15

Services näher betrachtet

[Funktionsweise von Services im Detail 1](#_Toc407042778)

[Optionen um Services zu erstellen 5](#_Toc407042779)

[Services direkt mittels Provider erstellen 7](#_Toc407042780)

[Verwendung eines Decorators 8](#_Toc407042781)

In der Einführung zu Angular wurde kurz das Thema Services vorgestellt. Wie gezeigt, ermöglichen es diese, wiederverwendbare Komponenten zu erstellen und Geschäftslogik aus Controllern auszulagern, um somit zu einer Trennung von Verantwortlichkeiten beizutragen. Neben diesem Effekt, wirkt auch das in AngularJS eingesetzte Dependency Injection (DI) System positiv dazu bei, definierte Services leichter testen und gegebenenfalls auch austauschen zu können. Dies hilft langfristig entwickelte Anwendungen stabil zu halten und erstellte Services gegebenenfalls auch für andere Projekte wieder zu verwenden.

Ein Blick in die AngularJS-Dokumentation zum Thema Services lässt jedoch rasch erkennen, dass die Welt nicht mehr so klar scheint. Obwohl von Services die Rede ist, geschieht die Umsetzung mittels eines *factory-*Konstrukts. Daneben existieren jedoch auch die *service-Funktion* sowie eine ominöse *provider-Funktion*.

Das Ziel dieses Kapitels ist es, einen verständlichen Überblick über die verschiedenen Varianten, um Services in AngularJS zu erzeugen, zu geben und wie diese einzusetzen sind.

# Funktionsweise von Services im Detail

Um den Unterschied der verschiedenen Varianten, einen Service zu erstellen, zu verstehen, ist es sinnvoll, einen Schritt zuvor bei dem *$provide*-Objekt anzusetzen. Dieses kommt in der Modul-Konfiguration zum Einsatz und hat die Aufgabe, durch verschiedene Methoden, andere Module und Komponenten mittels des *$injector*-Services in der Anwendung zu registrieren. Das *$injector*-Service übernimmt dabei die Aufgabe des Ladens, Wiederbeschaffens und Instanziieren eines Moduls oder Typen, sowie dem Ausführen von Service-Methoden. Dies geschieht über die Reihenfolge und Definition des *$provide*-Objektes. Zur Definition eines eigenen Services stellt *$provide* dafür mehrere Vorgehensweisen (siehe Tabelle 1) zur Verfügung.

Tabelle 1: Methoden des $provide-Services

|  |  |
| --- | --- |
| **Methode** | **Beschreibung** |
| provider(provider) | Registriert einen Service Provider im $injector |
| factory(fn) | Registriert eine Funktion welche ausgeführt wird um einen Service zu erzeugen |
| service(class) | Registriert eine Konstruktorfunktion, welche mittels *new* instanziiert wird um einen Service zu erzeugen |
| value(obj) | Erstellt ein Wert/Objekt welches nur durch Services verwendet werden kann. Nicht von Providern konsumierbar. |
| constant(obj) | Definiert einen sich nicht verändernden Wert, welcher sowohl von Services als auch Providern verwendet werden kann. |
| decorator(decorator) | Greift in die Instanziierung eines Services ein um es zu Verändern oder Anzupassen und gibt dann den originalen Service, oder eine ummantelte Version (Dekorator), welcher Zugriffe auf den originale Service weiterleitet, zurück. |

Um sich für die jeweils passende Variante entscheiden zu können, ist es wichtig zu wissen, wie der Prozess der Serviceerstellung in Angular funktioniert. Die Hierarchie dieser Erstellung ist in Abbildung 0‑1 veranschaulicht. *$provide* ist, wie erwähnt, der Service-Registrator, welcher für die Anmeldung eines Services zuständig ist. Die tatsächliche Erstellung übernimmt ein sogenannter Service-Provider. Dieser ist eine Methode des $provide-Objekts, welche die Aufgabe hat, einen Service anhand einer Service-Factory zu instanziieren. Wie **Listing 15.1** zeigt, geschieht dies durch das zurückgeben eines Objektes welches die Methode *$get* Implementiert. Diese definiert die tatsächliche Service-Funktionalität. Der Service-Provider ist somit als eine Art Blaupause zu verstehen, die definiert, wie ein Service konkret zu erstellen ist. Abschließend stehen dafür die Methoden *factory* sowie *service* zur Verfügung.

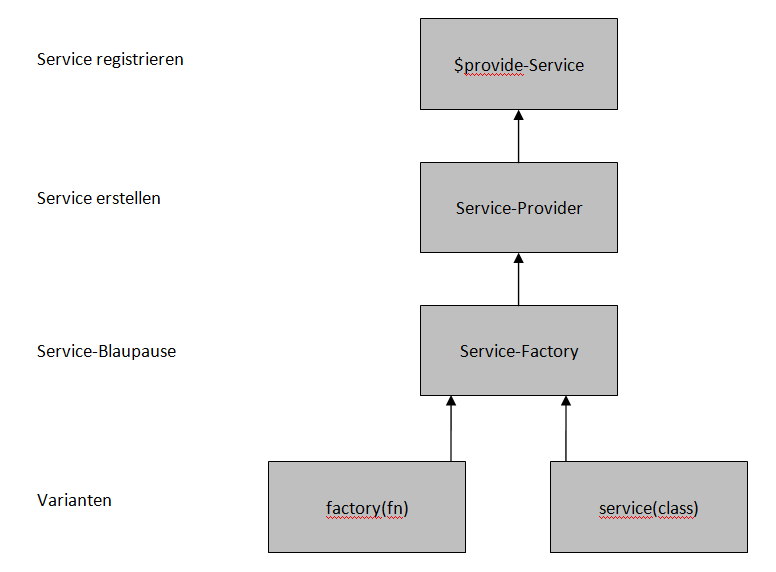


Abbildung 0‑1: Erstellungsprozess eines Services in Angular

app.provider('Demo', function () {

return {

$get: function () {

return {

// Service-Funktionalität hier implementieren

}

}

};

});

**Listing 15.1:** Funktionsweise des Provider-Services

Grundlegend erledigen sowohl die *service*- als auch die *factory*-Methode die gleiche Aufgabe, nämlich einen Singleton zu erschaffen. Dabei handelt es sich um ein Entwurfsmuster welches dafür sorgt, dass von einer Klasse (im Falle von JavaScript einer Funktion) nur eine Instanz existiert. Der grundlegende Gedanke dabei ist, einen einheitlichen Zugriff auf Daten, Zustände und Methoden eines Dienstes anzubieten. Somit ist der eigentliche Unterschied zwischen den beiden Varianten in erster Linie reine Geschmackssache da sich mit beiden Arten, das selbe Ziel erreichen lässt.

Eine Factory ist, wie in **Listing 15.2** zu sehen, im Grunde nur eine Ummantelung für den zuvor erwähnten Service-Provider. Dabei reicht sie die übergebene Funktion *fn* an dessen *$get*-Methode weiter, welche als definierte Schnittstelle für den *$injector*-Service gilt.

**function** factory(name, fn) {

**return** provider(name, { $get: fn });

}

**Listing 15.2:** Interne Umsetzung der Factory in AngularJS

Das Beispiel in **Listing 15.3** registriert eine Factory in einer Angular-App, mittels der *factory*-Methode. Diese erhält als Parameter den Namen des künftigen Services, sowie eine Service-Factory-Funktion. Diese beinhaltet den Service-Code. Hier zeigt sich ein möglicher Vorteil in der Verwendung der *factory*-Methode, da sich auf diese Weise recht einfach das Revealing-Module-Pattern einsetzen lässt. Dieses dient dazu, einerseits, mittels des zurückgegebenen Objekts, eine öffentliche Schnittstelle zu definieren, andererseits jedoch etwaige Implementierungen, in Form von privaten Methoden, zu verfassen, welche alsdann außerhalb des Moduls nicht sichtbar sind.

app.factory('DemoFactory', **function**() {

**var** privateMethod = **function**() {

**return** "DemoFactory";

};

**return** {

publicMethod: **function**() {

**return** privateMethod();

}

};

});

**Listing 15.3:** Beispiel zur Verwendung einer Factory

Alternativ zur *factory*-Methode, zeigt **Listing 15.5**, wie der Entwickler mittels der *service*-Methode einen neuen Service erzeugen kann. Was bei der Umsetzung der Methode (siehe **Listing 15.4**) auffällt, ist, dass intern die zuvor beschriebene Factory in Kombination mit dem *$injector*-Service Anwendung findet. Letzteres übernimmt die Instanziierung der übergebenen Konstruktorfunktion und stellt somit das Dasein eines Singletons sicher. Die innere Factory gibt dann das instanziierte Objekt zurück. Oftmals bevorzugen Entwicklern mit Hang zur klassischen Objektorientierung den Service-Ansatz, da sich damit, wie im Beispiel gezeigt, ein klassenähnliches Konstrukt erzeugen lässt, welches die *service*-Methode sodann als Parameter entgegennimmt.

**function** service(name, constructor) {

**return** factory(name, ['$injector', **function**($injector) {

**return** $injector.instantiate(constructor);

}]);

}

**Listing 15.4:** Interne Umsetzung der Service-Methode in AngularJS

**var** DemoClass = **function**() {

**var** privateMethod = **function**() {

**return** "DemoService";

};

**return** {

publicMethod: **function**() {

**return** privateMethod();

}

};

};

app.service('DemoService', DemoClass);

**Listing 15.5:** Beispiel der Service-Methode

Hinweis Beginn

Eine gängige Meinung in diversen Internet-Foren ist, dass der Entwickler die *factory*-Methode wegen der Verwendung des Revealing-Module-Patterns bevorzugen sollte. Wie **Listing 15.5** jedoch zeigt, lässt sich dies auch mit einem Service bewerkstelligen und stellt somit nur einen unbegründeten Mythos dar.

Hinweis Ende

# Optionen um Services zu erstellen

Wie in vorherigem Abschnitt erklärt, gibt es in der Verwendung der betrachteten Methoden kaum Unterschiede, da letzten Endes stets ein Service-Provider angestoßen wird. Das Abstraktionslevel dient somit in erster Linie dazu, dem Entwickler die Möglichkeit der Wahl der persönlich bevorzugten Arbeitsweise zu geben.

Um jedoch eine mögliche Entscheidungsfindung nachzuvollziehen, kann das folgende Beispiel Abhilfe leisten. Herr Mustermann beginnt seine Arbeit an einem neuen Feature, welches Daten in einer Liste darstellen soll, damit, dass er, wie in ***Listing 15.6*** gezeigt, einen v*alue*-Service erstellt, welcher Werte hartcodiert hinterlegt. Der Controller *ListenCtrl*, bekommt den *DatenService* nun injiziert und stellt eine Funktion *neuesElement* zur Verfügung, welche dem *value*-Service weitere Werte hinzufügen kann.

app.value('DatenService', ['Test1', 'Test2', 'Test3']);

app.controller("ListenCtrl", **function**($scope, DatenService) {

$scope.listenDaten = DatenService;

$scope.neuesElement = **function**(elem) {

DatenService.push(elem);

};

});

***Listing 15.6*:** *DatenService-Definition als Value-Service*

Nach der ersten Präsentation der Umsetzung, bekommt Herr Mustermann eine Zusatzanforderung, welche besagt, dass es sich nun nicht mehr um statische Werte handelt, sondern diese von einem Backend zu beziehen sind. Ebenso soll die Erweiterung der geladenen Daten unabhängig vom Controller geschehen, um eine Trennung von Zuständigkeiten zu gewährleisten.

Daraufhin wandelt der Entwickler, wie in **Listing 15.7** zu sehen ist, den Value-Service in einen Factory-Service um. Dieser verlangt nun nach dem *$http*-Service, welchen die private Methode *datenLaden* dazu verwendet, Daten vom Backend abzuholen und in der Factory an einem einheitlichen Ort zur Verfügung zu stellen. Der Rückgabewert ist ein anonymes Objekt mit den Funktionen *getDaten*, welche ein simpler Getter der Backend-Daten ist, sowie der aus dem Controller extrahierten Methode *neuesElement*. Wie im *ListenCtrl* zu sehen ist, ändert sich die Definition der Abhängigkeiten des Controllers dadurch nicht. In der neuen scope-Eigenschaft *$scope.service* liegt nun der *DatenService*. Die View bindet sodann auf dessen *getDaten*-Funktion. Die bisher bestehende Funktion *neuesElement*, ist nun ein Stellvertreter der vom Service zur Verfügung gestellten Version.

*// Schritt 2: Factory*

app.factory('DatenService', **function**($http) {

**var** backendDaten = **null**;

**var** datenLaden = **function**() {

$http.get('PFAD/ZUM/BACKEND').success(**function**(daten) {

backendDaten = daten;

});

};

datenLaden();

**return** {

getDaten: **function**() {

**return** backendDaten;

},

neuesElement: **function**(elem) {

backendDaten.push(elem);

}

};

});

app.controller("ListenCtrl", **function**($scope, DatenService) {

$scope.service = DatenService;

$scope.neuesElement = **function**(elem) {

$scope.service.neuesElement(elem);

};

});

**Listing 15.7:** DatenService-Definition als Factory

Beim nächsten Entwickler-Meeting stellt das Team jedoch fest, dass zahlreiche Programmierkollegen aus der klassischen OOP-Entwicklung kommen und sehnsüchtig nach einer Klasse suchen, um eine bereits definierte Formatierungsmethode einer bestehenden Klasse, wiederzuverwenden. Um dem entgegenzukommen, stellt Herr Mustermann die Factory auf eine objektorientiertere Service-Definition (siehe **Listing 15.8**) um.

Die Klasse *UpperCaseFormatter,* stellt mit der Methode *format*, eine Formatierungsfunktion zur Verfügung, welche der *DatenService* ebenso anbieten soll. Dazu erstellt der Entwickler eine Konstruktorfunktion *DatenServiceClass,* welcher das *$http*-Service als Parameter entgegen nimmt. Der erstellte Konstruktor kümmert sich nun um das Laden der Daten vom Backend. Als nächstes erhält die *DatenServiceClass*, als JavaScript-Prototypen, den bereits bestehenden *UpperCaseFormatter*. Anschließend ist die Eigenschaft *constructor* des Prototypen danach wieder zurückzusetzen, um weitere Vererbungen zu erlauben.

Sämtliche Funktionen aus dem Factory-Beispiel (siehe **Listing 15.7**), sind nun als Prototypen-Methoden definiert. Abschließend registriert die service-Methode die neu erstellte Klasse. Der *ListenCtrl*, kann, abgesehen vom neuen Formatierungsaufruf, unverändert bleiben, da die selben öffentlichen Schnittstellen erhalten bleiben.

**var** UpperCaseFormatter = **function**() {};

UpperCaseFormatter.prototype.format = **function**(val) {

**return** val + " :-) ";

};

**var** DatenServiceClass = **function**($http) {

**var** self = **this**;

self.backendDaten = **null**;

$http.get('PFAD/ZUM/BACKEND').success(**function**(daten) {

self.backendDaten = daten;

});

};

DatenServiceClass.prototype = **new** UpperCaseFormatter();

DatenServiceClass.prototype.constructor = DatenServiceClass;

DatenServiceClass.prototype.getDaten = **function**() {

**return** **this**.backendDaten;

};

DatenServiceClass.prototype.neuesElement = **function**(elem) {

**this**.backendDaten.push(elem);

};

app.service('DatenService', DatenServiceClass);

app.controller("ListenCtrl", **function**($scope, DatenService) {

$scope.service = DatenService;

$scope.neuesElement = **function**(elem) {

$scope.service.neuesElement($scope.formattiereAusgabe(elem));

};

$scope.formattiereAusgabe = **function**(elem) {

**return** DatenService.format(elem);

};

});

**Listing 15.8:** DatenService-Definition als Service

Dieses Beispiel zeigt auf, wie der Entwickler auf sich ändernde Anforderungen, im Rahmen der AngularJS-Entwicklung, eingehen kann. Das Refactoring eines Service-Typen in einen Anderen, kann zumeist ohne größere Schwierigkeiten geschehen. Ausschlaggebend dafür ist das flexible DI-System, welches unabhängig von der Art des Services, dieses richtig an den Controller übermittelt.

# Services direkt mittels Provider erstellen

Wie schon in **Listing 15.3** erwähnt, stellt eine *factory*- und damit schlussfolgender auch die *service*-Methode, eine alternative Schreibweise für die Umsetzung eines Providers. Der *provider*-Service dient somit als Grundgerüst für die Erstellung der anderen Service-Arten.

**Listing 15.9** zeigt wie sich der *provider*-Service dazu einsetzen lässt, einen einfachen *Hello*-Service zu erstellen, welcher dazu dient eine Begrüßung auf der Konsole auszugeben. Ähnlich wie die Methoden *factory* und *service*, ist die Methode *provider* direkt vom *app*-Modul selbst aus aufrufbar. Das Beispiel definiert nun eine Variable *whom*, die die zu begrüßende Person darstellt. Der Rückgabewert des Providers ist ein Objekt, welches eine Konfigurationsmethode *greetWhom* zur Verfügung stellt, welche der zuvor eingeführten Variable *whom* einen Wert vergibt. Damit der Provider auch als ein solcher erkannt wird, ist es notwendig, auch die Methode *$get* im Rückgabeobjekt zu definieren. Diese wiederum liefert sodann die tatsächliche Funktionalität des Services zurück, das hier ein Objekt mit der Eigenschaft *greeting* ist, welches die komplette Begrüßung beinhaltet. Bei näherer Betrachtung der *$get*-Funktion und der internen Umsetzung einer *factory*-Methode aus dem Beispiel in **Listing 15.3**, lässt sich die Ähnlichkeit der Vorgehensweisen erkennen. Nach der Definition des provider-Services, zeigt das Beispiel wie ein solcher, sich mit einem *app.config*-Block vor der Verwendung noch anpassen lässt. Um Zugriff auf den Provider zu erhalten, übergibt der Entwickler der Funktion den Service. Der Name setzt sich dabei aus dem Namen des *provider*-Services sowie dem Zusatz *Provider* zusammen. Nun ist es möglich, auf Methoden des Services zuzugreifen und somit, wie im Beispiel zu sehen, die Begrüßende Person zu setzen. Abschließend zeigt das Beispiel, dass die Verwendung des Services in einem Controller, auf die selbe Art und Weise wie bei anders erzeugten Services geschieht.

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

app.provider('Hello', function () {

var whom;

return {

greetWhom: function (value) {

whom = value;

},

$get: function () {

return {

greeting: 'Hello ' + whom + '!'

}

}

};

});

app.config(function (HelloProvider) {

HelloProvider.greetWhom('AngularJS User');

});

app.controller('MainCtrl', function (Hello) {

console.log(Hello.greeting);

});

**Listing 15.9:** Umsetzung eines Hello-Services mit dem provider-Service

# Verwendung eines Decorators

Nach den Möglichkeiten einen Service zu erstellen, behandelt dieser Abschnitt die in Tabelle 1 zuletzt erwähnten Dekoratoren. Diese dienen dazu, bestehende Services vor deren Initialisierung abzufangen und wenn nötig abzuändern. Es handelt sich hierbei um eine Variante des Dekorierer-Strukturmusters, welches dazu dient, ein Objekt vor dem Zugriff um weitere Funktionen zu erweitern. Wegen der dynamischen Natur JavaScripts, welche es erlaubt neue Methoden und Attribute auf ein bereits bestehendes Objekt zu hängen, eignet sich somit ein Angular-Decorator hervorragend dazu, ein bestehendes Service vor dem Instanziieren zu verändern.

Um die Funktionsweise zu verdeutlichen, zeigt das folgende **Listing 15.10**, wie sich das in **Listing 15.6** erläuterte *DatenService*, manipulieren lässt. Der Decorator ist während des Konfigurationsvorganges der App zu definieren und stellt die Methode *decorator* des *$provide*-Services dar. Der erste Parameter ist der Name des Services, der zu verändern ist. Der zweite Parameter ist eine Funktion, welche den originalen Service wiederum als Parameter erhält. Das folgende Beispiel in **Listing 15.10**, erweitert das bestehende Array des Value-Services, um einen weiteren Eintrag. Abschließend liefert der Dekorator als Ergebnis das veränderte Service zurück.

app.config(**function**($provide) {

$provide.decorator('DatenService', **function**($delegate) {

$delegate.push('Test 1000 -> Decorated');

**return** $delegate;

});

})

**Listing 15.10:** Angular-Decorator zum Verändern eines Value-Services

Tatsächlich wurde in obigem Beispiel, der Service selbst nicht verändert, sondern nur dessen Daten. Im Falle einer Factory oder über die *service*-Methode erstelltem Service, lässt sich aber genauso gut auch die Funktionalität selbst verändern. **Listing 15.11** zeigt dazu, wie der Entwickler die zweite Variante des *DatenServices* (siehe **Listing 15.7**) in Form einer Factory, so verändern kann, dass beim Zugriff auf die *getDaten-*Methode, diese zusätzlich eine Konsolennachricht ausgibt. Die Signatur des Decorators selbst bleibt dabei gleich. Das Beispiel bewahrt eine Referenz der originalen Implementierung der Zielmethode, in der Hilfsvariable *origFn* auf. Wichtig dabei ist, dass die Funktionsreferenz und nicht das Ergebnis selbst zu hinterlegen ist, was durch das Fehlen der runden Klammern nach dem Methodennamen zu erkennen ist. Als nächstes geschieht die Überladung der Methode *getDaten* des Delegaten, mit der neuen Methodenimplementierung. Diese schreibt nun zuerst das Konsolenkommentar und retourniert im Anschluss daran den Wert der zuvor gemerkten Funktionsreferenz.

app.config(**function**($provide) {

$provide.decorator('DatenService', **function**($delegate) {

**var** origFn = $delegate.getDaten;

$delegate.getDaten = **function**() {

console.log('Vom Decorator erweitert');

**return** origFn();

};

**return** $delegate;

});

})

**Listing 15.11:** Angular-Decorator zum Verändern einer Factory

Doch mit Dekoratoren lässt sich in AngularJS noch viel mehr anstellen. Neben Services, ist es auch möglich, Direktiven, sowohl Selbsterzeugte als auch Angular-native, zu verändern. Dies kann vor allem dann nützlich sein, wenn der Entwickler mit der Funktionalität einer bestehenden Direktive nicht ganz glücklich ist, jedoch nicht unnötigerweise eine komplett neue Direktive erstellen möchten.

Zur Erläuterung der Vorgehensweise, zeigt **Listing 15.12** die benutzerdefinierte Direktive *hello*, welche ein Tag, welches mit dem Attribut *hello* versehen ist, durch den Inhalt *Hallo AngularJS* ersetzt. Nun wäre es jedoch praktisch, auch ein eigenes Tag *hello* verwenden können, anstatt die Direktive nur als Attribut zu nutzen. Nebenbei wäre es auch noch nützlich jede Ausführung in der Konsole festzuhalten um die Anzahl der Aufrufe zu sehen.

*// HTML*

**<h3>**Dekorierte Hello Direktive**</h3>**

**<div** hello**/>**

**<hello** **/> ?!? KLAPPT NICHT ?!?**

*// Direktive*

angular.module('kapitel15ServicesApp')

.directive('hello', **function**() {

**return** {

restrict: 'A',

replace: **true**,

template: '<p>Hallo AngularJS</p>'

};

});

**Listing 15.12:** Beispiel der Hello-Directive

Wegen des *restrict*-Attributes, welches auf A, sprich Attribut, eingeschränkt ist, kann die Direktive diese Funktionalität ebenso wie das Logging nicht liefern. Genau hierbei kann ein Decorator helfen.

Dazu geht der Entwickler, ähnlich wie beim dekorieren eines Services vor. Im Konfigurationsmodus der App, ist die Hilfsfunktion *decorator* des *$provide*-Services aufzurufen. Der Name des Dekorators setzt sich aus dem Namen der zu dekorierenden Direktive und dem Zusatz *Directive* zusammen. Im Falle des Beispiels vermag somit der Dekorator den Namen *helloDirective* als Direktive zu erkennen und gibt das Original als Parameter an die Dekorator-Funktion weiter. Der Parameter *$delegate* enthällt nun nicht wie bei der Arbeit mit Services, das zu dekorierende Objekt selbst, sondern eine Ummantelung, welche als Eigenschaft *0*, das zu dekorierende Objekt enthält. Dieses legt das Beispiel in der Variable *directive* ab. Desweiteren erstellt es eine Variable *counter*, um künftig die Anzahl der Aufrufe mitzuzählen. Nun setzt das Beispiel die *restrict*-Eigenschaft auf den Wert *EA*, um somit die Direktive sowohl als Attribut, als auch als Element verwendbar zu machen. Daraufhin muss sich der Entwickler um die Neukompilierung der Direktive kümmern, wodurch diese die vollzogenen Änderungen auch tatsächlich ausführt. Dies ist ausführlich in Kapitel X behandelt. Dazu ist die Methode *compile* zu erstellen, welche als Rückgabewert eine Funktion mit der vordefinierten Signatur zurückliefert. Genau in dieser lässt sich nun die Anforderung des Loggens implementieren. Dazu reicht der Aufruf der *console.log-*Methode mit der zuvor erhöhten Variable *counter*. Diese Nachricht erscheint nun beim Initialisieren einer jeden *hello*-Direktive. Abschließend retourniert der Dekorator die erfolgreich veränderte Direktive.

app.config(**function**($provide) {

$provide.decorator('helloDirective', **function**($delegate) {

**var** directive = $delegate[0];

**var** counter = 0;

directive.restrict = 'EA';

directive.compile = **function**() {

**return** **function**(scope, element, attrs) {

console.log('Hello Direktive ' + (++counter) + ' mal ausgeführt!');

};

};

**return** $delegate;

});

})

**Listing 15.13:** Decorator für Hello-Directive