Webrontend Development

Angular 2.0

Dependency Injection und Unit-Testing mit Angular 2.0

DI und gute Testbarkeit waren schon immer ein Alleinstellungsmerkmal für AngularJS. Mit der neuen Version wurden viele Details verbessert.

**Hinweis** Das hier gezeigte Beispiel nutzt eine Vorschauversion von Angular 2.0. Der gezeigte Code muss für spätere Versionen wahrscheinlich angepasst werden.

Einleitung

In den beiden vorangegangenen Artikeln zu Angular 2.0 wurde zunächst ein modulares Setup auf Basis von SystemJS beschrieben. In der zweiten Ausgabe haben wir Ihnen anhand einer Beispielanwendung die neue Template-Syntax vorgestellt. Es wird Zeit, der Anwendung neue Funktionen zu geben, professionelle Entwurfsmuster anzuwenden und Fehlerfreiheit des Codes mit Unit-Tests zu beweisen!

Inversion of Control

Das Dashboard ist die Demo-Anwendung aus der letzten Ausgabe. Die Anwendung soll in diesem Artikel eine neue Funktionalität erhalten. Es soll per AJAX nach dem aktuell günstigsten Benzinpreis gesucht werden. Mit dem ermittelten Preis kann man dann die verwalteten Autos auftanken.

(Bild 1)

Wenn man an einer beliebigen Stelle im Programmcode eine andere Funktionalität benötigt, dann liegt es zunächst nahe, jene andere Funktionalität an Ort und Stelle zu initialisieren. Ein erster Ansatz könnte wie folgt ausschauen:

var Dashboard = function() {

this.gasService = new GasService();

// gasService verwenden

this.gasService.getBestPrice();

}

Dieses Vorgehen ist prinzipiell einwandfrei - nur stößt man mit steigender Menge an Code an eine Grenze. Der Code wird zunehmend unübersichtlicher, schwerer zu Warten und verweigert sich einem einfachen Test-Setup. Dem Problem lässt sich begegnen, in dem man die Verantwortung zum Erzeugen von Abhängigkeiten an eine übergeordnete Stelle abgibt.

Das ist die Idee hinter dem Prinzip des **"Inversion of Control"**. Bei diesem Prinzip kehrt man die Verantwortlichkeit einfach um. Das Prinzip findet sich in verschiedenen Entwurfsmustern bei allen Programmiersprachen wieder. AngularJS zum Beispiel verwendet das Entwurfsmuster **"Dependency Injection"**. Ein Framework im Kern von AngularJS sorgt dafür, dass die benötigte Abhängigkeit identifiziert wird und der Konstruktor-Funktion beim Aufruf bereitgestellt wird. In AngularJS 1.x kann man einen Service wie folgt anfordern:

Dependency Injection in Angular 2.0

(Bild 2)

Mit der Unterstützung von ECMAScript 6 bzw. von TypeScript wird die Bedienung nun viel vertrauter. So lässt sich mittels des Decorators @Inject die Abhängigkeit in den Konstruktor injizieren:

((Listingkasten))

Listing 1: Constructor Injection mit ES6

class GasService {

}

class Dashboard {

constructor(@Inject(GasService) gasService) {

console.log('Dependency:', gasService)

}

}

var injector = Injector.resolveAndCreate([Dashboard, GasService]);

var dashboard = injector.get(Dashboard);

((Ende Listingingkasten))

Die Methode *resolveAndCreate()* erzeugt einen einsatzbereiten Injector. Die Methode akzeptiert ein Array aus Typen oder Providern. Übergibt man einen Typ, so wird ein entsprechender Provider für diesen Typ erzeugt (*useClass* - wird später erläutert). Mit dem Einsatz von TypeScript, lässt sich die Schreibweise noch etwas mehr vereinfachen. Durch die Verwendung von Typen kann man auf den Decorator *@Inject* verzichten:

((Listingkasten))

Listing 2: Constructor Injection mit TypeScript

@Injectable()

class Dashboard {

constructor(gasService: GasService) {

console.log('Dependency:', gasService)

}

}

Damit dieses Beispiel funktioniert, muss TypeScript einen Hinweis dazu erhalten, dass der Konstruktor mit Decoratoren versehen werden soll. Dies geschieht mit dem Decorator *@Injectable()*. Das erzeugte JavaScript aus Listing 1 und Listing 2 unterscheidet sich schlussendlich kaum voneinander. Auf die Verwendung von *@Injectable()* kann verzichtet werden, sobald ein anderer Decorator die Klasse verziert. Weitere Decoratoren sind etwa *@Component()*, *@View()* oder *@RouteConfig()*. Da Angular 2.0 stark auf einen deklarativen Stil mittels Decoratoren setzt, benötigt man *@Injectable()* eigentlich nur für eigene Service-Klassen.

Der Injector versteht eine Reihe von Bauanleitungen. Hierzu verwendet man die Methode *provide()*. Das Beispiel aus Listing 1 kann auch in einer längeren Syntax ausgedrückt werden. Soll also ein **Token** als Klasse aufgelöst werden, so verwendet man **useClass**:

var injector = Injector.resolveAndCreate([Dashboard, GasService]);

// entspricht:

var injector = Injector.resolveAndCreate([

provide(Dashboard, {useClass: Dashboard}),

provide(GasService, {useClass: GasService}),

]);

Ebenso kann ein **Token** auch zu einem einfachen Wert aufgelöst werden (**useValue**):

var injector = Injector.resolveAndCreate([

provide('TEST', {useValue: 'Hello Angular2'})

]);

var test = injector.get('TEST');

Wie sie sehen, kann ein Token nicht nur ein Typ, sondern auch ein einfacher String sein.  
Ebenso findet man auch die aus AngularJS 1.x bekannten Factories wieder (**useFactory**):

// a factory can have own dependencies, too

var factory = (gasService: GasService) => {

return new Dashboard(gasService); // !!

};

var injector = Injector.resolveAndCreate([

provide(GasService, {useClass: GasService}),

provide(Dashboard, { useFactory: factory, deps: [GasService]})

]);

((Ende Listingingkasten))

Factories bieten sich immer dann an, wenn das Objekt eine speziellere Initialisierung benötigt. An dieser Stelle sei erwähnt, dass der Injector Klassen "lazily" instanziiert. Die Objekte werden erst zu dem Zeitpunkt erzeugt, zu dem sie benötigt werden und anschließend gecacht. Durch das Caching sind alle instanziierten Klassen Singletons. Dies gilt auch für die Rückgabewerte der Factory-Funktion. Die Methode *get()* verwendet den Cache, die Methode *resolveAndInstantiate()* hingegen nicht - in dem Fall wird auch die Factory ein weiteres Mal aufgerufen. In der Dokumentation auf Angular.io wird daher auch darauf hingewiesen, welche Methoden den Cache nutzen und welche nicht [[2](https://angular.io/docs/ts/latest/api/core/Injector-class.html)]. Weitere Beispiele für die unterschiedlichen Verwendungen von *provide* finden Sie in den Codebeispiele zum Artikel (Datei: *Injector\_tests.ts*).

Durchstarten

Die Methode *resolveAndCreate()* vom Injector kann man gut für ein schnelles Experiment oder in einem Unit-Test verwenden. Bei der Erstellung der eigentlichen Anwendung bedient man sich aber der bereits bekannten *boostrap* Methode. Zuvor haben wir bei dieser Methode nur den ersten Parameter verwendet. Über den ersten Parameter erwartet Angular die Einsteigs-Komponente der Anwendung - also im vorliegenden Fall die Dashboard-Komponente.

((Listingkasten))

Listing 3: Starten (bootstrapping) der Anwendung

// app.ts

import {bootstrap} from 'angular2/angular2';

import Dashboard from './components/dashboard-component';

bootstrap(Dashboard);

((Ende Listingingkasten))

Als zweiten Parameter akzeptiert die Methode wiederum ein Array aus Typen oder Providern. Sollte die Dashboard-Komponente oder eine andere Komponente den GasService benötigen, so lässt sich dieser wie folgt registrieren:

((Listingkasten))

Listing 4: Bootrapping mit Registrierung der Dependency GasService

// app.ts

import {bootstrap} from 'angular2/angular2';

import Dashboard from './components/dashboard-component';

import GasService from './models/gas-service';

bootstrap(Dashboard, [GasService]);

((Ende Listingingkasten))

Daten per AJAX laden

Die Demo-Anwendung muss natürlich noch den aktuell günstigsten Benzinpreis ermitteln können. Zu diesem Zweck verwenden wir die öffentliche Schnittstelle des Verbraucherinformationsdienstes Tankerkönig.de. Der Dienst bietet eine freie JSON-API für die Kraftstoffpreise des Bundeskartellamts an. Hierfür verwenden wir die injizierbare Klasse *http*. Der Rückgabewert der *get*-Methode ist ein Observable-Objekt. Hier kommen wir in Berührung mit dem neuen Prinzip der reaktiven Programmierung, welche das Arbeiten mit asynchronen Daten sehr vereinfacht. Für die aktuelle Aufgabe reicht es aus, die eintreffenden Daten mittels *map()* zu transformieren. An anderer Stelle kann dann der ermittelte Preis per *subscribe()* abonniert werden.

((Listingkasten))

Listing 5: Asynchrone Programmierung mit Observables

// gas-service.ts

import {Injectable} from 'angular2/core';

import {Http} from 'angular2/http';

import Station from './Station';

@Injectable()

export default class GasService {

apiUrl: string = 'https://creativecommons.tankerkoenig.de/json/list.php?lat=52.03&lng=13.0&rad=4&sort=price&type=diesel';

apiKey: string = '&apikey=XXX';

apiUrlAndKey: string;

constructor(private http: Http) {

this.apiUrlAndKey = this.apiUrl + this.apiKey

}

getBestPrice() {

return this.http.get(this.apiUrlAndKey)

.map(result => (<any>result).json().stations)

.map((stations: Array<Station>): number =>

stations[0].price

)

}

}

((Ende Listingingkasten))

((Listingkasten))

Listing 6: Mit subscribe das Ergebnis abonnieren (und sparsam tanken)

// dashboard-component.ts

export default class DashboardComponent {

constructor(private gasService: GasService) { }

refillTank(car: Car, amountOfMoneyToSpend: number) {

this.gasService

.getBestPrice()

.subscribe((bestPrice: number) => {

car.refillTank(amountOfMoneyToSpend / bestPrice);

},

err => console.error(err));

}

}

((Ende Listingingkasten))

Damit ist der Grundstein für das neue Feature gelegt. Der aktuell günstigste Preis liegt vor und kann in der Komponente verwendet werden. Es fehlen noch ein paar Anpassungen am Model und am Templating. Die vollständige Anwendung finden Sie im Codebeispiel zum Artikel [[1](https://github.com/Angular2Buch/angular2-testing)].

Karma einrichten

Unit-Tests verbessern die Qualität von Software. Tests beweisen, dass die Software das tut, wofür sie konzipiert wurde. Ebenso dokumentieren Tests fachliches Wissen und den Erkenntnisstand eines Entwicklers, den er zum Zeitpunkt der Erstellung hatte. Wenn man als Entwickler das existierende Wissen nicht durch Tests ausdrückt, ist die Wahrscheinlichkeit sehr hoch, dass das Wissen über die Zeit für einen selbst, für das Team und für das Unternehmen verloren geht. Die Verwendung von Angular erweist sich hierbei als großer Vorteil, da das Framework speziell darauf ausgerichtet ist, gut testbare Module zu erstellen.

Um Unit-Tests für JavaScript/TypeScript auszuführen, verwendet man am besten einen so genannten Test-Runner. Prinzipiell würde auch nur ein Browser ausreichen. Doch dieses Setup lässt sich schlecht automatisieren. Empfehlenswert ist der Test-Runner "Karma", welcher zusammen mit AngularJS von Google entwickelt wurde. Das Tool basiert auf Node.js und läuft somit auf allen gängigen Betriebssystemen. Erwähnenswert ist die Tatsache, dass Karma einen eigenen Webserver startet und dann einen echten Browser (z.B. den Internet Explorer, Firefox und Chrome) die JavaScript-Dateien ausführen lässt. Der eigene Webserver vermeidet technische Probleme, die man bei der Ausführung per lokalem Dateisystem hätte.

Die Installation von Karma nebst Plugins geschieht per NPM:

npm install karma karma-chrome-launcher karma-jasmine --save-dev

npm install karma-cli -g

Die Datei *package.json* wird dabei um neue "devDependencies" ergänzt. So kann man später per *npm install* das Setup jederzeit wiederherstellen. Die globale Installation des Karma command line interface (*karma-cli*) macht den Kommandozeilen-Befehl *karma* verfügbar. Mit *karma start* lassen sich nun die Unit-Tests starten. Beim Überprüfen der Datei *package.json* bietet es sich an, trotz der globalen Installation das Start-Script auf *karma start*festzulegen. So kann man später den Testrunner per *npm test* starten. Die Verwendung des "scripts"-Property ist eine empfehlenswerte Konvention in der Node.js-Welt. Mit den Befehlen *npm install*, *npm start* und *npm test* sollte jeder Node.js-Entwickler vertraut sein.

((Listingkasten))

Listing 7: Auszug aus der package.json

{

[...]

"devDependencies": {

"karma": "^0.13.15",

"karma-chrome-launcher": "^0.2.1",

"karma-jasmine": "^0.3.6"

},

"scripts": {

"start": "live-server",

"test": "karma start"

}

}

((Ende Listingingkasten))

Anschließend benötigt das Projekt eine Konfigurationsdatei, welche standardmäßig den Namen *karma.conf.js* trägt. Der Befehl *karma init* startet ein Kommandozeilen-Dialog, welcher bei der Erstellung der Datei hilft. Wie schon bei der Verwendung mit SystemJS/JSPM müssen anschließend noch Pfade gemappt werden. An dieser Stelle ist das Setup zum aktuellen Stand (Alpha-48) noch etwas unkomfortabel. Wir empfehlen Ihnen aktuell den "ng2-test-seed" [[3](https://github.com/juliemr/ng2-test-seed)] von Julie Ralph. Julie Ralph ist eine sehr bekannte Google-Mitarbeiterin, welche auch die Hauptentwicklerin des Oberflächen-Testtools Protractor ist. Kopieren Sie sich aus dem Github-Repository die beiden Dateien **karma.conf.js** und **karma-test-shim.js**. Die Codebeispiele zum Artikel enthalten ebenso die beiden Dateien. Achten Sie auf die verwendete Ordnerstruktur, sonst funktioniert es nicht. Die Datei **karma-test-shim.js** lädt die Tests per SystemJS. Überprüfen Sie im Fehlerfall in der Datei den Befehl *System.config()*. SystemJS haben wir bereits im 1. Artikel (Ausgabe 12/2015) kennen gelernt.

Unit-Tests mit Jasmine

Die Auswahl eines geeigneten Test-Frameworks fällt aktuell sehr leicht. Derzeit wird nur Jasmine vollständig von Angular 2 unterstützt. Jasmine hat eine Syntax im Behavior Driven Development (BDD)-Stil. Die Funktion *describe()* definiert eine Sammlung ("test suite") zusammenhängender Tests. Die Funktion erwartet zwei Parameter: Der erste Parameter ist ein String und beschreibt als Wort oder in wenigen kurzen Worten, was gerade getestet wird. Der zweite Parameter ist eine Funktion, die alle Spezifikationen ("Specs") beinhaltet. Die *it()* Funktion stellt konkret eine Spezifikation dar. Auch eine Spezifikation benötigt beschreibende Worte. Describe-Methoden können beliebig tief verschachtelt werden, um die Übersichtlichkeit zu erhöhen. Die eigentlichen Prüfungen geschehen durch die Funktion *expect()*. Die Funktion *beforeEach* läuft, wie der Name vermuten lässt, stets vor jeder Spezifikation ab. Hier lässt sich doppelter Code beim Initialisieren vermeiden. Der BDD-Stil von Jasmine ermöglicht es, Tests in natürlicher Sprache zu definieren. Listing Nr. 8 veranschaulicht die Syntax.

((Listingkasten))

Listing Nr. 8: Hello World mit Jasmine

describe("A suite", () => {

var number;

beforeEach(() => {

number = 1;

});

it("contains spec with an expectation", () => {

expect(number).toBeGreaterThan(0);

});

});

((Ende Listingingkasten))

|  |
| --- |
| ((Kasten))  **Die wichtigsten Methoden von Jasmine**   * describe(description: string, specDefinitions: () => void) - definiert eine Sammlung von Tests ("test suite") * beforeEach(action: () => void) - Setup * afterEach(action: () => void) - Teardown * it(expectation: string, assertion: () => void) - Spezifikation ("spec") * expect(actual: any) - Erwartung, wird zusammen mit einem Matcher verwendet |

Die Methoden von Jasmine (*describe()*, *it*, *expect* usw.) sind nicht neu, in jedem Unit-Test stehen diese Methoden seit jeher im globalen Gültigkeitsbereich zur Verfügung. Wir empfehlen aber, für einen Angular2-Test die globalen Methoden nicht direkt zu verwenden! Angular bietet dieselben Methoden über einen Import an (*angular2/testing*). Das Angular-Testing wrappt die Methoden und fügt neue Matcher hinzu. Durch diese Manipulation von Jasmine wird das Injizieren von Abhängigkeiten in Tests vereinfacht und das Testen von asynchronem Code ermöglicht. Zudem erhält man auch gleich die passenden "TypeScript type definitions". Folgende Zeile sollte demnach in keinem Test fehlen:

import { it, describe, expect, inject } from 'angular2/testing';

|  |
| --- |
| ((Kasten))  **Neue Matcher für Jasmine**  Angular-Testing wird mit einer Reihe von neuen Matchern ausgeliefert.   * toBePromise() * toBeAnInstanceOf(expected: any) * toHaveText(expected: any) * toHaveCssClass(expected: any) * toImplement(expected: any) * toContainError(expected: any) * toThrowErrorWith(expectedMessage: any) |

Komponenten testen

Das Modul Angular-Testing bietet eine neue Methode an, welche das Setup eines Unit-Tests sehr komfortabel gestaltet. Zu der bereits bekannten Methode *beforeEach* gesellt sich nun die Methode *beforeEachProviders*. Mit dieser Methode kann man vor der eigentlichen Ausführung des Tests den Injector mit Providern befüllen bzw. bestehende Provider überschreiben. Es lassen sich hierbei auch Kern-Funktionalitäten von Angular überschreiben. Wie bei den anderen Methoden zum DI-System akzeptiert *beforeEachProviders()* ein Array aus Typen oder Providern. Der Test aus Listing 9 beweist zum Beispiel, dass die Dashbard-Komponente stets mit einem gefüllten Array initialisiert wird.

((Listingkasten))

Listing 9: Verwendung von beforeEachProviders() und inject()

import { it, describe, expect, inject, beforeEachProviders } from 'angular2/testing';

import { HTTP\_PROVIDERS } from 'angular2/http';

import DashboardComponent from '../../app/components/dashboard-component';

import GasService from '../../app/models/gas-service';

describe('dashboard component', () => {

beforeEachProviders(() => [DashboardComponent, GasService, HTTP\_PROVIDERS]);

it('should have a predefined list of cars', inject([DashboardComponent], (dashboard: DashboardComponent) => {

expect(dashboard.cars.length).toBe(2);

}));

});

((Ende Listingingkasten))

Beachten Sie auch die Verwendung der Methode *inject()*. Sie ist dazu gedacht, in einer *beforeEach()* oder *it()* eine Abhängigkeit anzufordern. In der Quelltext-Dokumentation von Angular findet sich der Hinweis, dass es ggf. in Zukunft noch eine Syntax mit Decoratoren geben wird:

// aktuell

inject([DashboardComponent], (dashboard: DashboardComponent) => { /\* [...] \*/ })

// mögliche zukünftige Syntax

@Inject(dashboard: DashboardComponent) => { ... }

(Bild 3)  
(Bild 4)

Asynchronen Code testen

Oft muss man in der JavaScript-Welt auf etwas warten. Dies kann unter anderem die Antwort auf einen AJAX-Call sein, wie es auch unsere Anwendung bei den Preisdaten tut. Um das Problem abzubilden, verwendet man üblicherweise Callbacks, Promises oder Observables. Allen Herangehensweisen ist gemein, dass der ausgeführte Code asynchron abläuft. Der Ansatz in Angular 1.x war es, asynchronen Code in ein synchrones Format zu pressen. Man musste dazu in Unit-Tests speziell Rücksicht nehmen und zum Beispiel mit *$rootScope.$digest()*, *$httpBackend.flush()*oder *$timeout.flush()* manuell die Promises zu erfüllen um anschließend das Ergebnis überprüfen zu können. Dieser Ansatz ermöglicht zunächst elegante und leicht verständliche Tests. Doch gerade dieser Ansatz kann bei komplexeren Aufgabenstellungen eine Lösung erschweren, da das eigentliche Problem hinter der synchronen Fassade versteckt wird.

Angular 2 birgt das Potential, an dieser Stelle um einiges einfacher zu werden. Eine vergleichbare synchrone Fassade existiert nicht mehr. Daher müsste ein Test für asynchrone Methoden eigentlich stets wie folgt ausschauen:

((Listingkasten))

Listing 10: Normaler asynchroner Unit-Test

describe('async tests', () => {

it('usually need to signal that execution has been finished', (done) => {

setTimeout(() => {

expect(true).toBe(true);

done(); // !!!

}, 500)

});

})

((Ende Listingingkasten))

Seit Dezember 2015 (Alpha-47) ist jedoch ein sehr interessantes Feature in Angular vorhanden. Angular-Testing verwendet "zones.js" und "Microtasks" um selbstständig festzustellen, wann der Unit-Test abgeschlossen ist. Ein Beispiel für einen Unit-Test für asynchronen Code ist das Listing 11. Zunächst muss der Http-Service ausgemockt werden. Dies geschieht mit der bekannten Methode *provide()*. Mittels *subscribe()* können wir den Output des GasService empfangen. Der GasService wird dabei nicht gegen einen echten HTTP-Endpunkt gehen, sondern die gemockte Verbindung verwenden. Es fällt auf, dass ein Aufruf des Callbacks *done()* nicht notwendig ist - dies erledigt Angular für uns.

((Listingkasten))

Listing 11: ein asynchroner Unit-Test, der nicht mehr manuell abgeschlossen werden muss

import { beforeEachProviders, describe, expect, inject, it } from 'angular2/testing';

import { provide } from 'angular2/angular2';

import { MockBackend, BaseRequestOptions, Http, Response, ResponseOptions } from 'angular2/http';

import GasService from '../../app/models/gas-service';

describe('GasService', () => {

beforeEachProviders(() => [

BaseRequestOptions, MockBackend,

provide(Http, {

useFactory: (backend, defaultOptions) => new Http(backend, defaultOptions),

deps: [MockBackend, BaseRequestOptions]

}),

GasService

]);

it('should pick up the first price', inject([GasService, MockBackend], (gasService, backend) => {

backend.connections.subscribe(c => {

c.mockRespond(new Response(

new ResponseOptions({body: '{ "stations": [{ "price": 42 }, { "price": 4 }] }'})));

});

gasService.getBestPrice().subscribe((price) => {

expect(price).toBe(42);

});

}));

});

((Ende Listingingkasten))

Fazit

Mit Version 2 hat das Angular-Team viele technische Schwächen von AngularJS reagiert. Das neue Konzept wirkt wie aus einem Guss: Typen und Decoratoren durch TypeScript, modularer Code durch SystemJS und auf Basis dessen ein generalüberholtes DI-System. Das neue Prinzip leuchtet schnell ein und die Testbarkeit profitiert von dieser neuen Umsetzung. Die neuen Provider begegnen uns an vielen Stellen bei der täglichen Arbeit mit Angular 2. Ein Grundverständnis der dahinterliegenden Mechanismen erleichtert den Einstieg entsprechend signifikant.

In der nächsten Ausgabe der **Web und Mobile Developer** wird es wieder mehr um sichtbare Dinge in Angular 2 gehen. Wir betrachten dann die Formularverarbeitung und Validierung von Daten mit dem neuen Framework. Das "Cars Dashboard" wir dabei langsam zu einer vollwertigen Anwendung. Seien Sie gespannt.

((Autorenkasten))



Autor

**Johannes Hoppe** ist selbstständiger IT-Berater und Softwareentwickler. Er arbeitet derzeit als Architekt für ein Portal auf Basis von .NET und AngularJS. Er veranstaltet Trainings zu AngularJS und bloggt unter http://blog.johanneshoppe.de/.

((Ende Autorenkasten))

((Autorenkasten))



Autor

**Gregor Woiwode** ist als AngularJS und Mac-Entwickler für ein junges Unternehmen in Karlsruhe tätig. Er veranstaltet Trainings zu AngularJS und bloggt unter http://www.woiwode.info/blog/.

((Ende Autorenkasten))

(<https://github.com/Angular2Buch/angular2-testing>)

((Kasten))

Links zum Thema

[1] Codebeispiel zum Artikel https://github.com/Angular2Buch/angular2-testing

[2] Angular 2 for TypeScript: Injector

https://angular.io/docs/ts/latest/api/core/Injector-class.html

[3] ng2-test-seed von Julie Ralph

https://github.com/juliemr/ng2-test-seed

((Bildunterschriften))

((images/screenshot\_refill.png))

**Screenshot: Mit dem günstigsten Benzinpreis die Autos betanken** (Bild 1)

((images/images/injector.png))

**Screenshot: Die wichtigsten Bausteine für DI in Angular 2.0** (Bild 2)

((images/screenshot\_karma1.png))

**Screenshot: Karma führt Unit-Tests aus** (Bild 3)

((images/screenshot\_karma2.png))

**Screenshot: Karma führt Unit-Tests aus** (Bild 4)