coolCheckBoxOn");
                    checkBoxElement.addClass("coolCheckBoxOff");
                }

                //scope.$apply();
            };

            var checkBoxElement = iElement; //.children().first();
            setCssClass();

            checkBoxElement.on("click", function () {
                scope.toggleState();
                setCssClass();
                scope.$apply();
            });
        }
    };

});
```

Kommunikation zwischen Direktiven

Um zu zeigen, wie Direktiven untereinander kommunizieren können, erweitert dieser Abschnitt die in den letzten Abschnitten gezeigte Direktive `coolCheckbox`, sodass sie auch als Ersatz für Optionsfelder (engl. Radio Button) eingesetzt werden kann. Dazu sollen, wie in Beispiel 4-69 gezeigt, mehrere Instanzen von `coolCheckbox` innerhalb einer Instanz der neu zu schaffenden Direktive `coolRadioList` zum Einsatz kommen. Wie bei Optionsfeldern kann der Benutzer jedoch nur eine Option aktivieren. Dies wird erreicht, indem die `coolRadioList` beim Aktivieren einer Option sämtliche anderen Optionen deaktiviert. Um herauszufinden, welche Option aktiviert wurde, erhält `coolCheckbox` (die aus Gründen der Übersicht nicht in `coolRadiobutton` umbenannt wird) ein Attribut `value` sowie `coolRadioList` ein Attribut `selected`, wobei die über `selected` referenzierte Variable an den `value` der gewählten Option gebunden wird. Einen zu Beispiel 4-69 passenden Controller findet man in Beispiel 4-70.

Beispiel 4-69: Einsatz der entwickelten Direktive

```
<div ng-app="directivesSample">

    <div ng-controller="sampleCtrl">

        <h2>{{ title }}</h2>

        <div cool-radio-list selected="selected">

            <div cool-checkbox value="A"
                 is-checked="isChecked1"
                 onchange="changed()">Some Text</div>
```

```
            <div cool-checkbox value="B"
                 is-checked="isChecked2"
                 onchange="changed()">Some Text 2</div>

        </div>

        <p>
            isChecked1: {{ isChecked1 }}, isChecked2: {{ isChecked2 }}
        </p>

        <p>
            selected: {{ selected }}
        </p>

    </div>

</div>
```

Beispiel 4-70: Controller zum Testen der entwickelten Direktive

```
app.controller("sampleCtrl", function ($scope) {
    $scope.title = "Directive-Sample";
    $scope.isChecked1 = true;
    $scope.isChecked2 = false;

    $scope.selected = "A";

    $scope.changed = function () {
        if (window.console) window.console.log("changed ...");
    }
});
```

Beispiel 4-71 zeigt die Implementierung der Direktive `coolRadioList`. Sie besitzt eine einfache Template, die lediglich aus einem `div`-Element besteht, und einen isolierten Scope, der die Eigenschaft `selectedValue` an die über das Attribut `selected` angegebene Eigenschaft bindet.

Der Controller definiert ein Array `checkBoxes` zum Verwalten der Scopes der einzelnen `coolCheckbox`-Direktiven. Außerdem registriert er die beiden Funktionen `registerCoolCheckBox` und `select` unter Verwendung von `this` direkt bei der Direktive. Erstere nimmt den Scope einer untergeordneten `coolCheckbox` entgegen und legt diesen im Array `checkBoxes` ab. Anschließend prüft sie, ob die jeweilige `coolCheckbox` jenen Wert aufweist, der sich in der Eigenschaft `selectedValue` wiederfindet. Ist dem so, wird diese `coolCheckbox` unter Verwendung der weiter unten beschriebenen Funktion `setState` aktiviert. Ansonsten wird sie unter Verwendung derselben Funktion deaktiviert.

Die Funktion `select` wird von einer `coolCheckbox` aufgerufen, wenn sie der Benutzer durch einen Klick aktiviert bzw. deaktiviert. Dabei übergibt die `coolCheckbox` ihren Scope. Wurde die `coolCheckbox` aktiviert, übernimmt `select` ihren Wert in die Eigenschaft `selectedValue`. Wurde die `coolCheckbox` hingegen deaktiviert, bedeutet das, dass derzeit keine Option aktiv ist. Deswegen setzt `select` in diesem Fall die Variable `selectedValue` auf `null`. Da innerhalb einer `coolRadioList` maximal eine Option ausgewählt werden darf, deaktiviert `select` anschließend alle anderen `coolCheckbox`-Instanzen.

Beispiel 4-71: Mit untergeordneten Direktiven kommunizieren

```
var app = angular.module("directivesSample", []);

app.directive("coolRadioList", function () {

    return {
        restrict: 'EA',
        replace: true,
        transclude: true,
        template: '<div ng-transclude></div>',
        scope: {
            selectedValue: "=selected"
        },
        controller: function ($scope) {

            var checkBoxes = [];

            this.registerCoolCheckBox = function(coolCheckboxScope) {
                checkBoxes.push(coolCheckboxScope);

                if (coolCheckboxScope.value != $scope.selectedValue) {
                    coolCheckboxScope.setState(false);
                }
                else {
                    coolCheckboxScope.setState(true);
                }
            }

            this.select = function (selectedCheckboxScope) {

                // selectedValue setzen
                if (selectedCheckboxScope.isChecked) {
                    $scope.selectedValue = selectedCheckboxScope.value;
                }
                else {
                    $scope.selectedValue = null;
                }

                // alle anderen Optionen deaktivieren
                angular.forEach(checkBoxes, function (checkBoxScope) {
                    if (selectedCheckboxScope != checkBoxScope) {
                        checkBoxScope.setState(false);
                    }
                });
            }
        }
    };
});
```

Beispiel 4-72 zeigt die modifizierte Variante der Direktive `coolCheckbox`. Mit der Eigenschaft `require` legt sie fest, dass sie eine Instanz von `coolRadioList` benötigt. Standardmäßig geht AngularJS beim Einsatz von `require` davon aus, dass die benötige Instanz für dasselbe (HTML-)Element definiert wurde. Durch Angabe des hier verwendeten Präfixes (`^`) durchsucht AngularJS jedoch auch sämtliche übergeordneten (HTML-)Elementen nach einer solchen Direktive. Das hier zusätzlich verwendete Präfix (`?`) gibt darüber hinaus an, dass es sich bei der referenzierten Direktive um eine optionale Direktive handelt. Das hat zur Folge, dass AngularJS keinen Fehler auslöst, wenn es diese Direktive nicht findet.

Wird die coolRadioList-Direktive gefunden, übergibt sie AngularJS an die preLink- und postLink-Funktion, wobei die hier verwendete postLink-Funktion an die Funktion link delegiert und dabei die Direktive weiterreicht. Diese Funktion registriert bei der coolRadioList den Scope der coolCheckbox unter Verwendung der zuvor besprochenen Funktion registerCoolCheckBox. Außerdem legt sie fest, dass der Click-Handler der coolCheckbox die coolRadioList durch Aufruf der ebenfalls weiter oben definierten Funktion select zu benachrichtigen hat.

Damit die coolCheckbox nicht nur aufgrund eines Klicks, sondern ebenfalls bei einer Änderung von isChecked, welche nun auch von coolRadioList angestoßen werden kann, die entsprechende CSS-Klasse verwendet und die über das Attribut onchange registrierte Funktion ausführt, überwacht die hier gezeigte Implementierung unter Verwendung von $watch die Eigenschaft isChecked.

Beispiel 4-72: Mit übergeordneten Direktiven kommunizieren

```
app.directive("coolCheckbox", function () {

    return {
        //template: "<div class='coolCheckBoxOff'>Checkbox</div>"
        templateUrl: "/templates/coolCheckBoxTemplate3.html",
        transclude: true,
        restrict: "ECMA",
        replace:true,
        require: '^?coolRadioList',
        scope: {
            isChecked: "=isChecked",
            stateChanged: "&onchange",
            value: "@value"
        },
        //replace: true,
        controller: function ($scope, $element, $attrs, $transclude) {
            // $scope.title = "Haaacked"; // wenn nichts anderes angegeben wird, bekommt die Direktive
den selben Scope, wie ihr Parent

            // $transclude(); // Liefert den Inhalt des Elements

            $scope.toggleState = function () {
                $scope.isChecked = !$scope.isChecked;
            }

            $scope.setState = function (newState) {
                $scope.isChecked = newState;
            }
        },
        compile: function (tElement, tAttrs, transcludeFn) {

            var checkBoxElement = tElement; //.children().first();

            if (tAttrs.round && tAttrs.round.toLowerCase() == "true") {
                checkBoxElement.css("border-radius", "5px");
            }

            var this_link = this.link;

            return {
                pre: function (scope, iElement, iAttrs, coolRadioListController) {
```

```
                },
                post: function (scope, iElement, iAttrs, coolRadioListController) {

                    // Manuelle Transklusion
                    transcludeFn(scope.$parent, function (transElement) {
                        var checkBoxElement = iElement; //.children().first();
                        checkBoxElement.children().first().before(transElement);
                    });

                    this_link(scope, iElement, iAttrs, coolRadioListController);
                }
            }
        },
        link: function (scope, iElement, iAttrs, coolRadioListController) {

            if (coolRadioListController)
                coolRadioListController.registerCoolCheckBox(scope);

            var setCssClass = function () {
                if (scope.isChecked) {
                    checkBoxElement.removeClass("coolCheckBoxOff");
                    checkBoxElement.addClass("coolCheckBoxOn");
                }
                else {
                    checkBoxElement.removeClass("coolCheckBoxOn");
                    checkBoxElement.addClass("coolCheckBoxOff");
                }
            };

            var checkBoxElement = iElement;
            setCssClass();

            scope.$watch('isChecked', function () {
                setCssClass();
                scope.stateChanged();
            });

            checkBoxElement.on("click", function () {
                scope.toggleState();

                if (coolRadioListController)
                    coolRadioListController.select(scope);

                scope.$apply();
            });
        }
    };
});
```