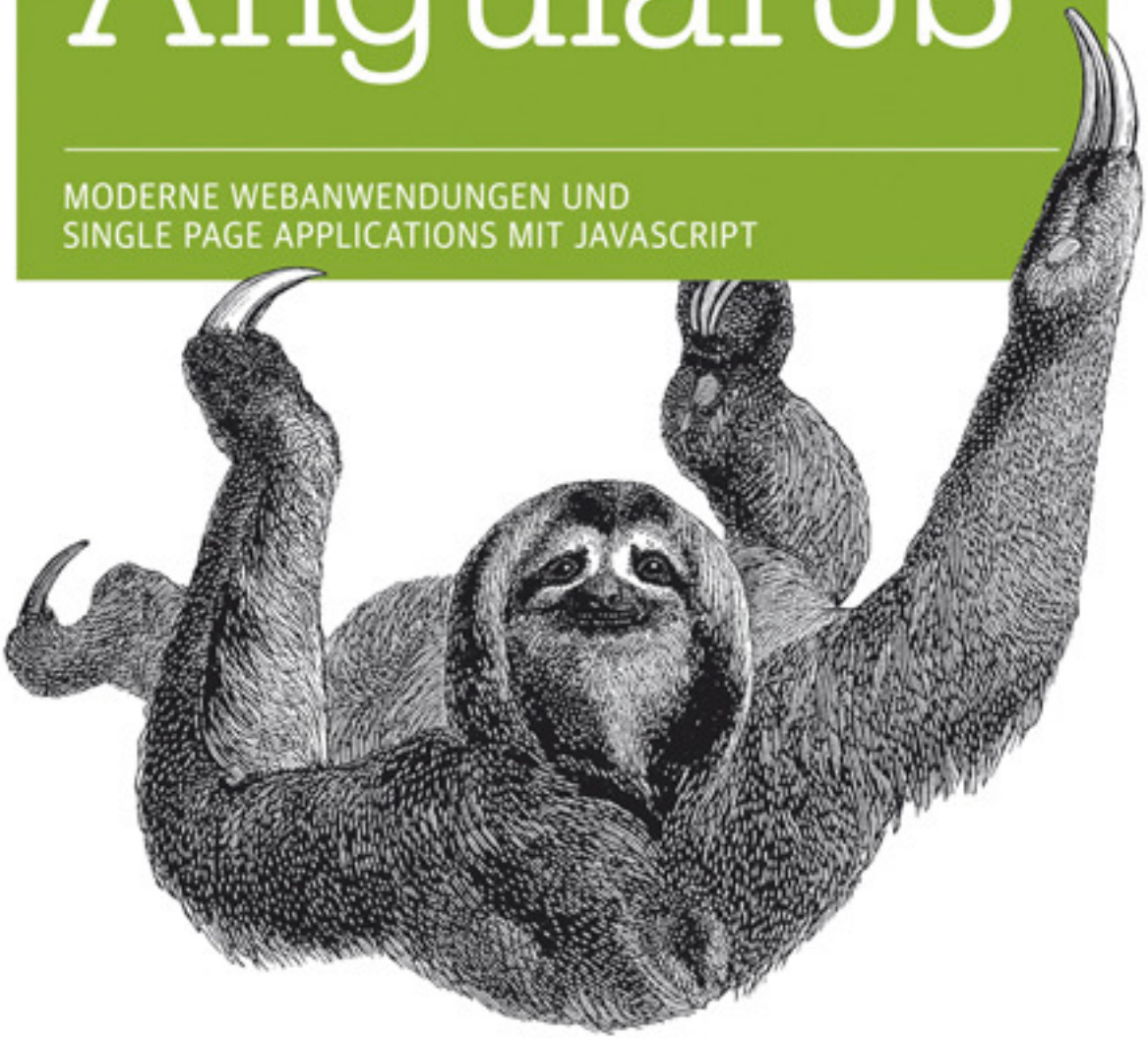
O'REILLY®

AngularJS

MODERNE WEBANWENDUNGEN UND
SINGLE PAGE APPLICATIONS MIT JAVASCRIPT



Manfred Steyer, Vildan Softic

Inhalt

KAPITEL 4

Formulare und Validierung

AngularJS bietet einige Möglichkeiten, Daten und Funktionen an Steuerelemente, wie Formularfelder, Kontrollkästchen (engl. Checkbox) oder Schaltflächen, zu binden. Darüber hinaus bietet AngularJS 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 den Umgang mit Formularen zu demonstrieren, verwenden die nachfolgenden Abschnitte den Controller in Beispiel 4-1. Dieser spendiert dem Scope eine Eigenschaft *vm*, welche auf ein Objekt verweist, das als View-Model fungiert und über die 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 4-1: 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",
        anmerkungen: "Leider um 30 Minuten verspätet ..."
    };

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

});
```

Datenbindung bei Textfeldern

Dass der Entwickler Eigenschaften von Models mit *ng-model* an ein Textfeld binden kann, hat das einführende Beispiel in Kapitel 2 gezeigt. Der Vollständigkeit halber wiederholt Beispiel 4-2 den Einsatz dieser Direktive, bevor die nächsten Abschnitte weitere Möglichkeiten der Datenbindung für Formulare präsentieren.

Beispiel 4-2: Daten an ein Textfeld 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>

        […]

    </form>
</div>
```

Datenbindung bei Checkboxes

Beispiel 4-3 zeigt, wie der Entwickler ein Kontrollkästchen (engl. Checkbox) an eine Eigenschaft des Models binden kann. Dazu verwendet es die, auch schon im letzten Beispiel verwendete, Direktive *ng-model*. Wertet JavaScript den an Checkbox gebundenen Wert als *true* aus, aktiviert AngularJS die Checkbox.

Beispiel 4-3: 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>
```

Datenbindung bei Optionsfeldern

Ein Beispiel für das Binden von Optionsfeldern (engl. Radio Button) findet sich in Beispiel 4-4. 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. Aktiviert der Benutzer ein Optionsfeld, schreibt AngularJS jenen Wert, den der Entwickler mit dem Attribut *value* angegeben hat, in die gebundene Eigenschaft.

Beispiel 4-4: 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 4-5 demonstriert dies.

Beispiel 4-5: 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>
```

Datenbindung bei Dropdown-Feldern

Für das Generieren eines Dropdown-Felds bietet AngularJS die Direktive *ng-options* an. Diese Direktive erwartet einen Ausdruck, der einer simplen vorgegebenen Grammatik folgt. Beispiel 4-6 demonstriert dies, indem es ein Dropdown-Feld 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 4-6: Daten an ein Dropdown-Feld 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 Dropdown-Feld, welches die Einträge nach der in dieser Klausel angegebenen Eigenschaft gruppiert. Auf diese Weise gruppiert das Listing in Beispiel 4-7 die Vorschlagswerte nach der Eigenschaft *allianz*.

Beispiel 4-7: Dropdown-Feld 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>
```

Datenbindung bei textarea-Elementen

Die Datenbindung bei textarea-Elementen (Textbereich), welche ein mehrzeiliges Eingabefeld darstellen, erfolgt analog zur Datenbindung bei input- und select-Elementen mit der Direktive *ng-model* (vgl. Beispiel 4-8).

Beispiel 4-8: Nutzung eines Textbereichs

```
<div class="form-group">

    <label for="anmerkungen">Anmerkungen</label>

    <textarea ng-model="vm.flug.anmerkungen" id="anmerkungen" name="anmerkungen"
              class="form-control"></textarea>
</div>
```

Eingaben validieren

AngularJS bietet einige Bordmittel, um Benutzereingaben zu validieren. Dieser Abschnitt präsentiert diese samt der dahinterliegenden Konzepte.

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 *$dirty* auf. Diese zeigt an, ob der Benutzer ein Feld des Formulars geändert hat. Genau das Gegenteil davon zeigt die Eigenschaft *$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 sogenannte Model-Controller, 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 Listing in Beispiel 4-9 demonstriert dies, indem es die genannten Eigenschaften sowohl auf Formular- als auch auf Steuerelementebene anzeigt.

Beispiel 4-9: 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 weisen Form-Controller und Model-Controller Eigenschaften auf, die darüber informieren, ob AngularJS das Formular bzw. das jeweilige Steuerelement erfolgreich validieren konnte. Der nächste Abschnitt geht hierauf genauer ein.

Valdierungsattribute

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 Textes definieren. Daneben steht noch eine Direktive *ng-pattern* zur Verfügung, welche die Eingabe anhand eines regulären Ausdrucks validiert. Beispiel 4-10 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*, verhindert. Dieses Attribut wirkt sich jedoch nicht auf die Validierung durch AngularJS aus. Wichtig 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 Model-Controllers, ob AngularJS das Steuerelement erfolgreich validieren konnte. Dazu greift es auf die Eigenschaft *form.flugNummer.$invalid* zu. Eine Fehlermeldung erscheint jedoch nur dann, wenn der Benutzer zusätzlich den Wert im Eingabefeld geändert hat. Diesen Umstand kann der Entwickler über die Eigenschaft *form.flugNummer.$dirty* prüfen. 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 AngularJS sämtliche Felder im Formular erfolgreich validieren konnte.

Darüber hinaus prüft das gezeigte Beispiel ebenfalls, welche Validierungslogik fehlgeschlagen ist. Dazu greift es auf die Eigenschaft *form.flugNummer.$error* zu. Diese verweist auf ein Objekt, welches 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 4-10: 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 Listing in Beispiel 4-11 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 4-11: 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>
```

AngularJS validiert nicht nur Eingabefelder des Typs *number*, sondern unterstützt auch andere durch HTML5 definierte Typen wie *url* oder *email*. Auch für datumsbezogene Typen wie *date* oder *time* bietet es standardmäßig eine Validierung, wobei diese, zumindest bei der den Autoren vorliegenden Version 1.3.x, nicht in lokalisierter Form erfolgt. Lösungen hierfür bietet das Kapitel 6.

Validierungsfehler anzeigen

Zum Anzeigen von Validierungsfehlern kann der Entwickler, wie in den vorausgegangenen Abschnitten gezeigt, *ng-show* einsetzen. Um diese Aufgabe zu vereinfachen, bietet AngularJS seit Version 1.3 ein Modul namens *ngMessages*. Um in den Genuss dieser Datei zu kommen, bindet der Entwickler zunächst die Skriptdatei, welche *ngMessages* beinhaltet, ein:

```
<script src="scripts/angular-messages/angular-messages.min.js"></script>
```

Das Modul *ngMessages* steht auch via Bower zur Verfügung:

```
bower install angular-messages
```

Zusätzlich ist das Modul zu importieren:

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

Sind diese Voraussetzungen geschaffen, kann der Entwickler basierend auf dem von AngularJS pro Eingabefeld veröffentlichen Objekt *$error* eine Fehlermeldung anzeigen. Beispiel 4-12 veranschaulicht dies. Die Direktive *ng-messages* bekommt das *$error*-Objekt übergeben und alle untergeordneten Elemente repräsentieren die einzelnen Fehlermeldungen. Für jedes dieser Elemente legt *ng-message* die damit zu verknüpfende Validierungsregel, wie zum Beispiel *required*, fest. Verletzt der Benutzer eine dieser Regeln, erhält *$error* von AngularJS eine Eigenschaft mit dem Regelnamen, die den Wert *true* aufweist. In diesem Fall blendet das Zusammenspiel von *ng-message* die jeweilige Fehlermeldung ein.

Beispiel 4-12: Anzeigen von Fehlermeldungen mit ngMessages

```
<div ng-messages="form.von.$error">

    <div ng-message="required">Pflichtfeld!</div>

    <div ng-message="minlength">Mindestens 3 Zeichen!</div>

    <div ng-message="maxlength">Maximal 30 Zeichen!</div>

</div>
```

Standardmäßig zeigt die Direktive *ng-messages* nur eine einzige Fehlermeldung an. Um diese zu ermitteln, durchläuft sie sämtliche konfigurierten Fehlermeldungen in der Reihenfolge ihres Vorkommens und zeigt die erste an, deren Validierungsregel nicht erfüllt ist. Um *ng-messages* dazu zu bringen, für alle Validierungsfehler eine Meldung anzuzeigen, verwendet der Entwickler das Attribut *multiple*:

```
    <div ng-messages="form.von.$error" multiple>
```

Die Direktiven *ng-messages* und *ng-message* lassen sich auch in Form von Elementen einsetzen. Das Listing in Beispiel 4-13 veranschaulicht diese Schreibweise. Das Attribut for nimmt das *$error*-Objekt entgegen, während das Attribut *when* den Fehler, für den das *ng-message*-Element eine Fehlermeldung parat hält, aufnimmt.

Beispiel 4-13: Alternativer Einsatz von ngMessages

```
<ng-messages for="form.von.$error" >

    <ng-message when="required">Pflichtfeld!</ng-message>

    <ng-message when="minlength">Mindestens 3 Zeichen!</ng-message>

    <ng-message when="maxlength">Maximal 30 Zeichen!</ng-message>

</ng-messages>
```

Um nicht immer dieselben Fehlermeldungen hinterlegen zu müssen, kann der Entwickler die einzelnen Fehlermeldungen auch in eigene Dateien auslagern. Dazu gibt er mit dem Attribut *include* der Direktive *ng-messages* den Namen der heranzuziehenden Datei bekannt:

```
    <div ng-messages="form.von.$error" include="errors.html"></div>
```

Die Datei besteht in diesem Fall lediglich aus Elementen, die sich auf die Direktive *ng-message* stützen:

```
    <div ng-message="required">Pflichtfeld!</div>

    <div ng-message="minlength">Zu kurz!</div>

    <div ng-message="maxlength">Zu lang!</div>
```

Anstatt diese Fehlermeldungen in eine eigene Datei auszulagern, kann der Entwickler sie auch in einem globalen Script-Tag platzieren (Beispiel 4-14). Dieser muss sich innerhalb des mit *ng-app* markierten Elements befinden, damit AngularJS ihn entdeckt. Über das Attribut *type* legt der Entwickler den Typ *text/ng-template* fest und über das Attribut *id* benennt er den Script-Tag.

Beispiel 4-14: Interne Vorlage mit Fehlermeldungen

```
<script type="text/ng-template" id="errors">

    <div ng-message="required">Pflichtfeld!!!</div>

    <div ng-message="minlength">Mindestens 3 Zeichen!!!</div>

    <div ng-message="maxlength">Maximal 30 Zeichen!!!</div>

</script>
```

Um auf solch einen Script-Bereich zu verweisen, gibt der Entwickler dessen Id über das Attribut *include* an:

```
    <div ng-messages="form.von.$error" include="errors"></div>
```

Fehlerhafte Eingaben mit CSS hervorheben

Damit die Anwendung Eingabefelder in Hinblick auf die Validität ihrer Inhalte formatieren kann, weist ihnen AngularJS abhängig von ihrem Zustand vordefinierte CSS-Klassen zu (Tabelle 4-1). 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 AngularJS korrekt validieren konnte, erhalten die Klasse *ng-valid*. Felder, bei denen das nicht der Fall ist, erhalten die Klasse *ng-invalid*. AngularJS vergibt hingegen die Klasse *ng-dirty*, wenn der Inhalt des Felds geändert wurde.

Zur Veranschaulichung der damit verbundenen Möglichkeiten definiert Beispiel 4-15 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, stellt die Anwendung hingegen mit einer grünen Hintergrundfarbe dar.

Beispiel 4-15: CSS-Klassen zur Formatierung von Steuerelementen

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

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

Tabelle 4-1: CSS-Klassen, mit denen AngularJS auf die Validität von Feldern hinweist

CSS-Klasse	Beschreibung
ng-valid	Es liegen keine Validierungsfehler vor
ng-invalid	Es liegen Validierungsfehler vor
ng-valid-[key]	Es liegen Validierungsfehler für die Validierungsregel [key] vor (z.B. *ng-valid-require* für die Validierungsregel *require*)
ng-invalid-[key]	Es liegen Validierungsfehler für die Validierungsregel [key] vor (z.B. *ng-invalid-require* für die Validierungsregel *require*)
ng-pristine	Der Benutzer hat den Wert nicht verändert
ng-dirty	Der Benutzer hat den Wert verändert
ng-touched	Der Benutzer hat das Feld (mit oder ohne) Änderungen verlassen
ng-untouched	Der Benutzer hat das Feld noch nicht verlassen
ng-pending	AngularJS wartet noch auf das Ergebnis asynchroner Validatoren (vgl. Abschnitt »Asynchrone Validatoren«)

Auf Validierungsergebnisse programmatisch zugreifen

Um zur Laufzeit zu ermitteln, ob ein Formular veränderte Werte beinhaltet bzw. Validierungsfehler aufweist, greift der Entwickler über den Scope auf den mit dem Formular assoziierten Form-Controller zu. Beispiel 4-16 demonstriert dies.

Beispiel 4-16: 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 4-17 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 Validierungslogiken den erfassten Wert nicht korrekt validieren konnte.

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

Beispiel 4-17: 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 musste. Um dies zu umgehen, zeigt Beispiel 4-18, wie sämtliche Form-Controller im aktuellen Scope zu finden sind. Da die Konstruktorfunktion der Form-Controller nicht außerhalb von AngularJS verfügbar ist, muss die Funktion prüfen, ob Objekte existieren, die so aussehen wie Form-Controller.

Beispiel 4-18: 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;
}
```

Verschachtelte Formulare

Wiederholungen in Formularen führen zu nicht ganz offensichtlichen Problemen bei der Validierung, Beispiel 4-19 demonstriert dies. Es zeigt ein Formular mit einer Tabelle, welche Airlines auflistet. Jede Zeile steht für eine Airline und jede Zelle enthält ein Eingabefeld für eine Eigenschaft der jeweiligen Airline. Das Feld, welches an die Eigenschaft *name* gebunden ist, hat der Entwickler mit dem Attribut *required* als Pflichtfeld gekennzeichnet. Im Fall einer erfolglosen Validierung erhält die umschließende Zelle die CSS-Klasse *has-error*, welche dazu führt, dass das Eingabefeld einen roten Rahmen erhält.

Beispiel 4-19: Problematisches Markup mit sich wiederholenden Formular-Elementen

```
<form name="form1">
    <table>

        <tr>
            <th>Id</th>
            <th>Name</th>
            <th>Allianz</th>
        </tr>

        <tr ng-repeat="a in vm.airlines">
            <td><input ng-model="a.id" name="id" class="form-control"></td>
            <td class="form-group" ng-class="{'has-error': form1.name.$invalid}">
                <input ng-model="a.name" name="name" required class="form-control">
            </td>
            <td><input ng-model="a.allianz" name="allianz" class="form-control"></td>
        </tr>

    </table>
</form>
```

Auf den ersten Blick funktioniert dieses Beispiel wie gewünscht. Bei näherer Betrachtung fällt hingegen auf, dass die Anwendung nur Validierungsfehler für die letzte Zeile anzeigt. Ein solcher Validierungsfehler führt jedoch dazu, dass AngularJS sämtliche Felder mit einem roten Rahmen darstellt. Dies liegt daran, dass die Direktive *ng-repeat* pro Wiederholung eine Eigenschaft *form1.name* einrichtet. Existiert diese Eigenschaft bereits, über-

schreibt *ng-repeat* sie. Aus diesem Grund ist nur die letzte Iteration für die Validierung ausschlaggebend.

Dieses Problem kann der Entwickler durch Einsatz der Direktive *ng-form* umgehen. Diese Direktive ist nicht als Ersatz für das Element *form* gedacht, sondern kommt zum Unterteilen eines Formulars in mehrere Abschnitte zum Einsatz. Pro Abschnitt richtet AngularJS einen eigenen untergeordneten Scope mit einem *Form*-Controller ein. Somit existiert pro Abschnitt ein eigener Form-Controller und der Entwickler kann jeden Abschnitt für sich separat validieren.

Beispiel 4-20 definiert zur Veranschaulichung für jede mit *ng-repeat* durchgeführte Wiederholung einen Abschnitt, den AngularJS auch separat validiert, ein. An *ng-form* übergibt das betrachtete Beispiel den Namen des jeweiligen Abschnitts. Dieser lautet auf *airlineForm*. Auf den für die Validierung zu nutzenden Controller greift es in weiterer Folge unter Angabe dieses Namens zu (*airlineForm.name*).

Beispiel 4-20: Wiederholgruppen in Formular mit ng-form

```
<form name="form2">

    <table>

        <tr>
            <th>Id</th>
            <th>Name</th>
            <th>Allianz</th>
        </tr>

        <tr ng-repeat="a in vm.airlines" ng-form="airlineForm">
            <td><input ng-model="a.id" name="id" class="form-control"></td>
            <td class="form-group" ng-class="{'has-error': airlineForm.name.$invalid}">
                <input ng-model="a.name" name="name" required class="form-control">
            </td>
            <td><input ng-model="a.allianz" name="allianz" class="form-control"></td>
        </tr>

    </table>


</form>
```

Möchte der Entwickler nun herausfinden, ob in einem der durch das Zusammenspiel von *ng-form* und *ng-repeat* generierten Abschnitten ein Validierungsfehler vorliegt, kann er für diese Abschnitte einen gemeinsamen übergeordneten Abschnitt definieren. Das Listing in Beispiel 4-21 nennt diesen Abschnitt, welchen es auf der Ebene der Tabelle einrichtet, *outerForm*. Im unteren Bereich des Beispiels prüft das Beispiel unter Verwendung von *outerForm.$invalid*, ob innerhalb dieses Abschnitts, welcher die Wiederholgruppe beinhaltet, ein Validierungsfehler vorkommt.

Beispiel 4-21: Validierungsinformationen für mehrere Formularabschnitte ermitteln

```
<form name="form2">

    <table ng-form="outerForm">

        <tr>
            <th>Id</th>
            <th>Name</th>
            <th>Allianz</th>
        </tr>

        <tr ng-repeat="a in vm.airlines" ng-form="airlineForm">
            <td><input ng-model="a.id" name="id" class="form-control"></td>
            <td class="form-group" ng-class="{'has-error': airlineForm.name.$invalid}">
                <input ng-model="a.name" name="name" required class="form-control">
            </td>
            <td><input ng-model="a.allianz" name="allianz" class="form-control"></td>
        </tr>

    </table>

    <div ng-show="outerForm.$invalid" style="color: red">
        Tabelle mit Airlines weist Fehler auf!
    </div>
</form>
```

Auf unsichere Eingaben reagieren

Um zu vermeiden, dass Angreifer, zum Beispiel über Formularfelder, Schadcode in die Anwendung einschleusen, kodiert AngularJS standardmäßig unsichere Inhalte, wie z.B. HTML-Elemente, bei der Datenbindung. Dies soll hier anhand des Models in Beispiel 4-22 veranschaulicht werden.

Beispiel 4-22: Eigenschaft mit HTML-Elementen

```
app.controller("editFlugCtrl", function ($scope) {
    $scope.vm = {};

    $scope.vm.flug = {
        flugNummer: "LH4711",
        von: "Graz",
        nach: "Frankfurt",
        datum: new Date(),
        anmerkungen: "<b>Hallo Welt!</b><a onmouseover='this.textContent = \"Aua!\"'>Klick
mich!</a>"
    }
};
```

Bindet der Entwickler die Eigenschaft *anmerkungen*, welche HTML-Markup beinhaltet, unter Verwendung von *{{vm.flug.anmerkungen}}*, schreibt AngularJS eine HTML-codierte Version dieses Markups in die Seite, sodass diese den String ohne Berücksichti-

gung des Markups wiederspiegelt. Möchte der Entwickler hingegen, dass der Browser das Markup interpretiert, muss er die Direktive *ng-bind-html* heranziehen:

```
<span ng-bind-html="vm.flug.anmerkungen"></span>
```

Um zu verhindern, dass AngularJS auf diesem Weg Schadcode in der Anwendung platziert, erzwingt es eine sogenannte Bereinigung (engl. Sanitization) des Markups. Ohne diese quittiert *ng-bind-html* den Dienst mit einer Fehlermeldung. Damit *ng-bind-html* solch eine Bereinigung durchführen kann, bindet der Entwickler das Modul *ngSanitize* ein. Dazu ist die Skriptdatei dieses Moduls zu inkludieren:

```
<script src="scripts/angular-sanitize/angular-sanitize.min.js"></script>
```

Darüber hinaus ist das Modul in das eigene zu importieren:

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

Ist damit die Voraussetzung für die Bereinigung geschaffen, erledigt *ng-bind-html* den Rest. Im betrachteten Beispiel würde *ng-bind-html* im Zuge der Datenbindung die Ereignisbehandlungsroutine *onmouseover* aus dem Markup in der Eigenschaft *anmerkungen* entfernen.

Der Entwickler hat aber auch die Möglichkeit, einen String manuell zu bereinigen. Dazu lässt er sich die Funktion *$sanitize* injizieren, an welche er den String übergibt. Die Funktion *getAnmerkungenSanitized* im Model, das Beispiel 4-23 verwendet, veranschaulicht dies.

Beispiel 4-23: Manuelle Bereinigung mit $sanitize

```
app.controller("editFlugCtrl", function ($scope, $sanitize) {
    $scope.vm = {};

    $scope.vm.flug = {
        flugNummer: "LH4711",
        von: "Graz",
        nach: "Frankfurt",
        datum: new Date(),
        anmerkungen: "<b>Hallo Welt!</b><a onmouseover='this.textContent = \"Aua!\"'>
            Klick mich!</a>",
        getAnmerkungenSanitized: function() { return $sanitize(this.anmerkungen); },
    }
};
```

Alternativ dazu kann der Entwickler auch explizit einen String als sicher markieren. Dazu besorgt er sich über den Dependency-Injection-Mechanismus von AngularJS den Service *$sce* (Strict Contextual Escaping). Dieser bietet eine Funktion *trustAsHtml*, die einen String entgegennimmt und ein als sicher markiertes Gegenstück retourniert (Beispiel 4-24). Bindet der Entwickler solch einen String mit *ng-bind-html*, platziert AngularJS den String ohne Kodierung in der Seite. Im betrachteten Fall würde somit *ng-bind-html* auch die Ereignisbehandlungsroutine *onmouseover* einrichten.

Beispiel 4-24: String als sicher markieren

```
app.controller("editFlugCtrl", function ($scope, $sce) {
    $scope.vm = {};

    $scope.vm.flug = {
        flugNummer: "LH4711",
        von: "Graz",
        nach: "Frankfurt",
        datum: new Date(),
        anmerkungen: $sce.trustAsHtml(
            "<b>Hallo Welt!</b><a onmouseover='this.textContent = \"Aua!\"'>Klick mich!</a>")
    }
};
```

Auch wenn es nicht empfehlenswert erscheint, kann der Entwickler den hier beschriebenen Sicherheitsmechanismus, den die Dokumentation als Strict-Contextual-Escaping-Modus bezeichnet, deaktivieren. Dazu besorgt er sich den Service *$sceProvider* via Dependency Injection und übergibt *false* an dessen Funktion *enabled*:

```
$sceProvider.enabled(false);
```

Dies hat zur Folge, dass der Entwickler jeden beliebigen String mit *ng-bind-html* an die Ausgabe binden darf.

Datenbindungs- und Validierungsverhalten konfigurieren

Standardmäßig validiert AngularJS unmittelbar nach jeder Wertänderung durch den Benutzer die gebundenen Felder. Das bedeutet, dass AngularJS bei Eingabefeldern im Zuge jedes Tastendrucks eine neue Validierung anstößt. Konnte AngularJS den Wert korrekt validieren, schreibt es ihn in das Model zurück; ansonsten schreibt es den Wert *undefined* in die jeweilige Model-Eigenschaft, was dazu führt, dass JavaScript die Eigenschaft (vorübergehend) entfernt.

Seit Version 1.3 gibt AngularJS dem Entwickler die Möglichkeit, dieses Verhalten anzupassen. Dazu verwendet er die Direktive *ng-model-options*. Diese Direktive platziert er auf demselben Element wie *ng-model*, oder auf einem übergeordneten Element. Das ist möglich, weil *ng-model* den DOM-Baum (Document Object Model) nach oben klettert und die erste gefundene *ng-model-options*-Direktive heranzieht. Auf diese Weise kann der Entwickler Standardwerte auf der Ebene des Formulars festlegen und diese bei Bedarf auf der Ebene einzelner Felder überschreiben.

Das Listing in Beispiel 4-25 veranschaulicht den Einsatz von *ng-model-options*. Es platziert eine Instanz der Direktive auf einem *form*-Element. Diese Direktive erhält ein JSON-Objekt mit den festzulegenden Optionen. Die Eigenschaft *updateOn* legt jene JavaScript-Ereignisse, die zur Validierung bzw. zum Retourschreiben des Werts in das Model führen sollen, fest. Der Wert von *updateOn* ist ein String, welcher die Namen dieser Ereignisse, getrennt durch Leerschritte, beinhaltet. Das betrachtete Beispiel vergibt hierzu den Wert

blur. Das führt dazu, dass das Validieren bzw. das Retourschreiben erst beim Verlassen des jeweiligen Felds im Zuge des *blur*-Ereignisses stattfindet.

Das zweite *input*-Element überschreibt die durch das übergeordnete *form*-Element festgelegten Einstellungen, indem es ebenfalls eine Instanz der *ng-model-options*-Direktive nutzt. Die Eigenschaft *updateOn* erhält hier den Wert *default*, was das Standardverhalten von AngularJS – zumindest für das betroffene *input*-Element – wiederherstellt. Wie bereits angemerkt, sieht dieses Standardverhalten vor, dass das Validieren und Zurückschreiben bei jeder Änderung des Werts stattfindet.

Beispiel 4-25: Einsatz von ng-model-options

```
<form name="form" id="form" ng-model-options="{updateOn: 'blur'}">

        <div class="form-group">
            <label for="flugNummer">FlugNummer</label>
            <div><input class="form-control"
                        ng-model="vm.flug.flugNummer"
                        name="flugNummer"
                        id="flugNummer" /></div>
            <span ng-show="form.flugNummer.$invalid" style="color:red">Validierung
            fehlgeschlagen!</span>
        </div>

        <!-- Textbox -->
        <div class="form-group">
            <label for="preis">Preis</label>
            <div><input class="form-control"
                        type="number"
                        ng-model="vm.flug.preis"
                        ng-model-options="{updateOn: 'default'}"
                        name="preis"
                        id="preis" /></div>
            <span ng-show="form.preis.$invalid" style="color:red">Validierung
            fehlgeschlagen!</span>
        </div>
</form>
```

Neben *updateOn* bietet *ng-model-options* auch eine Einstellung *debounce* an. Diese legt eine zeitliche in Millisekunden anzugebende Verzögerung für das Validieren bzw. Zurückschreiben fest:

```
<form name="form" ng-model-options="{updateOn: 'default', debounce: 500}">
```

Diese Option ist unter anderem nützlich, wenn der Entwickler zwar die Validierung im Zuge der Eingabe anstoßen möchte, jedoch verhindern will, dass ungültige Werte, die kurzzeitig durch die Bearbeitung vorliegen, unmittelbar eine Fehlermeldung provozieren. Außerdem verhindert diese Verzögerung ein unnötig häufiges Anstoßen von registrierten Ereignis-Handlern im Zuge der Eingabe durch den Benutzer. Dies verbessert auch die Leistung des Systems.

Gibt der Entwickler über *updateOn* mehrere Ereignisse an, hat er auch die Möglichkeit, für jedes Ereignis eine eigene Verzögerung zu definieren:

```
<form name="form" ng-model-options="{ updateOn: 'default blur', debounce: { 'default':
3000, blur: 1 }}">
```

Eine weitere von *ng-model-options* gebotene Eigenschaft ist *allowInvalid*. Diese Eigenschaft steuert den Wert, den AngularJS nach einer fehlerhaften Validierung ins Model zurückschreibt. Das Standardverhalten von AngularJS sieht vor, dass in diesem Fall der Wert *undefined* ins Model zurückzuschreiben ist. Setzt der Entwickler *allowInvalid* jedoch auf *true*, übernimmt AngularJS stattdessen den nicht erfolgreich validierten Wert ins Model:

```
<form name="form" ng-model-options="{ allowInvalid:true }">
```

Dank *ng-model-options* kann der Entwickler nun auch Funktionen, die er als Setter und Getter für einen Wert des Models eingerichtet hat, mittels *ng-model* binden. Wie unter JavaScript üblich, liefern diese Funktionen den jeweiligen Wert zurück, wenn der Aufrufer keinen Wert übergibt. Übergibt der Aufrufer hingegen einen Wert, weist die Funktion diesen Wert der verwalteten Eigenschaft zu. Beispiel 4-26 zeigt ein Model mit solch einer Funktion, die sich *ziel* nennt und den Wert der Eigenschaft *nach* verwaltet.

Beispiel 4-26: Model mit Setter und Getter

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

app.controller("editFlugCtrl", function ($scope) {
    $scope.vm = {};

    $scope.vm.flug = {
        flugNummer: "LH4711",
        von: "Graz",
        nach: "Frankfurt",
        datum: new Date(),
        preis: 400.90,
        maxPassagiere: 300,
        verspaetet: false,
        airline: "LH",
        ziel: function(val) {
            if (arguments.length == 0) return this.nach;
            this.nach = val;
        }
    };
}
```

Um diese Funktion zu binden, setzt der Entwickler die Eigenschaft *getterSetter* von *ng-model-options* auf *true*:

```
<form name="form" ng-model-options="{ getterSetter:true }">
```

Das Binden erfolgt anschließend mit *ng-model* ohne Angabe einer Parameterliste und somit auch ohne runde Klammern:

```
<input class="form-control" ng-model="vm.flug.ziel" />
```

Benutzerdefinierte Erweiterungen

Bei *Formatter*, *Parser* und *Validatoren* handelt es sich um AngularJS-interne Konzepte, derer sich der Entwickler bedienen kann, um die Datenbindung bei Formularfeldern sowie die Validierung von Eingaben zu beeinflussen. Dieser Abschnitt zeigt anhand von Beispielen, was sich hinter diesen Konzepten verbirgt und wie man eigene Formatter, Parser und Validatoren entwickeln kann.

Formatter und Parser

Die Direktive *ng-model* verwendet sogenannte *Formatter* zum Formatieren der in Formularfeldern dargestellten Daten, welche ihren Ursprung im Model haben. Bevor sie die Eingaben nach der Modifikation durch den Benutzer ins Model zurückschreibt, übergibt sie den Wert aus den Eingabefeldern an sogenannte *Parser*. Diese bereiten den ggf. vom Benutzer erfassten oder modifizierten Wert für das Model auf (Abbildung 4-1). Ein Formatter könnte beispielsweise ein Date-Objekt in einen String, das ein Datum repräsentiert, umwandeln, während ein dazugehöriger Parser solch einen String für die Darstellung im Model wieder in ein Date-Objekt umwandeln könnte.

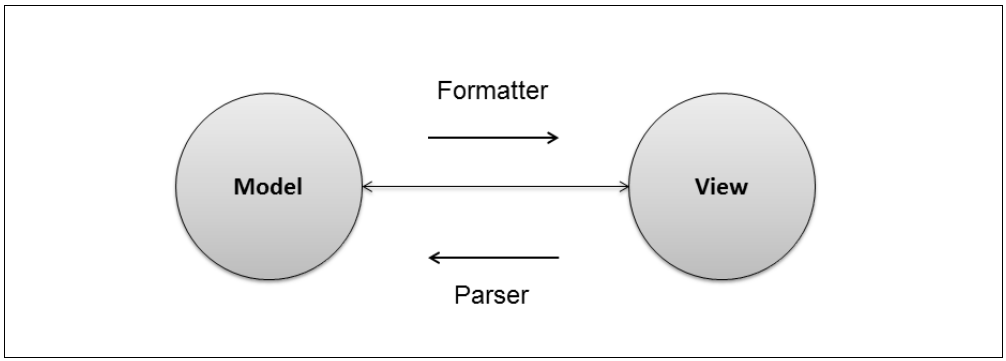


Abbildung 4-1:

Parser, die einen Validierungsfehler entdecken, können diesen an AngularJS melden. Auf diese Weise kann der Entwickler eigene Validierungsregeln bereitstellen. Vor AngularJS 1.3 war dies auch die einzige Möglichkeit, die Validierungsregeln von AngularJS um eigene zu erweitern. Seit Version 1.3 existiert parallel hierzu das Konzept der Validatoren (siehe Abschnitt »Validatoren«), welches sich auf das Validieren von Daten beschränkt und somit einfacher zu nutzen ist.

Bei Formatter und Parser handelt es sich um Funktionen, die einen Wert entgegennehmen und diesen in aufbereiteter Form zurückliefern. Jede Instanz der Direktive *ng-model* verwaltet ein Array mit Formatter sowie ein weiteres mit Parser. Um die Formatter abzurufen, durchläuft sie dieses Array in umgekehrter Reihenfolge. Parser iteriert es hingegen in der durch das Array festgelegten Reihenfolge.

Um einen Wert für die Darstellung in der View aufzubereiten, übergibt sie ihn an die erste auf diese Weise aus dem Formatter-Array abgerufene Funktion und erhält im Gegenzug dafür einen neuen, aufbereiteten Wert. Diesen Wert übergibt sie an die nächste Funktion, welche wiederum einen neuen Wert retourniert. Dieses Verfahren wiederholt sie, bis sie sämtliche Funktionen aufgerufen hat. Das Ergebnis der letzten Funktion bindet sie an das Eingabefeld. Dies zeigt, dass die Reihenfolge der Funktionen im Array für das Ergebnis ausschlaggebend ist.

Beim Aufbereiten einer Eingabe aus der View für das Retourschreiben ins Model geht *ng-model* analog vor: Sie übergibt den Wert an die erste Funktion im Parser-Array und das erhaltene Ergebnis an die zweite Funktion usw.

Um Formatter und Parser bei einer Instanz von *ng-model* zu registrieren, bedarf es einer benutzerdefinierten Direktive. Während Kapitel 14 ausführlich auf die Erstellung benutzerdefinierter Direktiven eingeht, fokussiert dieser Abschnitt auf die Erstellung benutzerdefinierter Direktiven für Formatter und Parser. Beispiel 4-27 zeigt eine solche Direktive, welche sowohl einen Formatter als auch einen Parser bei *ng-model* registriert. Der Formatter schlüsselt eine ID in einen Namen zur Präsentation in der View um. Der Parser beschreitet den umgekehrten Weg, indem er für erfassten Namen jene ID, die im Model zu platzieren ist, umwandelt. Entdeckt der Parser einen Fehler, meldet er diesen an AngularJS.

Zum Definieren der Direktive verwendet das betrachtete Beispiel die Funktion *directive*, welche Modul-Objekte anbieten. Der erste Parameter ist der Name der Direktive; der zweite Parameter erwartet eine Funktion, welche ein Objekt zurückgibt, das die Direktive beschreibt. Die Eigenschaft *require* dieses Objekts legt fest, dass die hier definierte Direktive nur gemeinsam mit *ng-model* zum Einsatz kommen darf. 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 Objekt, das das auf die Direktive angewandte HTML-Element repräsentiert (*element*), ein Objekt, welches die Attribute dieses Elements beinhaltet (*attrs*), sowie ein Objekt, welches die benachbarte *ng-model*-Direktive repräsentiert (*ngModel*).

Das Objekt hinter *ngModel* bietet ein Array *$parsers*, welches sämtliche Parser aufnimmt, sowie ein Array *$formatters*, welches die Formatter verwaltet. Mit *unshift* fügt die *link*-Funktion einen Parser am Beginn des Arrays hinzu und mit *push* platziert sie einen Formatter am Ende. Da *ngModel* die Parser in Array-Reihenfolge und die Formatter in umgekehrter Reihenfolge iteriert, kommen die hier eingerichteten Funktionen jeweils als erste zur Ausführung, sofern keine anderen Direktiven das Array verändern.

Der Formatter nimmt einen Wert aus dem Model und sucht mit der Funktion *find* der beliebten freien JavaScript-Bibliothek *Underscore* (*http://underscorejs.org*) im Array *flughaefen* nach einen Flughafen, dessen ID dem übergebenen Wert entspricht. Daraufhin retourniert er den Namen des gefundenen Flughafens. Wird der Formatter nicht fündig, retourniert er den erhaltenen Wert.

Der Parser geht analog dazu vor, um den Namen aus der View in eine ID zu überführen. Gelingt ihm das, retourniert er die ermittelte ID. Ansonsten zeigt er mit dem Rückgabewert *undefined* an, dass ein Fehler in der Eingabe vorliegt. Darüber hinaus bedient er sich der Funktion *$setValidity* von *ngModel*, um AngularJS über eine erfolgreiche oder gescheiterte Validierung zu informieren. Der erste Parameter dieser Funktion ist der Name einer Validierungsregel, der zweite informiert darüber, ob die Validierung erfolgreich war. Der Wert *true* lässt dabei auf eine erfolgreiche Validierung schließen; der Wert *false* auf einen Verstoß gegen die repräsentierte Validierungsregel.

Beispiel 4-27: Direktive zum Einrichten eines Parsers sowie eines Formatters

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

    var flughaefen = [
        { id: "FRA", name: "Frankfurt" },
        { id: "HAM", name: "Hamburg" },
        { id: "GRA", name: "Graz" }
    ];

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

            ngModel.$formatters.push(function (value) {
                var fh = _.find(flughaefen, function(fh) { return fh.id == value; });

                if (fh) {
                    return fh.name;
                }
                else {
                    return value;
                }
            });

            ngModel.$parsers.unshift(function (viewValue) {
                var fh = _.find(flughaefen, function(fh) { return fh.name == viewValue; });

                if (fh) {
                    ngModel.$setValidity('flughafen', true);
                    return fh.id;
                }
                else {
                    ngModel.$setValidity('flughafen', false);
                    return undefined;
                }
            });
```

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

Zum Testen der betrachteten Direktive kann der Controller in Beispiel 4-28 sowie das Markup in Beispiel 4-29 genutzt werden. Der Controller in Beispiel 4-28 definiert ein Model, das einen Flug repräsentiert. Seine Eigenschaften *von* und *nach* beinhalten lediglich die IDs von Flughäfen.

Beispiel 4-28: Controller zum Testen des Formatters und des dazugehörigen Parsers

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

app.controller("editFlugCtrl", function ($scope) {
    $scope.vm = {};

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

Das Markup in Beispiel 4-29 bindet die Eigenschaft des Flugs an ein Eingabefeld. Dieses erhält neben dem Validierungsattribut *required* auch das Attribut *flughafen*, welches die zuvor beschriebene Direktive aktiviert. Falls die Eigenschaft *form.von.$error.flughafen* den Wert *true* erhält, zeigt es eine Fehlermeldung an. Wie gewohnt, handelt es sich bei *form* um den Namen des Formulars und somit um den Namen des von AngularJS dafür eingerichteten Form-Controllers. Den Model-Controller, welcher den darzustellenden Wert repräsentiert, repräsentiert die Eigenschaft *von*. Die Eigenschaft *flughafen* des *$error*-Objekts ergibt sich aus dem Regelnamen, den der Parser an *$setValidity* übergeben hat.

Beispiel 4-29: Markup zum Testen des Formatters und des dazugehörigen Parsers

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

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

            <div class="form-group">
                <label for="von">Von</label>
                <div><input class="form-control" ng-model="vm.flug.von"
                            required flughafen name="von" id="von" /></div>
                <div ng-show="form.von.$error.required">Pflichtfeld!</div>
                <div ng-show="form.von.$error.flughafen">Kein gültiger Flughafen!</div>
            </div>
```

```
    </form>
</div>
```

AngularJS verlangt, dass sich der Entwickler an die in HTML sowie JavaScript vorherrschenden Konventionen hält. Da diese voneinander abweichen, kann es bei Namen von Direktiven zu unterschiedlichen Schreibweisen kommen, denn während JavaScript bei zusammengesetzten Hauptwörtern auf Camel-Case setzt, bedient sich HTML der Bindestriche. Aus diesem Grund müsste der Entwickler eine Direktive, die er in JavaScript *myDirective* nennt, unter HTML als *my-directive* referenzieren. Im hier betrachteten Beispiel fällt dies nicht ins Gewicht, da der unter JavaScript verwendete Name lediglich aus Kleinbuchstaben besteht und somit ohne Änderungen auch in HTML zu verwenden ist.

Validatoren

Bis Version 1.2 mussten Entwickler Parser bereitstellen, um eigene Validierungsregeln zu implementieren. Seit Version 1.3 gibt es parallel dazu ein eigenes Validatorkonzept. Da sich dieses Konzept auf das Validieren von Eingaben beschränkt, ist es etwas einfacher zu nutzen. Darüber hinaus bietet dieses neue Konzept auch die Möglichkeit, Validierungsregeln asynchron zu prüfen. Das ist vor allem dann nützlich, wenn hierzu ein HTTP-basierter Service anzustoßen ist. Dieser Abschnitt geht zunächst auf einfache Validatoren, welche ihre Arbeit synchron verrichten, ein und wendet sich danach ihren asynchronen Gegenstücken zu.

Synchrone Validatoren

Wie bei Formatter und Parser (siehe Abschnitt »Formatter und Parser«) registriert der Entwickler auch Validatoren bei *ng-model* über eine benutzerdefinierte Direktive (Beispiel 4-30). Hierzu nutzt die Direktive die von *ngModel* angebotene Eigenschaft *$validators*. Hinter dieser Eigenschaft steht ein Objekt, dessen Eigenschaften auf Validatoren verweisen. Die Namen dieser Eigenschaften sind die Namen der Validierungsregeln. Bei den Validatoren handelt es sich um Funktionen, die den zu validierenden Wert entgegennehmen und einen Boolean, der über das Ergebnis der Validierung informiert, retournieren. Der Wert *true* zeigt hierbei an, dass die Validierung erfolgreich verlaufen ist; *false* lässt hingegen auf einen Verstoß gegen die jeweilige Validierungsregel schließen. Das in Beispiel 4-30 gezeigte Beispiel demonstriert dies anhand eines Validators, der eine Flugnummer unter Verwendung eines regulären Ausdrucks validiert.

Beispiel 4-30: Benutzerdefinierter Validator

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

    var pattern = /^\w+\d{3,}$/;
```

```
    return {
        require: 'ngModel',
        link: function (scope, element, attrs, ngModel) {

            ngModel.$validators.flugnr = function(value) {

                var ok = value.match(pattern);
                if(ok) return true;
                return false;
            }

        }
    }

});
```

Das Listing in Beispiel 4-31 zeigt, wie der Entwickler einen solchen Validator anwenden kann. Dazu spendiert es dem zu validierenden Eingabefeld ein Attribut *flugnr-validator*, welches die zuvor betrachtete Direktive aktiviert. Dabei fällt auf, dass die Schreibweise zwischen dem Direktivennamen und dem Attributnamen ein wenig abweicht. Während die Direktive den Namen *flugnrValidator* hat, nennt sich das Attribut *flugnr-validator*. Hierbei berücksichtigt AngularJS die üblichen Namenskonventionen in JavaScript und HTML: In JavaScript kommt Camel-Case zum Einsatz und HTML-Bezeichner sind in Kleinbuchstaben gehalten und verwenden ein Minus als Worttrennzeichen. Um zu ermitteln, ob eine Fehlermeldung anzuzeigen ist, prüft das betrachtete Beispiel die Eigenschaft *form.flugNummer.$error.flugnr*, wobei *flugnr* dem Namen des Validators und somit dem Namen der hierdurch ausgedrückten Validierungsregel wiederspiegelt.

Beispiel 4-31: Einsatz eines benutzerdefinierten Validators

```
<div ng-app="flug">
    <form ng-controller="editFlugCtrl" name="form" id="form">

        <div class="form-group">
            <label for="fnr">Flug-Nummer</label>
            <div><input class="form-control"
                        ng-model="vm.flug.flugNummer"
                        required
                        flugnr-validator
                        name="flugNummer"
                        id="flugNummer" /></div>
            <div ng-show="form.flugNummer.$error.required">Pflichtfeld!</div>
            <div ng-show="form.flugNummer.$error.flugnr">Keine gültige Flugnummer!</div>
        </div>

    </form>
</div>
```

Registriert der Entwickler für eine Instanz von *ng-model* sowohl Parser als auch Validatoren, so führt AngularJS zunächst sämtliche Parser aus und wendet sich erst danach den Validatoren zu.

Asynchrone Validatoren

Asynchrone Validatoren, welche eine Validierungsregel im Hintergrund prüfen, verwaltet *ng-model* über die Eigenschaft *$asyncValidators*. Sie kommen nur zum Einsatz, wenn weder Parser noch synchrone Validatoren einen Validierungsfehler gemeldet haben. Wie bei ihren synchronen Gegenstücken handelt es sich bei asynchronen Validatoren auch lediglich um Funktionen, denen AngularJS den zu validierenden Wert übergibt. Im Gegensatz zu synchronen Validatoren liefern asynchrone Validatoren jedoch ein *Promise* zurück. Dabei handelt es sich um ein Objekt, welches eine asynchrone Operation repräsentiert und Callbacks anstößt, die über das Ergebnis dieser asynchronen Operation informieren.

Beispiel 4-32 zeigt eine Direktive, welche einen asynchronen Validator bereitstellt. Sie ähnelt den Direktiven in den vorangegangenen Abschnitten, weswegen das betrachtete Beispiel hier auch nur auf die Unterschiede dazu eingeht. Der eingerichtete Validator ruft einen HTTP-Service auf, um den übergebenen Wert zu prüfen. Dazu nutzt er die Methode *get* des *$http*-Services. Diese Funktion liefert ein Promise retour. Unter Verwendung von dessen *then*-Funktion registriert der Validator einen Handler, den AngularJS anstoßen soll, wenn die angeforderten Daten vorliegen. Das Ergebnis von *then* ist ein weiteres Promise, welches der Validator retourniert. Der Handler nimmt, wie gewohnt, die vom HTTP-Service erhaltenen Daten entgegen. Handelt es sich dabei um einen Wert, den JavaScript als *false* ausgewertet, liefert er einen Promise zurück, der über einen negativen Ausgang einer asynchronen Operation informiert. Diesen Promise erzeugt er mit dem von AngularJS bereitgestellten Service *$q*. Liefert der Callback keinen eigenen Promise retour, geht AngularJS davon aus, dass die asynchrone Validierungsoperation erfolgreich verlaufen ist.

Kapitel 8 geht auf Promises sowie auf den Service *$q* im Detail ein.

Beispiel 4-32: Beispiel für asynchronen Validator

```
app.directive('flughafenAsyncValidator', function($http, $q, $log) {

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

            ngModel.$asyncValidators.flughafen = function(value) {

                $log.log("validate flughafen: " + value);
                var urlParams = { name: value, delay: 0 };
```

```
                var url = "http://flugdemo.azurewebsites.net/api/Flughafen";

                return $http.get(url, { params: urlParams }).then(function(response) {

                    $log.log("got data from service: " + response.data);
                    if (!response.data) {
                        return $q.reject();
                    }
                });
            }
        }
    }
});
```

Wie auch synchrone Validatoren registriert der Entwickler ihre aynchronen Gegenstücke, indem er die dafür bereitgestellte Direktive für ein Eingabefeld aktiviert (Beispiel 4-33). Der Model-Controller, welcher dieses Eingabefeld repräsentiert, gibt über die Eigenschaft *$pending* darüber Auskunft, ob er noch auf das Ergebnis von asynchronen Validatoren wartet. Unter Verwendung dieser Eigenschaft kann der Entwickler den Benutzer darüber informieren, dass die Validierung noch nicht abgeschlossen ist.

Beispiel 4-33: Verwenden asynchroner Validatoren

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

    <form ng-controller="editFlugCtrl" name="form" id="form">
        <div class="form-group">
            <label for="von">Von</label>
            <div><input class="form-control" ng-model="vm.flug.von"
            required flughafen-async-validator name="von" id="von" /></div>
            <div ng-show="form.von.$pending">Validierung wird durchgeführt!</div>
            <div ng-show="form.von.$error.required">Pflichtfeld!</div>
            <div ng-show="form.von.$error.flughafen">Kein gültiger Flughafen!</div>
        </div>
    </form>

</div>
```

Zusammenfassung

Neben der Möglichkeit, Models an Formularfelder zu binden, bietet AngularJS auch Unterstützung bei der damit einhergehenden Validierung. Dazu stellt es einige Validierungsregeln zur Verfügung. Kommt der Entwickler damit nicht aus, hat er die Möglichkeit, eigene Validierungsregeln zu definieren. Seit Version 1.3 kann AngularJS diese auch asynchron ausführen. Daneben bietet AngularJS mit Formattern und Parsern Einsprungpunkte für das Aufbereiten von Daten für das Formular sowie für das Rückschreiben der erfassten Daten ins Model.