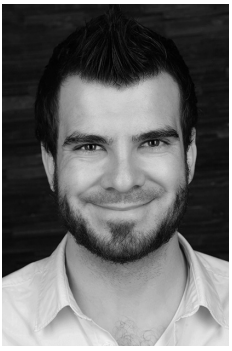
Philipp Tarasiewicz / Robin Böhm, AngularJS, dpunkt.verlag, ISBN 978-3-86490-154-6

Über die Autoren

Philipp Tarasiewicz ist im Web groß geworden und arbeitet als freiberuflicher Technologieberater, Autor, Sprecher und Coach. Seit einigen Jahren hat er sich auf den Bereich Enterprise JavaScript, insbesondere AngularJS, spezialisiert und unterstützt Unternehmen bei der Aus- und Fortbildung ihrer Mitarbeiter wie auch beim Ramp-up neuer Projekte. Gemeinsam mit Sascha Brink und Robin Böhm betreibt er das deutsche Portal zu AngularJS (AngularJS.DE).



Robin Böhm ist leidenschaftlicher Softwareentwickler, Berater und Autor im Bereich der Webtechnologien und speziell zu Enterprise JavaScript. Er beschäftigt sich seit einigen Jahren intensiv mit der Erstellung clientseitiger Webapplikationen und unterstützt Unternehmen sowohl bei der Aus- und Fortbildung von Mitarbeitern als auch bei der Umsetzung von Projekten. Außerdem ist er Mitgründer des Portals AngularJS.DE.

Philipp Tarasiewicz · Robin Böhm

AngularJS

Eine praktische Einführung in das JavaScript-Framework

dpunkt.verlag

Philipp Tarasiewicz
philipp.tarasiewicz@angularjs.de

Robin Böhm
robin.boehm@angularjs.de

Lektorat: René Schönfeldt
Copy Editing: Annette Schwarz, Ditzingen
Satz: Da-TeX, Leipzig
Herstellung: Frank Heidt
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN 978-3-86490-154-6

1. Auflage 2014

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

1 AngularJS Schnellstart

Dieses Kapitel soll in Form von kurzen Beispielen die mächtigsten Features von AngularJS demonstrieren. Außerdem wollen wir ein erstes Gefühl dafür vermitteln, wie typische Quellcodefragmente in einer AngularJS-Anwendung aussehen. Die Beispiele sind absichtlich sehr einfach gehalten, um sie ohne tiefe Kenntnisse des Frameworks verstehen und ausführen zu können. Wir werden nicht alle Einzelheiten erklären, sondern die Erklärungen mit Absicht recht kurz halten. Wenn Sie das Buch durchgearbeitet haben, werden Sie die Beispiele in ihren Einzelheiten verstehen.

1.1 Zwei-Wege-Datenbindung: Boilerplate-Code war gestern

Eines der ausschlaggebenden Features, die für AngularJS sprechen, ist sicherlich die *Zwei-Wege-Datenbindung* (Two-Way Data-Binding). Die Zwei-Wege-Datenbindung sorgt dafür, dass sich Änderungen im Datenmodell automatisch auf die entsprechenden Elemente in der Ansicht auswirken und sich Benutzerinteraktionen innerhalb der Ansicht auch automatisch in dem Datenmodell widerspiegeln. Durch diesen automatischen Abgleich in beide Richtungen sprechen wir von einer Zwei-Wege-Datenbindung. Somit können wir uns sehr viel »Klebe-Code« (Boilerplate-Code) sparen, den wir sonst schreiben müssten, um die äquivalente Synchronisierungslogik zu implementieren. Das folgende Beispiel zeigt das Feature in Aktion.

Listing 1-1
Zwei-Wege-Datenbindung mit einem Eingabefeld

```
<!DOCTYPE html>
<html ng-app>
<head>
    <meta charset="utf-8">
</head>
<body>
```

```
<div>
    <label>Name:</label>
    <input type="text" ng-model="yourName">
    <hr>
    <h1>Hello {{yourName}}!</h1>
</div>

<script src="lib/angular/angular.js">
</script>

</body>
</html>
```

Ohne auf die Einzelheiten detailliert einzugehen, sehen wir, dass Listing 1-1 eine einfache HTML-Datei enthält, die ein `input`-Eingabefeld beinhaltet und eine Ausgabe in Form einer `h1`-Überschrift definiert. Wenn wir dieses Beispiel nun in einem Browser ausführen (siehe Abbildung 1-1), dann stellen wir fest, dass sich die Überschrift jedes Mal automatisch aktualisiert, wenn wir den Text in dem Eingabefeld ändern. Der Mechanismus, der das für uns übernimmt, ist also die Zwei-Wege-Datenbindung.

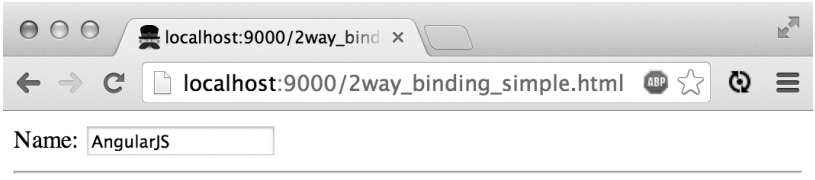


Abb. 1-1
Ausgabe des Beispiels aus Listing 1-1 in Chrome

ng-model ist eine Direktive.

Für die Herstellung der Zwei-Wege-Datenbindung ist in diesem Beispiel das Attribut `ng-model` verantwortlich. Oder anders erklärt: Das, was für HTML-Kenner aussieht wie ein unbekanntes HTML-Attribut, ist in der Welt von AngularJS eine sogenannte *Direktive*. Die *ngModel*-Direktive stellt eine Zwei-Wege-Datenbindung zwischen dem Eingabefeld und der Variable `yourName` her. Damit ändert sich der Wert dieser Variable automatisch entsprechend der Eingabe im Eingabefeld. Wo die Variable `yourName` definiert ist, ist zunächst einmal unerheblich.[1]

Zum Verständnis dieses Beispiels fehlt jetzt noch die Erklärung zu dem Ausdruck in den doppelten geschweiften Klammern innerhalb des

[1] Für Ungeduldige: AngularJS nutzt das *ViewModel*-Konzept zur Realisierung der Zwei-Wege-Datenbindung. Dabei tragen die sogenannten ViewModels innerhalb von AngularJS den Namen *Scopes*. Die Variable `yourName` ist also innerhalb eines solchen Scopes definiert.

Expressions

`h1`-Tags. Dabei handelt es sich um eine sogenannte *Expression*. Das ist der AngularJS-Mechanismus, mit dem wir Ausgaben produzieren können. Insbesondere erlauben uns Expressions, Variablenwerte auszugeben. Sie unterliegen dabei ebenfalls der Zwei-Wege-Datenbindung. Wenn sich also der Wert der Variable `yourName` verändert, dann wird die Expression neu ausgewertet, was schließlich dazu führt, dass sich die Ansicht automatisch aktualisiert. Auf diese Weise überträgt das Framework den Inhalt des Eingabefeldes automatisch in die Überschrift.

ng-app definiert eine AngularJS-Anwendung.

Wir sollten an dieser Stelle noch die `ng-app`-Direktive erwähnen, die in diesem Beispiel das `html`-Tag annotiert. Mithilfe dieser Direktive teilen wir AngularJS mit, welchen Teil des *DOM* (Document Object Model) das Framework als AngularJS-Anwendung betrachten soll. Dadurch dass wir die Direktive hier an das `html`-Tag schreiben, teilen wir AngularJS also mit, dass unsere Anwendung auf dem kompletten DOM operieren soll, weil das `html`-Tag unser Wurzelknoten ist. Wir könnten die `ng-app`-Direktive aber auch an einen tiefer verschachtelten DOM-Knoten schreiben und nur für diesen entsprechenden Unterbaum des DOM eine AngularJS-Anwendung ausweisen. Somit würde das Framework alle Ausdrücke außerhalb dieses Unterbaums unangetastet lassen und ignorieren.

Mehrere AngularJS-Anwendungen in einem HTML-Dokument

Wir können auch mehrere Geschwister-Knoten des DOM mit der `ng-app`-Direktive annotieren, um in einem HTML-Dokument mehrere autonome AngularJS-Anwendungen zu definieren. Für den Großteil der Applikationen hat diese Verwendung wenig Sinn, weil die einzelnen Anwendungen zunächst einmal keine Möglichkeiten besitzen, um miteinander zu interagieren. Dennoch sollten wir diesen Aspekt hier erwähnen, weil die offizielle Webseite zu AngularJS[2] diese Eigenschaft ausnutzt, um eine Vielzahl von in sich geschlossenen Beispielen zu präsentieren.

Ein zweites Beispiel

Durch das Feature der Zwei-Wege-Datenbindung lassen sich bereits viele Anwendungsfälle elegant umsetzen, ohne eine einzige Zeile JavaScript-Quellcode schreiben zu müssen. Um das Ganze mit einem weiteren Beispiel zu belegen, schauen wir uns den nachfolgenden *Color-Picker* an. Ein Color-Picker ist eine UI-Komponente, die dem Benutzer die Auswahl einer Farbe erleichtert, indem sie ihm ein sofortiges visuelles Feedback bezüglich der aktuellen Belegung der RGBA-Werte liefert.

[2] http://angularjs.org/

Listing 1-2 *Zwei-Wege-Datenbindung mit HTML5-Schiebereglern*

```
<!DOCTYPE html>
<html ng-app>
<head>
    <meta charset="utf-8">
</head>
<body ng-init="r=255; g=0; b=123; a=0.7">

R:<input type="range" name="color_r"
        min="0" max="255" step="1" ng-model="r"><br>
G:<input type="range" name="color_g"
        min="0" max="255" step="1" ng-model="g"><br>
B:<input type="range" name="color_b"
        min="0" max="255" step="1" ng-model="b"><br>
A:<input type="range" name="color_a"
        min="0" max="1" step="0.01" ng-model="a">

<div style="width: 300px; height: 100px;
           background-color:
           rgba({{ r }}, {{ g }}, {{ b }}, {{ a }});">
</div>

<script src="lib/angular/angular.js"></script>

</body>
</html>
```

Eine einfache Umsetzung solch eines Color-Pickers sehen wir in Listing 1-2. Vom Grundaufbau entspricht das Beispiel dem Beispiel davor. Wir definieren vier *HTML5-Schieberegler*, indem wir innerhalb des `input`-Tags diesmal dem `type`-Attribut den Wert `range` zuweisen. Drei Schieberegler brauchen wir für die Steuerung der Rot-, Grün- und Blau-Werte unseres Color-Pickers und einen weiteren für die Steuerung des Alpha-Kanals. Mit den Attributen `min` und `max` können wir den Minimum- und Maximumwert des Reglers festlegen. Das `step`-Attribut gibt die Schrittweite an.

Auch in diesem Beispiel nutzen wir die *ngModel*-Direktive, um zwischen den Schiebereglern und den entsprechenden Scope-Variablen `r`, `g`, `b` und `a` eine Zwei-Wege-Datenbindung herzustellen. Weiterhin können wir erkennen, dass auch Expressions zum Einsatz kommen. Diesmal produzieren wir mit den Expressions aber keine direkte Ausgabe, sondern setzen damit die einzelnen Komponenten der `background-color`-CSS-Eigenschaft des `div`-Elements, in dem die Farbvorschau gerendert werden soll.

Das Ergebnis ist recht beeindruckend. Dadurch dass wir zwischen den vier Reglern und den vier Einzelkomponenten der `rgba`-Eigenschaft

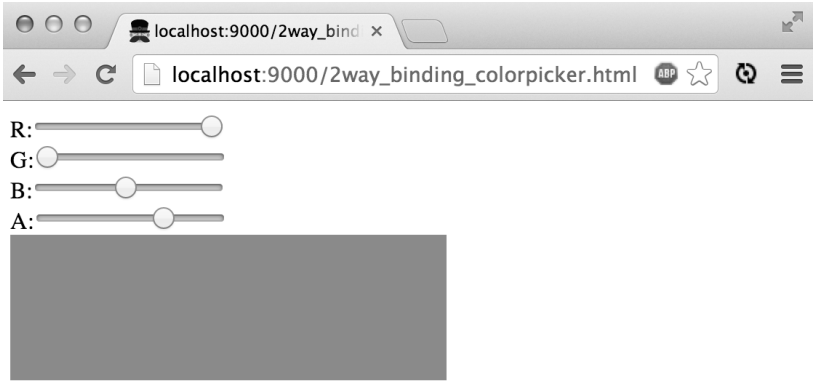


Abb. 1-2
Ausgabe des Color-Picker-Beispiels aus Listing 1-2 in Chrome

auf Basis der Scope-Variablen `r`, `g`, `b` und `a` eine Zwei-Wege-Datenbindung geschaffen haben, haben wir mit übersichtlichem Aufwand einen simplen Color-Picker mit Live-Farbvorschau erstellt (siehe Abbildung 1-2). Jedes Mal, wenn wir einen der vier Regler verschieben, ändert sich entsprechend die Farbe in unserem `div`-Element für die Farbvorschau.

ng-init zur Initialisierung von Scope-Variablen in einem Template

Im Vergleich zu dem Beispiel davor gibt es hier allerdings noch eine kleine Erweiterung. Wir nutzen die *ngInit*-Direktive, um die Variablen `r`, `g`, `b` und `a` mit Initialwerten zu belegen.

Hinweis zur ngInit-Direktive

Die Definition von Initialwerten innerhalb eines Templates mithilfe von `ng-init` gehört in größeren Projekten zwar nicht zum guten Ton, weil wir damit einen Teil der Logik ins Template verlagern. Für unsere Zwecke können wir diese Variante aber nutzen, um in dem einfachen Beispiel das Schreiben des äquivalenten JavaScript-Codes zu vermeiden.

1.2 Direktiven: Eigene HTML-Elemente und Attribute

Ein weiteres mächtiges Feature und außerdem eines der Alleinstellungsmerkmale von AngularJS sind *Direktiven*. Wie wir bereits in den beiden vorangegangenen Beispielen gesehen haben, sind Direktiven allgegenwärtig in AngularJS. Das Framework selber baut sehr stark auf diesem Konzept auf.

Mithilfe von Direktiven erweitern wir HTML um eigene Elemente und Attribute. Diesen Mechanismus können wir für viele interessante Dinge verwerten. Wir können uns z. B. für unsere Anwendung eine

eigene Tag-Sammlung definieren, mit der wir große Teile der Anwendung deklarativ beschreiben können. Oder wir könnten eine eigene wiederverwendbare Bibliothek schreiben, die HTML um eine Vielzahl spezieller UI-Komponenten erweitert, die in der Domäne unserer Applikation unverzichtbar sind. Denkbar sind Tags wie `<chart>`, `<spreadsheet>` oder auch `<input auto-complete="data">`. Die entsprechende Logik würden wir dann innerhalb der Direktiven kapseln.

Ein Beispiel

In dem nachfolgenden Beispiel erfinden wir ein `<color-picker>`-Tag, das die Logik aus dem Color-Picker-Beispiel kapselt und somit wiederverwendbar macht. Dabei haben wir uns dafür entschieden, dass unsere Direktive zwei Anforderungen erfüllen muss. Es soll eine Möglichkeit geben, die initiale Farbe des Color-Pickers zu konfigurieren. Außerdem wollen wir bei Farbänderungen von der Direktive benachrichtigt werden, um darauf in irgendeiner Form reagieren zu können. Das Template in Listing 1-3 zeigt, wie wir unsere eigene `<color-picker>`-Direktive den Anforderungen entsprechend nutzen wollen.

Listing 1-3 *Verwendung der colorPicker-Direktive*

```
<!DOCTYPE html>
<html ng-app="colorPickerApp">
<head>
    <meta charset="utf-8">
</head>

<body ng-controller="MainCtrl">

<h1>AngularJS Schnellstart ColorPicker</h1>
<color-picker init-r="255"
              init-g="0"
              init-b="0"
              init-a="1.0"
              on-change="onColorChange(r,g,b,a)">
</color-picker>

<script src="lib/angular/angular.js"></script>
<script src="scripts/colorPickerApp.js"></script>
</body>
</html>
```

Unsere Direktive wird genau wie die Standard-HTML-Tags benutzt, indem wir die gewöhnliche Tag-Syntax verwenden und einige selbstdefinierte Attribute setzen. Im Einzelnen sind das die Attribute `init-r`, `init-g`, `init-b`, `init-a` und `on-change`. Entsprechend unserer Anforderung nutzen wir die `init`-Attribute, um den initialen RGB- und

Alpha-Wert des Color-Pickers zu konfigurieren. Mithilfe des `on-change`-Attributs spezifizieren wir eine Callback-Funktion[3], die die Direktive im Falle eine Farbänderung aufrufen soll. Somit haben wir von außen die Möglichkeit, auf eine Farbveränderung zu reagieren. Wir teilen der Direktive also mit, dass `onColorChange()` die Callback-Funktion ist, die die Direktive aufrufen soll. Definiert ist diese Funktion in dem `MainCtrl`-Controller. AngularJS weiß, dass dieser Controller in dem DOM-Unterbaum des `<body>`-Tags gelten soll, weil wir dieses Tag mit der `ng-controller`-Direktive annotiert haben. Was das genau heißt, soll uns an dieser Stelle zunächst nicht weiter interessieren.

In dem Template in Listing 1-3 nutzen wir außerdem die *ngApp*-Direktive anders als in den Beispielen davor, nämlich mit der Angabe eines zu ladenden Moduls. Wird – wie in den Beispielen davor – kein Modul angegeben, lädt AngularJS für uns ein Default-Modul. Für kleine Beispiele ist das vollkommen ausreichend, aber in größeren Anwendungen mit mehreren Modulen funktioniert diese Methode nicht, weil dem Framework nicht klar ist, welches Modul geladen werden soll, nachdem der Browser das DOM vollständig geladen hat. Wir wollen uns ab sofort an Best Practices halten und teilen AngularJS in Listing 1-3 mit, dass das Framework das `colorPickerApp`-Modul laden soll, sobald der Browser das DOM geladen hat. Entsprechend müssen wir dieses Modul und den zuvor genannten `MainCtrl`-Controller in unserem JavaScript-Code definieren.

```
var colorPickerApp = angular.module('colorPickerApp', []);

colorPickerApp.controller('MainCtrl', function ($scope) {
    $scope.onColorChange = function(r,g,b,a) {
        console.log('onColorChange', r, g, b, a);
    };
});
```

Listing 1-4
Definition des colorPickerApp-Moduls inkl. MainCtrl-Controller

Scope

Interessant an dieser Definition ist die Tatsache, dass AngularJS dem `MainCtrl`-Controller einen Scope übergibt. Den Begriff des Scopes (Geltungsbereich) haben wir in dem ersten Beispiel bereits eingeführt. Ein Scope definiert alle Variablen und Funktionen, die in einem bestimmten Kontext gültig sein sollen, und ist weiterhin maßgeblich dafür verantwortlich, dass in diesem Kontext die Zwei-Wege-Datenbindung genutzt

[3] Eine Callback-Funktion ist eine Funktion, die eine Komponente A einer Komponente B übergibt, damit A auf bestimmte Ereignisse innerhalb von B reagieren kann. Dabei nimmt B die Callback-Funktion entgegen und ruft sie beim Auftreten bestimmter Ereignisse auf, um A darüber zu informieren. Bildlich gesprochen ruft die Komponente B die Komponente A im Falle eines bestimmten Ereignisses zurück (deswegen »Callback«).

werden kann. In dem Framework gibt es einige Komponenten, für die im Falle eines Aufrufs ein neuer Scope erzeugt wird. Ein Controller ist eine dieser Komponenten. Somit übergibt AngularJS dem Controller seinen Scope. Auf diesen Scope können wir innerhalb des Controllers mittels des übergebenen `$scope`-Parameters zugreifen. Hiermit definieren wir in dem Scope des `MainCtrl`-Controllers die angesprochene Callback-Funktion `onColorChange()`, die die vier Parameter `r`, `g`, `b` und `a` erwartet und von unserer `colorPicker`-Direktive bei Farbveränderungen aufgerufen wird. Bei einer Farbveränderung wollen wir also lediglich einen Text in der Konsole des Browsers ausgeben.

Das Direktiven-Template

Definition der colorPicker-Direktive

Nun definieren wir die eigentliche `colorPicker`-Direktive. Dazu fangen wir mit dem Template an, das sich hinter dem `<color-picker>`-Tag verbergen soll.

Listing 1-5 *Definition des Templates der colorPicker-Direktive*

```
R:<input type="range"
        min="0" max="255" step="1"
        ng-model="r"><br>
G:<input type="range"
        min="0" max="255" step="1"
        ng-model="g"><br>
B:<input type="range"
        min="0" max="255" step="1"
        ng-model="b"><br>
A:<input type="range"
        min="0" max="1" step="0.01"
        ng-model="a">

<div style="width: 300px; height: 100px;
           background-color:
           rgba({{ r }}, {{ g }}, {{ b }}, {{ a }});">
</div>
```

Wie wir in Listing 1-5 unschwer erkennen können, ist das Template identisch mit dem Template aus dem Color-Picker-Beispiel, bei dem wir keine eigene Direktive definiert haben (siehe Listing 1-2).

Die Direktivendefinition

Kommen wir nun zu dem interessanten Teil des Beispiels, der eigentlichen Direktivendefinition.

Listing 1-6
Definition der colorPicker-Direktive

```
colorPickerApp.directive('colorPicker', function () {
    return {
        scope: {
            r: '@initR',
            g: '@initG',
            b: '@initB',
            a: '@initA',
            onChange: '&'
        },
        restrict: 'E',
        templateUrl: 'colorPickerTemplate.html',
        link: function(scope) {
            var COLORS = ['r', 'g', 'b', 'a'];

            COLORS.forEach(function(value) {
                scope.$watch(value, function(newValue, oldValue) {
                    if (newValue !== oldValue) {
                        if (angular.isFunction(scope.onChange)) {
                          scope.onChange(generateColorChangeObject());
                        }
                    }
                });
            });

            var generateColorChangeObject = function() {
                var obj = {};

                COLORS.forEach(function(value) {
                    obj[value] = scope[value];
                });

                return obj;
            };
        }
    };
});
```

Wir wollen die Direktivendefinition in Listing 1-6 nicht im Detail besprechen, sondern nur auf einige Kernaspekte eingehen, um ein Gefühl dafür zu vermitteln, wie eigene Direktiven implementiert werden können.

Namenskonventionen bei Direktiven

Wir führen eine neue Direktive mit der `directive()`-Funktion von AngularJS ein. Der erste Parameter gibt dabei den Namen unserer Direktive an. Dieser Name ist unmittelbar dafür verantwortlich, wie das entsprechende HTML-Tag bzw. HTML-Attribut heißen wird, das die Direktive in unseren Templates repräsentiert. Dabei lässt sich bereits

in diesem Beispiel eine Besonderheit erkennen. Das HTML-Tag unseres Color-Pickers lautet `<color-picker>` und der Direktivenname ist `colorPicker`. Für die Abbildung von dem Direktivennamen zum HTML-Element bzw. HTML-Attribut verwendet AngularJS eine interne Regel.

Regel zur Benennung von Direktiven

Wenn der Name einer Direktive aus mehreren Wörtern besteht, geben wir ihn mit der *CamelCase*-Notation an. Die Abbildung zum entsprechenden HTML-Tag bzw. HTML-Attribut erfolgt, indem statt der CamelCase-Notation die *SnakeCase*-Notation verwendet wird. Als Trennzeichen sind dabei der Doppelpunkt (»:«), Bindestrich (»-«) und Unterstrich (»_«) erlaubt. Folglich könnten wir die `colorPicker`-Direktive auch mittels `<color:picker>` und `<color_picker>` benutzen. Für gewöhnlich benutzen wir allerdings den Bindestrich als Trennzeichen.

Außerdem gibt es zwei weitere valide Möglichkeiten, AngularJS-Direktiven aufzurufen. Und zwar einerseits mit dem Präfix `data-` und andererseits mit dem Präfix `x-`. Die Intention dahinter ist, dass ältere Browser selbstdefinierte Tags bzw. Attribute nur dann zulassen, wenn sie eines dieser Präfixe besitzen. Demnach könnten wir unseren Color-Picker auch mithilfe von `<data-color-picker>` und `<x-color-picker>` aufrufen. Es ist wichtig, diese Namenskonvention zu kennen, um bei der Erstellung eigener Direktiven keine bösen Überraschungen zu erleben.

Direktiven-Definitions-Objekt (DDO)

Als zweiten Parameter erwartet die `directive()`-Funktion eine Funktion, die entweder eine Funktion oder das sogenannte *Direktiven-Definitions-Objekt* (DDO) mit bestimmten Eigenschaften zurückgibt. Der Unterschied zwischen diesen beiden Rückgaben ist an dieser Stelle zunächst einmal unerheblich. In den meisten Fällen definieren wir Direktiven so, dass wir ein DDO zurückgeben, welches das Verhalten der Direktive anhand bestimmter Objekteigenschaften beschreibt. Auch in diesem Beispiel definieren wir unsere `colorPicker`-Direktive mithilfe eines solchen Objekts. Dabei gibt es eine Reihe von Eigenschaften, die wir in diesem Konfigurationsobjekt setzen können, um ein bestimmtes Verhalten zu erreichen. Zu diesen Eigenschaften gehören u. a. die hier verwendeten Eigenschaften `scope`, `restrict`, `templateUrl` und `link`.

Die templateUrl-Eigenschaft

Mit der `templateUrl`-Eigenschaft können wir eine URL zu dem Direktiven-Template angeben. Wir referenzieren also unser zuvor definiertes Template aus der Datei `colorPickerTemplate.html`.

Die restrict-Eigenschaft

Die `restrict`-Eigenschaft erwartet eine Zeichenkette, die die Buchstaben `E`, `A`, `C` oder `M` enthält. Damit geben wir an, auf welche HTML-Fragmente sich die Direktive auswirken soll. Die Bedeutung der Buchstaben ist dabei:

- `E` – HTML-Element bzw. HTML-Tag
- `A` – HTML-Attribut
- `C` – CSS-Klasse
- `M` – HTML-Kommentar

Auch eine Kombination dieser vier Buchstaben ist möglich. Also würde z. B. die Zeichenkette `EA` dazu führen, dass eine Direktive nur auf HTML-Tags und HTML-Attribute wirkt und in CSS-Klassen und HTML-Kommentaren ignoriert wird. An dieser Stelle sollten wir auch erwähnen, dass wir Direktiven nicht nur dazu nutzen können, um eigene Tags und Attribute zu definieren, sondern offensichtlich auch dazu, um HTML mithilfe von CSS-Klassen und Kommentaren neue Tricks beizubringen.

Machen wir weiter mit den beiden vermutlich interessantesten Eigenschaften: `scope` und `link`.

Die scope-Eigenschaft

Mithilfe der `scope`-Eigenschaft können wir spezifizieren, ob die Direktiveninstanzen bei der Verwendung einen eigenen Scope erhalten sollen. Außerdem gibt es im Kontext von Direktiven noch eine besondere Ausprägung von Scopes: die *isolierten Scopes*, die durch ihre völlige Isolation dafür sorgen, dass sich bestimmte Scope-Variablen mehrerer Direktiveninstanzen nicht versehentlich gegenseitig überschreiben und zu schwer identifizierbaren Fehlern führen. Dadurch dass wir in unserer `colorPicker`-Direktive die `scope`-Eigenschaft mit einem Objekt belegen, spezifizieren wir für diese Direktive solch einen isolierten Scope. Isolierte Scopes haben aber trotz ihrer Isolation die Möglichkeit, mit ihrer Umgebung zu kommunizieren. Wir wollen darauf in diesem Fall nicht näher eingehen, sondern nur die in diesem Beispiel relevanten Aspekte erläutern. Wir können unschwer erkennen, dass es eine Verbindung gibt zwischen dem Objekt, mit dem die `scope`-Eigenschaft belegt ist, und unseren selbst erfundenen HTML-Attributen `init-r`, `init-g`, `init-b` und `init-a` des `<color-picker>`-Tags. Diese Verbindung sorgt dafür, dass wir auf die übergebenen Attributwerte innerhalb unserer Direktive zugreifen können.

Isolierte Scopes

Die link-Eigenschaft

Link-Funktion einer Direktive

Die Kerneigenschaft unserer Direktive definieren wir mit `link`. Dabei handelt es sich um die sogenannte *Link-Funktion*. Vereinfacht ausgedrückt führt AngularJS die Link-Funktion für jede Instanz einer Direktive genau einmal aus und erlaubt uns somit, für jede Instanz eine Initialisierungslogik zu definieren. Typischerweise registrieren wir in der Link-Funktion *Event-Handler* oder nehmen – falls nötig – DOM-Manipulationen vor. Somit können wir in der Link-Funktion die Logik implementieren, die dafür sorgt, das die Callback-Funktion aufgerufen wird, wenn sich der Wert der einzelnen Schieberegler unseres Color-Pickers verändert und daraus eine neue Farbe resultiert. Tatsächlich ist es so, dass wir nicht auf die Veränderungen der Regler horchen, sondern die entsprechenden Scope-Variablen `r`, `g`, `b` und `a` mit sogenannten *Beobachtern* (Watcher) ausstatten. Dadurch dass zwischen den Reglern und den entsprechenden Scope-Variablen eine Zwei-Wege-Datenbindung existiert, werden die Werte der Variablen bei Verschiebung automatisch aktualisiert. Mit den Beobachtern beobachten wir die Werte der Scope-Variablen und rufen im Falle einer Veränderung die Callback-Funktion auf.

Watcher mit scope.$watch()

Auf diese Weise haben wir ein neues HTML-Tag `<color-picker>` erfunden, das wir in jedem Template unserer AngularJS-Anwendung wiederverwenden können.

1.3 Filter: Formatierte Ausgaben im Handumdrehen

Obwohl das *Templating* in AngularJS eine wichtige Rolle spielt, besitzt das Framework *keine* eigene *Template-Engine*. Der Grund dafür ist die Tatsache, dass gewöhnliches HTML in Verbindung mit Expressions und Direktiven mächtig genug ist, um die typischen Aufgaben von Template-Engines zu erledigen.

Darüber hinaus gibt es in dem Framework die sogenannten *Formatierungs- und Collection-Filter*. Mit ihnen lassen sich die Möglichkeiten in einem Template nochmals um Formatierungs- und Filterungsfunktionalitäten erweitern. An dieser Stelle wollen wir uns stellvertretend den `date`-Filter anschauen, der zur Klasse der Formatierungsfilter gehört. Mit diesem Filter können wir die Ausgaben von Datumswerten auf eine sehr bequeme Weise formatieren.

Der date-Filter

Listing 1-7
Nutzung des date-Filters

```
<!DOCTYPE html>
<html ng-app="dateApp">
<head>
    <meta charset="utf-8">
</head>
<body ng-controller="DateCtrl">

Today's date:
{{ now | date:'dd. MMMM yyyy' }}<br>
And the time is:
{{ now | date:'HH:mm:ss' }}

<script src="lib/angular/angular.js"></script>
<script src="scripts/dateApp.js"></script>

</body>
</html>
```

Listing 1-7 zeigt eine beispielhafte Benutzung des `date`-Filters. Wir können erkennen, dass der Filter – wie alle anderen Formatierungsfilter auch – innerhalb einer Expression zum Einsatz kommt. Die Syntax lehnt sich dabei an die Pipe-Syntax diverser Unix-Shells an. Mit der Pipe (»|«) teilen wir AngularJS also mit, dass eine evaluierte Expression noch von einem Filter verarbeitet werden soll, bevor sie ausgegeben wird. Die Angabe des entsprechenden Filters erfolgt unmittelbar nach der Pipe. Genauso wie es in allen gängigen Unix-Shells üblich ist, lassen sich auch mehrere Filter hintereinander schalten. Die Verarbeitung geschieht dabei von links nach rechts.

Was der `date`-Filter an dieser Stelle nun macht, ist ziemlich einleuchtend. Bevor der Wert der Scope-Variable `now` ausgegeben wird, versucht der `date`-Filter ihn noch anhand der Formatierungszeichenkette zu formatieren. Innerhalb dieser Zeichenkette lassen sich die bekannten Platzhalter für Datums- bzw. Zeitkomponenten nutzen. Damit der Filter allerdings funktioniert, muss die Scope-Variable `now` eine Voraussetzung erfüllen. Sie muss mit einem der folgenden Werte belegt sein:

- `Date`-Objekt
- Zeichenkette oder `Number`-Objekt, welche die Anzahl der Millisekunden seit dem 01.01.1970 angeben
- Zeichenkette, die ein gültiges Datum im ISO 8601-Format enthält, also z. B. ein Datum, das dem folgenden Muster folgt: yyyy-MM-ddTHH:mm:ss.SSSZ

Der Vollständigkeit halber folgt nun noch der Quellcode, der das `dateApp`-Modul und den `DateCtrl`-Controller definiert.

Listing 1-8 *Definition des dateApp-Moduls inkl. DateCtrl-Controller*

```
var dateApp = angular.module('dateApp', []);
dateApp.controller('DateCtrl', function ($scope, $timeout) {
  $scope.now = 'Loading...';

  var updateTime = function() {
    $timeout(function() {
      $scope.now = new Date();
      updateTime();
    }, 1000);
  };

  updateTime();
});
```

Wie wir in Listing 1-8 erkennen können, weisen wir der Scope-Variable `now` initial den Wert `Loading...` zu. Dieser Wert erfüllt allerdings nicht die Voraussetzung des `date`-Filters und wird somit zunächst von dem Filter ignoriert. Anschließend definieren wir in unserem Controller die Funktion `updateTime()`, die der Scope-Variable `now` im Sekundentakt ein neues Date-Objekt zuweist. Dazu nutzen wir den `$timeout`-Service von AngularJS, anstatt die entsprechenden JavaScript-Funktionen `setTimeout()` bzw. `setInterval()` direkt aufzurufen. Grund dafür ist, dass der `$timeout`-Service intern zusätzlich noch die Mechanik anstößt, die dafür sorgt, dass das Framework im Kontext der Zwei-Wege-Datenbindung überprüft, inwiefern sich die Werte unserer Scope-Variablen verändert haben, und die Ansicht entsprechend aktualisiert. Das führt dazu, dass wir in unserer Ansicht im Sekundentakt eine aktualisierte Zeitausgabe erhalten.

Neben den Filtern, die AngularJS mitbringt, haben wir die Möglichkeit, eigene Filter zu implementieren. Somit können wir Ausgabeformatierungen, die wir öfter in unserer Anwendung benötigen, in einem Filter kapseln und auf diese Weise wiederverwenden. Um das zu demonstrieren, wollen wir an dieser Stelle noch einen eigenen Formatierungsfilter schreiben, der einen längeren Text ab einer bestimmten Buchstabenanzahl abschneidet und außerdem an das Ende drei Punkte (»...«) anhängt. Unseren Filter nennen wir `truncate`.

Listing 1-9 *Definition des truncate-Filters*

```
dateApp.filter('truncate', function () {
    return function (input, charCount) {
        var output = input;
```

```
        if (output.length > charCount) {
            output = output.substr(0, charCount) + '...';
        }

        return output;
    };
});
```

In Listing 1-9 sehen wir die Definition unseres `truncate`-Filters. In AngularJS definieren wir einen Filter mit einem Aufruf der `filter()`-Funktion, die zwei Parameter erwartet. Der erste Parameter ist eine Zeichenkette, die den Namen des Filters enthält. Als zweiten Parameter erwartet die Funktion eine Funktion, die eine Funktion zurückgibt. Es mag sein, dass das Ganze an dieser Stelle etwas kompliziert klingt. Wie wir aber auch schon in den Beispielen davor gesehen haben, ist die Nutzung von *anonymen Funktionen* und *Funktionen höherer Ordnung* zur Definition von Anwendungsbausteinen ein gängiges Muster in AngularJS.

Ausschlaggebend für die Definition unseres Filters ist die sogenannte Filterfunktion. Das ist die Funktion, die in Listing 1-9 die Filterlogik enthält und die wir letztendlich zurückgeben. Das Framework ruft diese Funktion jedes Mal auf, wenn die Expression, in der der Filter genutzt wird, neu evaluiert wird. Als ersten Parameter erhalten wir den bis dahin evaluierten Wert der Expression. Und mithilfe des zweiten Parameters können wir optional auf den Parameter des Filters zugreifen. Wie wir bei dem `date`-Filter bereits gesehen haben, erwarten viele Filter einen Parameter. Der `date`-Filter erwartet z. B. eine Formatierungszeichenkette für das Datum, also beispielsweise `dd. MMMM yyyy`.

Auch unser `truncate`-Filter erwartet einen Parameter. Er gibt an, ab welcher Anzahl an Buchstaben die Eingabezeichenkette abgeschnitten werden soll. Die eigentliche Implementierung unseres Filters ist eher trivial. Wir überprüfen, ob die Zeichenkette mehr Buchstaben enthält als die Anzahl der Buchstaben, ab der abgeschnitten werden soll. Ist das der Fall, schneiden wir die Zeichenkette entsprechend ab und hängen hinten – wie in unserer Anforderung definiert – drei Punkte an.

Der Vollständigkeit halber zeigt der nächste Codeausschnitt die Benutzung unseres soeben definierten `truncate`-Filters.

Listing 1-10
Nutzung unseres truncate-Filters

```
<!DOCTYPE html>
<html ng-app="dateApp">
<head>
    <meta charset="utf-8">
</head>
```

```
<body ng-controller="DateCtrl">

Today's date:
{{ now | date:'dd. MMMM yyyy' }}<br>

And the time is:
{{ now | date:'HH:mm:ss' }}<br>
{{ 'This is an example for a long text' | truncate:18 }}

<script src="lib/angular/angular.js"></script>
<script src="scripts/dateApp.js"></script>

</body>
</html>
```

Wie in Listing 1-10 zu sehen ist, erfolgt die Benutzung äquivalent zum `date`-Filter. In diesem Beispiel geben wir allerdings keine Scope-Variable aus, sondern definieren die Eingabezeichenkette direkt innerhalb der Expression. Für unsere Zwecke ist das an dieser Stelle ausreichend.

Das Ergebnis ist also, dass die Zeichenkette nach dem 18. Zeichen abgeschnitten wird und als Indikator drei Punkte an sie angehängt werden.

Zusammenfassung

- Mithilfe der *Zwei-Wege-Datenbindung* aktualisiert AngularJS unsere Ansicht automatisch, wenn sich ein Wert in dem für diese Ansicht gültigen *Scope* verändert hat.
- Das Ganze funktioniert auch in die Gegenrichtung. Bei Benutzerinteraktionen, die eine Änderung in der Ansicht nach sich ziehen, sorgt das Framework dafür, dass sich diese Änderungen auch in dem Scope widerspiegeln.
- Mithilfe von *Direktiven* können wir HTML um neue Elemente und Attribute erweitern. Auf diese Weise können wir z. B. wiederverwendbare Komponenten erstellen, die hinterher sogar anwendungsübergreifend verwendet werden können.
- In AngularJS existieren zwei Arten von Filtern: die *Formatierungs- und die Collection-Filter*.
- Mittels *Formatierungsfiltern* können wir die Ausgaben, die wir mit Expressions produzieren, vorher transformieren. Dazu bringt das Framework schon von Haus aus eine solide Sammlung von Filtern mit. Dazu gehören z. B.: `date`, `uppercase` und `lowercase`.

- Die *Collection-Filter* werden in Verbindung mit der *ngRepeat*-Direktive verwendet und in dem Kapitel näher betrachtet, in dem wir für unser Beispielprojekt eine Listenansicht erstellen.
- Nun haben wir einen ersten Eindruck von AngularJS erhalten und können nachvollziehen, inwiefern uns das Framework bei der Erstellung von modernen Webanwendungen unterstützen kann.