DDC .NET Developer Conference 2015

Angular 2.0

Einführung & Schnellstart

Angular 2.0 wird in naher Zukunft fertig gestellt sein. Es gibt es bereits regelmäßige Vorabversionen für interessierte Entwickler. Das Angular-Team hat sich entschieden, alte Zöpfe rigoros abzuschneiden und ein komplett überarbeitetes Framework zu entwickeln. Die neue Version bricht mit bestehenden Konzepten - was für viel Aufregung gesorgt hat. Die Template-Syntax ist neu und man setzt nun Komponenten statt Controller ein. Auch der Einsatz von TypeScript rüttelt am einher gebrachten.

In diesem Workshop werden wir auf maßgebliche Änderung eingehen. Das vorliegende Handout ist ein Sammlung von (**unfertigen**) Fragmenten aus unseren Artikeln sowie unserem kommenden Buch zu Angular 2 beim dpunkt.verlag.

Wir wünschen Viel Spaß beim Kennenlernen von Angular 2.0 Alpha-46!

Über die Trainer

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

Gregor Woiwode ist als Softwareentwickler im Bereich des Competitive Intelligence bzw. Enterprise Knowledge Managements für ein Softwareunternehmen in Leipzig tätig. Er veranstaltet Trainings AngularJS. Er bloggt unter http://www.woiwode.info/blog/ .

.NET Developer Conference 2015

Inhalt

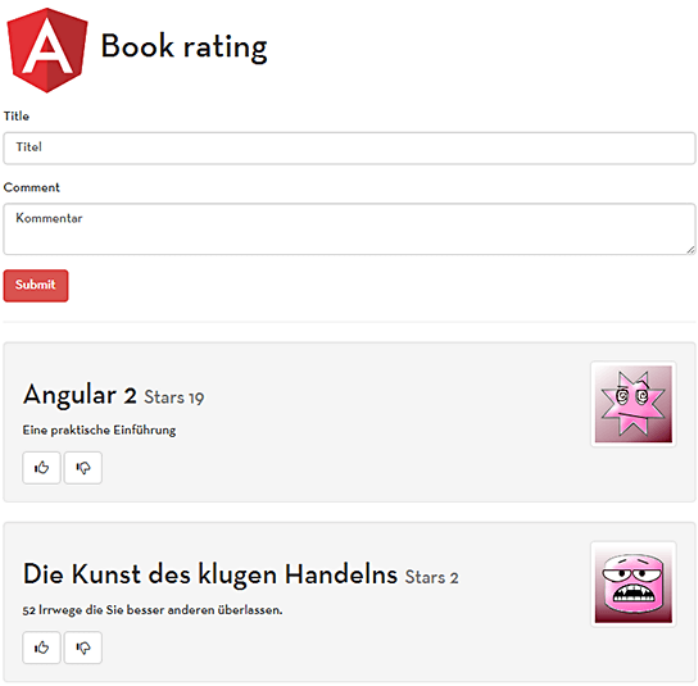
Quelltexte & Demos

- zu 1. https://github.com/Angular2Buch/angular2-module
- zu 2. https://github.com/Angular2Buch/template-syntax
- zu 3. https://github.com/Angular2Buch/angular2-testing

Folien zur "book-rating-app"

Während des Workshops entwickeln wir eine Anwendung mit Ihnen.
Zur Unterstützung sehen Sie Quelltext-Fragmente in einem Folienset.
Sie finden diese Folien unter:

https://github.com/Angular2Buch/presentations



Start: Angular 2.0 mit Atom nutzen

Ein schnelles Setup von Angular 2 wird im Folgenden beschrieben.

Atom und atom-typescript installieren

1. Atom installieren - Admin-Rechte sind übrigens nicht notwendig
2. atom-typescript installieren - entweder über die grafische Oberfläche (Settings > Install) oder in der Shell per `apm install atom-typescript`

Projekt anlegen

Der Aufbau orientiert sich am Angular 2 Quickstart. Wir werden drei Dateien erstellen:

1. `index.html`
2. `src/app/app.ts` - für das Bootstrapping
3. `src/app/MyAppComponent.ts` - eine Komponente, um auch etwas anzuzeigen

Wichtig ist die Tatsache, dass man alle TypeScript-Dateien in einen Unterordner legt (hier `app/`). So kann man die korrekte Dateiendung einstellen. Da unsere TypeScript-Dateien (`*.ts`) zu JavaScript-Dateien (`*.js`) transpiliert werden, geben wir dies als `defaultExtension: 'js'` entsprechend an.

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>Angular 2 Demo</title>

    <script src="https://code.angularjs.org/tools/system.js"></script>
    <script src="https://code.angularjs.org/2.0.0-alpha.46/angular2.dev.js"></script>

    <script>
      System.config({
        packages: { app: { defaultExtension: 'js'}}
      });
      System.import('./app/app');
    </script>

  </head>
  <body>
    <my-app>loading...</my-app>
  </body>
</html>
```

index.html

```
// app/app.ts
import { bootstrap } from 'angular2/angular2';
import MyAppComponent from './MyAppComponent';

bootstrap(MyAppComponent);
```

app/app.ts

```
// app/MyAppComponent.ts
import { Component, View } from 'angular2/angular2';

@Component({
  selector: 'my-app'
})
@View({
  template: '<h1>Hello {{ name }}</h1>'
})
export default class MyAppComponent {
  name: string;

  constructor() {
    this.name = 'Alice';
  }
}
```

app/MyAppComponent.ts

Transpilieren

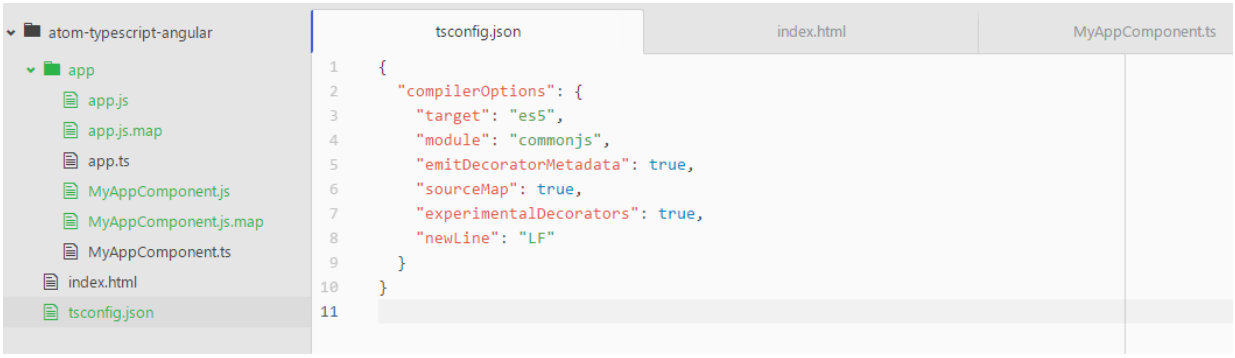
Würden wir jetzt schon den Entwicklungsstand kontrollieren, so würden wie eine Fehlermeldung erhalten. Die ganze Logik liegt nämlich nur in Form von TypeScript-Dateien vor. Der Browser soll jedoch JavaScript-Dateien laden und ausführen.

Die Erzeugung jener Dateien holen wir nach, indem wir eine Datei Namens `tsconfig.json` in das Projektverzeichnis einfügen. Hierfür kann man `ctrl(bzw. cmd) + shift + p` drücken (Liste der Kommandos) und den Befehl `tsconfig` auswählen ("TypeScript: Create tsconfig.json Project File").

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "emitDecoratorMetadata": true,
    "sourceMap": true,
    "experimentalDecorators": true,
    "newLine": "LF"
  }
}
```

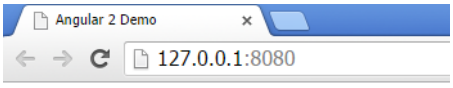
tsconfig.json

Sobald die Konfigurations-Datei vorliegt, wandelt Atom alle TypeScript-Dateien beim Speichern automatisch um.



Das Ergebnis lässt sich mit einem Webserver kontrollieren:

```
npm install -g live-server
live-server
```



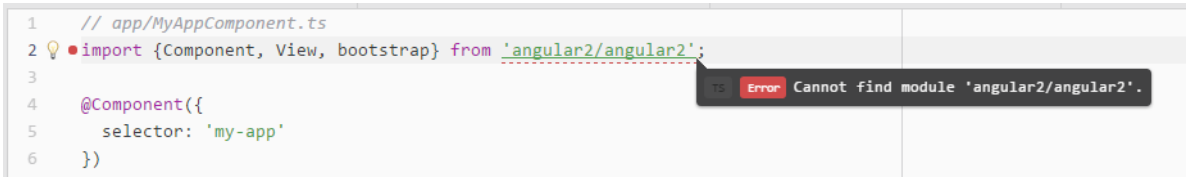
Hello Alice

Das war gar nicht schwer! :-)

Angular 2.0 Type Definitions

TypeScript wird die von uns erstellen TS-Dateien zwar transpilieren, jedoch erscheint ein Fehler, dass Angular nicht gefunden werden kann

(`TS: Error: Cannot find module 'angular2/angular2'`).



Ohne Typings kann der Compiler nicht die korrekte Verwendung der Angular-Types prüfen. Ebenso steht keine automatische Vervollständigung zu Verfügung. Die notwendigen Type Definitions von DefinitelyTyped sind leider derzeit nicht aktuell. Die beste Quelle von Angular 2.0 Type Definitions ist derzeit das NPM Paket. Wenn man schon mal dabei ist, kann man auch gleich noch SystemJS auf die Platte laden.

```
npm install angular2@2.0.0-alpha.46 systemjs@0.19.5
```

Ein wenig Bauchschmerzen macht mir übrigens die direkte Verwendung des `node_modules` Ordners. Es ist eine ewige Streitfrage, ob man diesen Ordner unter Versionsverwaltung stellt oder nicht. Den Ordner nun auch noch im Webserver verfügbar zu machen, ist eine neue Qualität. Ich bin damit noch nicht wirklich glücklich - aber das ist ein anderes Thema.

Man kann nun eine lokale Kopie der beiden Frameworks verwenden. Bei dieser Gelegenheit lassen wir auch die Einstellung für die `defaultExtension: 'js'` weg, da diese für ein definiertes Paket sowieso die Standardeinstellung ist (siehe #759).

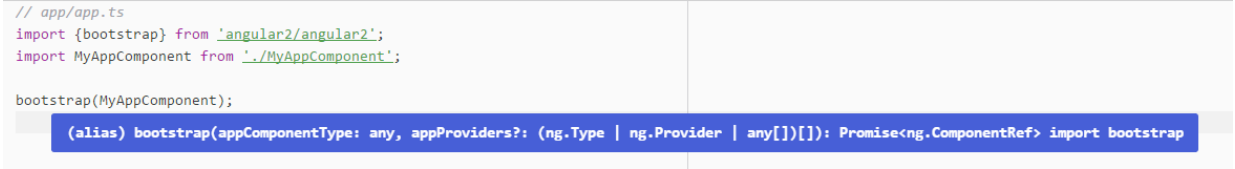
```
<!-- index_local.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>Angular 2 Demo</title>

    <script src="../node_modules/systemjs/dist/system.js"></script>
    <script src="../node_modules/angular2/bundles/angular2.dev.js"></script>

    <script>
      System.config({ packages: { app: {}}});
      System.import('./app/app');
    </script>

  </head>
  <body>
    <my-app>loading...</my-app>
  </body>
</html>
```

> index_local.html

Weiterhin kann man nun TypeScript die korrekten Pfade aufzeigen. Eine geniale Erfindung ist die `filesGlob`-Einstellung, welches man mit dem Snippet `fg` in die `tsconfig.json` einfügt:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "emitDecoratorMetadata": true,
    "sourceMap": true,
    "experimentalDecorators": true,
    "newLine": "LF"
  },
  "filesGlob": [
    "./**/*.ts",
    "../node_modules/angular2/angular2.d.ts",
    "../node_modules/angular2/bundles/typings/**"
  ],
  "files": [
     "-- wird automatisch gefüllt! --"
  ]
}
```

Das manuelle Bekanntmachen von Dateien über das `files`-Property oder über inline-Kommentare in den TS-Dateien (z.B. `/// <reference path="../node_modules/angular2/bundles/typings/angular2/angular2.d.ts"/>`) ist damit Geschichte! Atom hält die Liste der Dateien automatisch aktuell. Nun werden auch sofort die Typings von Angular 2.0 gefunden. Nun stehen Hilfen wie "Type information on hover", "Autocomplete", "Goto Declaration" und eine Reihe weiterer Features zur Verfügung.

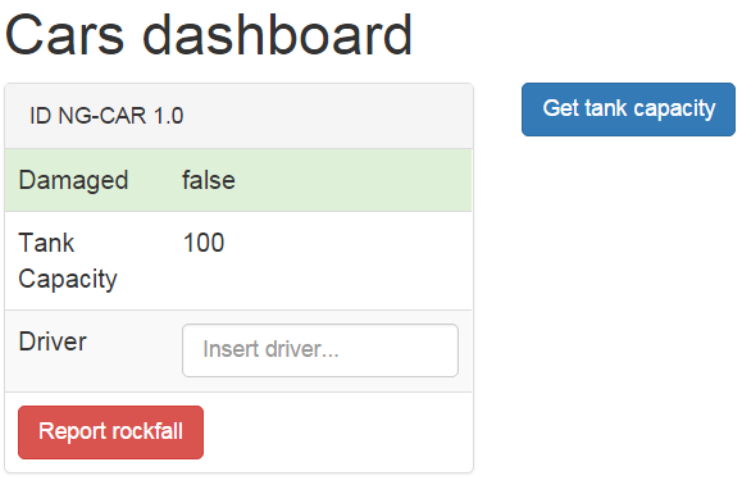


> Type information on hover

Angular Template-Syntax

Dependecy Injection gute Testbarkeit waren schon immer ein Alleinstellungsmerkmal für AngularJS. Mit der neuen Version wurden viele Details verbessert. Es wird Zeit tiefer in das Framework einzutauchen. Dieser Artikel stellt die neue Template-Syntax von Angular 2.0 vor. Es halten zahlreiche, neue Möglichkeiten Einzug, um Oberflächenelemente zu beschreiben. Die Entwickler von Angular verfolgen hierbei ein großes Ziel: Das Konzept der Template-Syntax eindeutiger und nachvollziehbarer zu formulieren, als es bei der Vorgängerversion der Fall ist.

Zur näheren Erläuterung wird ein Prototyp genutzt, der als Dashboard für Schäden an Autos dienen soll.



Neben einer ID und dem Schadensstatus kann auch der aktuelle Füllstand des Fahrzeugs abgefragt werden. Des Weiteren kann mit einem Klick ein Steinschlag (engl: "rockfall") gemeldet werden.

> **Übrigens** Sie finden das hier vorgestellten Beispiel auf GitHub unter: https://github.com/Angular2Buch/template-syntax

Components & Views

Angular 2 Anwendungen bestehen aus verschiedenen Komponenten (Components), die miteinander agieren können. Für das Dashboard wird eine Komponente benötigt. Im Dashboard wird eine Liste von Autos abgebildet werden. Das bedeutet, dass hierfür eine weitere Komponente implementiert wird.

Eine Angular 2.0 Komponente ist wie folgt aufgebaut.

```
// dashboard.component.ts
import { Component, View } from 'angular2/angular2';

@Component({ selector: 'dashboard' })
@View({
    template: `<p>{{ id }}</p>`
})
export default class DashboardComponent {
    id: string = 'NG-Car 2015';
}
```

Von Angular werden zunächst zwei Module `@Component()` und `@View()` importiert. Diese beiden Module sind im Speziellen TypeScript-Dekoratoren. Dekoratoren ermöglichen es Klassen zu
durch Meta-Angaben erweitern. `@Component()` spezifiziert, dass die Dashboard-Komponente über den `selector` <dashboard> im DOM des HTML-Dokuments eingesetzt wird.
Mit @View() definiert man das Template, das mit der Komponenten verknüpft ist. In diesem Beispiel wird das Feld `id`, aus der Klasse `DashboardComponent`, im Template gebunden und angezeigt. An dieser Stelle wird deutlich, was eine Komponente ist: Komponenten sind die neuen zentralen Bausteine von Angular 2.0. Sie übernehmen die Rolle von Direktiven und Controllern aus AngularJS.

> Eine Komponente ist ein angereichertes Template, das im Browser zur Anzeige gebracht wird. Das Template verfügt über ein spezifisches Verhalten, das in Angular 2.0 durch TypeScript-Dekoratoren beschrieben wird.

Interpolation

Wie wird nun aus dem Ausdruck `{{ id }}` ein angezeigter Text im Browser?
Bereits in AngularJS 1.x konnten Daten mithilfe zweier geschweifter Klammern an ein HTML Template gebunden werden. Der Wert wurde mittels Interpolation ausgewertet und angezeigt.
Dieses Konzept bleibt in Angular 2.0 erhalten.

```
<p>{{ id }}</p>
```

Diese Schreibweise ist eine Vereinfachung der tatsächlichen Syntax. Denn bevor dieses Template im Browser ausgegeben wird, setzt Angular diesen Ausdruck in ein Property-Binding um. [6]

```
<p [text-content]="interpolate(['Gregor'], [name])"></p>
```

Komponenten miteinander verknüpfen

Um in dem Dashboard nun ein Auto abbilden zu können wird eine weitere Komponente benötigt.

```
// car.component.ts
import { Component, View, Input } from 'angular2/angular2';

@Component({ selector: 'car' })
@View({
    template: `<p>{{ id }}</p>`
})
export default class CarComponent {
    @Input() id: string;
}
```

Im ersten Schritt soll diese Komponente lediglich die zugewiesene Identifikationsnummer ausgeben. Die @Input()-Dekorator bietet die Möglichkeit, Werte an die `CarComponent` zu übergeben. Näheres wird im folgenden Abschnitt erläutert.

Nun kann die CarComponent im Dashboard referenziert und im Template verwendet werden.

```
// dashboard.component.ts
import { Component, View } from 'angular2/angular2';
import { CarComponent } from '../car/car.component';

@Component({ selector: 'dashboard' })
@View({
    directives: [CarComponent],
    template: `<car [id]="id"></car>`
})
export default class DashboardComponent {
    id: string = 'NG-Car 2015';
}
```

Im Wesentlichen wurden drei Anpassungen vorgenommen.

1. Über ein weiteres import statement wird `CarComponent` geladen.
2. @View() wird durch die Eigenschaft `directives` ergänzt, damit `CarCompoennt` im Template verwendet werden kann.
3. Das Feld `id` wird and die gleichnamige Eigenschaft der `CarComponent` gebunden (Hierbei handelt es sich um ein Porperty-Binding).

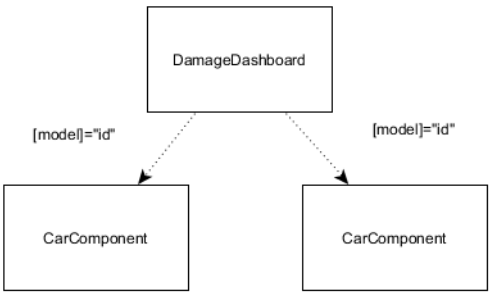
So wurde über die Datenbindung die erste Interaktion zwischen zwei Komponenten realisiert.

Input- und Output-Properties

Input- und Output-Properties sind Eigenschaften die die API einer Angular-Komponente beschreiben. Über Inputs werden Informationen an eine Komponente übergeben. Mit Outputs kommuniziert die Komponente Änderungen nach außen.
Inputs werden durch `Property-Bindings` beschrieben. Outputs können über `Event-Bindings` abonniert werden.

Property-Bindings

Mit Properties werden einer Komponente Daten übermittelt.



Property-Bindgins zeichnen sich durch eckicke Klammern aus (`[id]`)

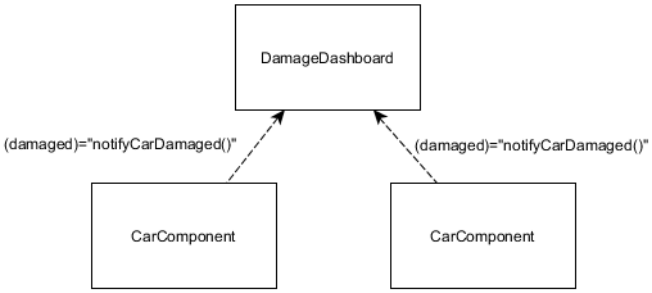
```
// dashboard.component.ts
<car [id]="id"></car>
```

Anstatt eckiger Klammern können Property-Bindings auch mit der vollständigen Syntax `bind-{property-name}="{expression}"` beschrieben werden.

```
// dashboard.component.ts
<car bind-id="id"></car>
```

Event-Bindings

Events bieten die Möglichkeit auf Veränderungen einer Komponente zu reagieren. Sie bieten einer Komponente die Möglichkeit mit ihrer Außenwelt zu kommunizieren.



Event-Bindings zeichnen sich durch runde Klammern aus (`(damaged)`). Sie triggern die Ausführung eines Statements.

```
<car (damaged)="report(damage)"></car>
```

Auch für diese Syntax existiert eine längere Syntax in der Form `on-{event-name}="{statement}"`:

```
<car on-damaged="report(damage)"></car>
```

Um solch ein Event aus einer Komponente heraus zu erzeugen, wird der Dekorator `@Output()` verwendet. Das dazugehörige Property ist ein `EventEmitter`, der Ereignisse auslösen kann.

```
// car.component.ts
import { EventEmitter } from 'angular2/angular2';

@Component({ /* ... */ })
class CarComponent() {
  @Input() id:string;
  @Output() damaged:EventEmitter = new EventEmitter();

  reportDamage() {
    // Event auslösen
    this.damaged.next(this.id);
  }
}
```

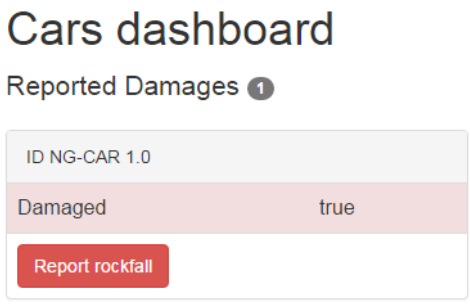
Neben der Verwendung runder Klammern, können Event-Bindings auch mit dem Ausdruck `on-{Event-Name}="{callback()}"` deklariert werden.

```
// dashboard.component.ts
<car on-damaged="report(damage)"></car>
```

In der Dashboard Komponente muss lediglich eine Methode ergänzt werden, die nach dem Ausläsen des Events `(damaged)`, ausgeführt wird.

```
// dashboard.component.ts
export default class DashboardComponent {
    /* ... */
    notifyCarDamaged() {
        this.totalDamages++;
    }
}
```

In diesem Fall wird im Dashboard die Anzahl der gemeldeten Schadensfälle zusammengezählt.



Two-Way Bindings mit `ng-model`

> **ACHTUNG** Um die Direktive `[(ng-model)]` zu verwenden, muss vorher das Modul { FORM_DIRECTIVES } importiert werden.

Aus Sicht einer Komponente werden mit Property-Bindings schreibende und den Event-Bindings lesende Operationen spezifiziert. Wie in AngularJS 1.x, ist es auch möglich Zwei-Wege-Bindungen (Two-Way-Bindings) zu realisieren. In der Template-Syntax von Angular 2.0 werden hierfür die Schreibweisen beider Binding-Arten kombiniert.

```
<input [(id)]="id">
```

Die eckigen Klammern legen fest, dass ein gegebener Wert an das <input>-Element gebunden wird. Die runden Klammern machen deutlich, dass Änderungen der Eigenschaft überwacht werden und diese mithilfe der Direktive `ng-model` in die Eigenschaft zurückschreiben werden.

Wie in den vorangehenden Beispielen gibt es auch hier eine alternative Schreibweise.

```
<input bindon-ng-model="id">
```

Die Zwei-Wege-Bindung lässt sich auch ohne `ng-model` realisieren. Das Markup wird so allerdings etwas komplexer.

```
<input
  [value]="id"
  (input)="id=$event.target.value">
```

Hierbei gibt `$event` Zugriff auf das auslösende Ereignis. Es ist ein natives Javascript-Event. Daher kann dessen API verwendete werden, um auf das betroffene Element zuzugreifen und dessen werden auszulesen (`id=$event.target.value`).

Template Referenzen

Innerhalb eines Templates können Referenzen auf HTML-Elemente, Komponenten und Datenbindungen erzeugt werden, um mit ihnen zu arbeiten.

```
<input #id type="text"/>
{{ id.value }}
```

> Das Binding {{ id.value }} macht deutlich, dass die lokale Referenz das HTML-Element referenziert und nicht nur dessen Wert.

Anstatt der # können lokale Refrenzen auch mit dem Prefix `var-` deklariert werden.

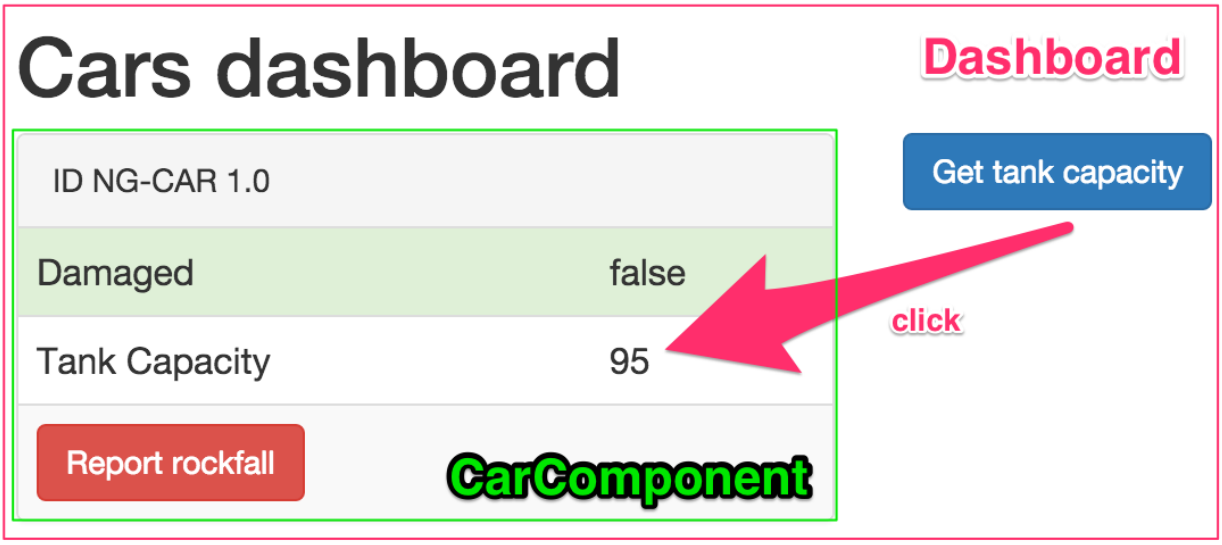
```
<input var-id type="text"/>
{{ id }}
```

Lokale Referenzen auf Komponenten unterscheiden syntaktisch nicht im Vergleich zu den HTML-ELementen. Zusätzlich können die Methoden der Komponente genutzt werden, um so mit ihr zu interagieren.

```
<car #car></car>
<button (click)="car.getTankCapacity()">Get tank capacity</button>
```

In diesem Fall wird bei einem Click-Event die Methode `getTankCapicity` ausgeführt.
In dieser Demo reduziert sich der Tankinhalt des Fahrzeug jedes Mal im 5%, wenn dieser angefragt wird.

```
// car.component.ts
export default class CarCmp {
  /* ... */
  getTankCapicity() {
    this.model.tankCapacity -= 5;
  }
}
```



Lokale Referenzen können auch auf Objekte zeigen. Im folgenden Beispiel wird der Platzhalte `#c` genutzt, um für jedes Element einer Liste `cars` die Komponente `Car` zu rendern (Hier lohnt sich ein Blick, in das erwähnte GitHub Code-Repository).

```
<car *ng-for="#c in cars" [model]="c">
```

> Bei dem Stern (`*`) vor der ng-for Direktive handelt es sich um eine Kurzschreibweise. Näheres erfahren Sie im nächsten Abschnitt.

* und <template>

Direktiven wie `ng-for`,`ng-if` und `ng-switch` werden zusammen mit einem Stern (`*`) verwendet. Diese Direktiven werden strukturelle Direktiven (Structural Directives) genannt, da sie DOM-Elemente hinzufügen oder entfernen.

```
<div *ng-if="totalDamages > 0">{{ totalDamages }}</div>
```

In diesem Beispiel wird die Anzahl aller gemeldeten Schäden nur dann im Dashboard angezeigt, wenn derenz Anzahl größer 0 ist.
Bei dem `*` handelt es sich, um eine Kurzschreibweise, die das Schreiben des Templates vereinfachen soll.

Sie wird als *Micro Syntax* bezeichnet, da Angular 2.0 diesen Ausdruck interpretiert und wieder in die uns bekannten Bindings umsetzt.
Beispielsweise ist auch folgende Verwendung der ng-if Direktive zulässig.

```
<template [ng-if]="totalDamages > 0">
  <div>{{ totalDamages }}</div>
</template>
```

Angular übersetzt die Micro Syntax in ein Property-Binding und umschließt das Template mit einem <template>-Tag. Dadurch entfällt der `*` , vor dem `ng-if`.[5]

Der Pipe-Operator `|`

Pipes korrespondieren zu `filters` aus AngularJS 1.x und werden genutzt, um Daten zu für die Anzeige zu transformieren. Sie nehmen Eingabeargumente entgegen und liefern das transformierte Ergebnis zurück.
In einem Binding-Ausdruck werden sie durch das Symbol `|` (genannt Pipe) eingeleitet.

```
/* Der Wert von name wird in Großbuchstaben ausgegeben */
<p>{{ id | uppercase}}</p>
```

Pipes können auch aneinander gehangen werden, um mehrere Transformationen durchzuführen.

```
<p>{{ id | uppercase | lowercase}}</p>
```

Der Elvis-Operator `?`

Die Bezeichnung "Elvis Operator" ist eine Ode an den Mythos, der sich damit befasst, ob Elvis tatsächlich tot ist oder nicht.

Der `?`-Operator ist ein nützliches Instrument, um zu prüfen, ob ein Wert `null` oder `undefined` ist. So können Fehlermeldungen bei der Template-Erzeugung vermieden werden.

```
<p>{{ car?.driver }}</p>
```

Hier wird geprüft, ob das Objekt `car` existiert. Wenn ja, wird der Namen des Fahrers ausgegeben. Der `?`-Operator funktioniert ebenfalls in komplexeren Objektbäumen.

```
<p>{{ car?.driver?.licences?.B1 }}</p>
```

W3C-Konformität

Auch wenn sich die Syntax zu Beginn ungewohnt ist, handelt es sich hierbei um valides HTML. [6, 8] In der HTML Spezifikation des W3C heist es:

> Attribute names must consist of one or more characters other than the space characters, U+0000 NULL, """, "'", ">", "/", "=", the control characters, and any characters that are not defined by Unicode.

Vollständiges Beispiel

```
import { Component, View , Input, Output } from 'angular2/angular2';
import { EventEmitter, FORM_DIRECTIVES }   from 'angular2/angular2';

@Component({ selector: 'car' })
@View({
  directives:[FORM_DIRECTIVES],
  template: `
  <p>ID {{ id | uppercase }}</p>
  <p>{{ tankCapacity }}</p>
  <input [(ng-model)]="id"
         placeholder="Change ID ...">
  <button (click)="rockfall()">Report rockfall</button>
  `
})
export default class CarComponent {
  @Input() id: string;
  @Input() tankCapacity: number;
  @Output() damaged: EventEmitter = new EventEmitter();

  rockfall() {
    this.damaged.next(this.id);
  }

  getTankCapacity() {
    this.tankCapacity = Math.floor(Math.random() * 100);
  }
}
```

[car.component.ts]

```
import { Component, View, NgIf } from 'angular2/angular2';
import CarComponent from '../car/car.component';

@Component({ selector: 'dashboard' })
@View({
  directives: [CarComponent, NgIf],
```

```
  template: `
    <p *ng-if="totalDamages > 0" class="lead">Reported Damages:{{ totalDamages }}</p>
    <car #car
         [id]="id" [tank-capacity]="tankCapacity"
         (damaged)="notifyCarDamaged()"></car>
    <button (click)="car.getTankCapacity()">Get tank capacity</button>
  `
})
export default class DashboardComponent {
  id: string;
  tankCapacity: number;
  totalDamages: number;

  constructor() {
    this.id = 'ng-car 1.0';
    this.tankCapacity = 100;
    this.totalDamages = 0;
  }

  notifyCarDamaged() {
    this.totalDamages++;
  }
}
```

[dashboard.component.ts]

```
import {bootstrap} from 'angular2/angular2';
import Dashboard   from './components/dashboard/dashboard.component';

bootstrap(Dashboard);
```

[app.ts]

```
<html>
  <head>
    <title>Demo | Template-Syntax</title>
    <script src="https://code.angularjs.org/tools/system.js"></script>
    <script src="https://code.angularjs.org/2.0.0-alpha.45/angular2.dev.js"></script>
    <script>
      System.config({ packages: { app: { defaultExtension: 'js'}}});
      System.import('./app/app');
    </script>
  </head>
  <body>
    <dashboard>loading...</dashboard>
  </body>
</html>
```

[index.html]

Zusammengefasst

- Input- und Output-Properties beschreiben die API einer Komponente
- Über Inputs "fließen" Daten in die Komponente hinein.
- Inputs werden über Property-Bindings aktualisiert ([property])
- Über Outputs "fließen" Daten aus der Komponente heraus.
- Outputs werden mithilfe von Event-Bindings abonniert ((event)).
- Ein Property-Binding und Event-Binding können kombiniert werden, um ein Two-Way-Binding zu beschreiben ([(twoWay)]). [2]

Polymer Webkomponenten nutzen

In AngularJS 1.x ist Entwicklungsaufwand nötig, um Webkomponenten anderen Bibliotheken integrieren zu können. Es müssen Direktiven geschrieben werden, um Angular die Statusänderungen der "Fremdkomponenten" mitzuteilen. [9]

Mit Angular 2.0 ist diese Arbeit nicht mehr nötig. Es wird nicht mehr unterschieden, ob es sich um ein natives Browserelement oder eine Web Component handelt. Angular hat nur Kenntnis davon, dass es an bestimmten Stellen im DOM Elemente instanzieren muss und es Eigenschaften schreiben, sowie Event-Listener erzeigen soll.

Das ermöglicht Beispielsweise die direkte Verwendung der Komponente `google-youtube` aus dem Polymer-Projekt. [10]

```
<google-youtube
  #player
  [video-id]="videoId">
</google-youtube>
<button (click)="player.play()"></button>
<button (click)="player.pause()"></button>
```

```
@View({ /* … */})
export default class DashboardComponent {
  /* ... */
  videoId: string;

  constructor() {
    /* ... */
```

```
    this.videoId = "ewxEFdMPMF0";
  }
```

Alle im Artikel beschriebenen Konzepte können hier nahtlos verwendet werden. Anhand der Online Dokumentation von `google-youtube` ist bekannt welche Eigenschaften und Aktionen zur Verfügung stehen [10]. Das Elementattribut `video-id` kann über ein Property-Binding gesetzt werden (`[video-id]`). Wird der Komponente eine gültige Id eines Videos von Youtube übergeben initialisiert sich der Video-Player selbstständig und kann verwendet werden.
Unter Verwendung der Referenz `#player` können die Aktionen der Webkomponente von anderen Webelementen gesteuert .

Angular stellt über die Template-Syntax eine einheitliche API zur Verfügung die auf jeder Web Component angewendet werden kann.

Fazit

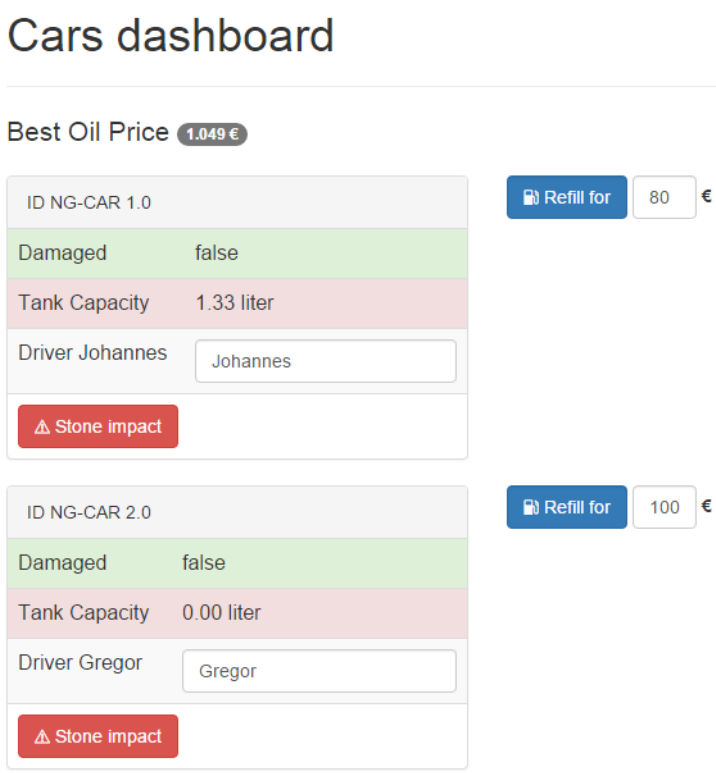
In Angular 2.0 wird die Template-Syntax in mehrere Konzepte aufgebrochen. Der Datenfluss zwischen Komponenten wird dadurch konkret definiert. Daher ist es mit einem Blick auf ein Template möglich, zu erkennen, wie sich eine Komponente verhält. Somit können, im Gegensatz zur Vorgängerversion AngularJS, Templates in Angular 2.0 diffiziler und genauer beschrieben werden.
Allerdings sind auch mehrere Möglichkeiten vorhanden Templates und Bindings zu definieren. Daher ist es ratsam, sich im Team auf jeweils eine der angebotenen Schreibweisen zu einigen, um ein vertrautes und homogenes Bild im Markup zu schaffen.

Dependecy Injection und Unit-Testing mit Angular 2.0

Dependecy Injection gute Testbarkeit waren schon immer ein Alleinstellungsmerkmal für AngularJS. Mit der neuen Version wurden viele Details verbessert. 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 war die Demo-Anwendung aus der letzten Ausgabe. Im Dashboard soll man nun Informationen zum aktuell günstigsten Benzinpreis erhalten.



> Screenshot: Mit dem günstigsten Benzinpreis die Autos betanken

Wenn man an einer beliegen 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();
}
```

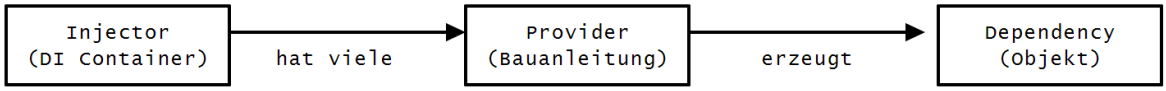
Dieses Vorgehen ist prinzipiell einwandfrei - nur stößt man mit zunehmender Menge an Code an eine Grenze. Der Code wird zunehmend unübersichtlicher, schwerer zu Warten und verweigert sich einem einfachen Test-Setup. Das Problem lässt sich dadurch begegnen, dass man die Verantwortung zum Erzeugen von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

Dies 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 in allen Programmiersprachen wieder. AngularJS zum Beispiel verwendet das Entwurfsmuster **"Dependency Injection"**. Ein Framework im Kern von AngularJS sorgt dafür, das die benötigte Abhängigkeit identifiziert wird und der Konstruktor-Funktion beim Aufruf bereit gestellt wird. In AngularJS 1.x kann man einen Service wie folgt anfordern:

```
// AngularJS 1.x
var Dashboard = ['GasService', function(GasService) {
  this.gasService = GasService;
}]
```

Wer das DI-Framework aus AngularJs 1.x kennt, der wird mit Sicherheit auch an dessen Grenzen gestoßen sein. Besonders hinderlich sind fehlende Namespaces und die Notwendigkeit, stets alle Abhängigkeiten per Name zu identifizieren. Dies ist doppelter Schreibaufwand. Im vorliegenden Beispiel muss man zum Beispiel zwei mal "GasService" schreiben.

Dependency Injection in Angular 2.0



> Die wichtigsten Bausteine für DI in Angular 2.0

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:

```
class GasService {
}

class Dashboard {
  constructor(@Inject(GasService) gasService) {
    console.log('Dependency:', gasService)
  }
}

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

> Listing 1: Constructor Injection mit ES6

Die Methode `resolveAndCreate()` erzeugt einen einsatzbereiten Injector. Die Methode akzeptiert ein Array aus Typen oder Providern. Wird nur ein Typ übergeben, so wird ein entsprechender Provider für diesen Typ erzeugt (`useClass` - wird später erläutert). Sofern man TypeScript einsetzt, kann man die Schreibweise noch etwas mehr vereinfachen. Durch die Verwendung von Typen, kann man auf den Decorator `@Inject` verzichten:

```
@Injectable()
class Dashboard {
  constructor(gasService: GasService) {
    console.log('Dependency:', gasService)
  }
}
```

> Listing 2: Constructor Injection mit TypeScript

Damit dieses Beispiel funktioniert, muss TypeScript einen Hinweis dazu erhalten, dass die Konstruktor mit Dekoratoren versehen werden soll. Dies geschieht mit dem Dekorator `@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 Dekoratoren sind etwa `@Component()`, `@View()` oder `@RouteConfig()`. Da Angular 2.0 stark auf einen deklarativen Stil mittels Dekoratoren setzt, benötigt man `@Injectable()` eigentlich nur für eigene Service-Klassen.

Der Injector versteht eine Reihe von Baunleitungen. 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 auflösen (**`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 Angular 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]})
]);
```

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. [1] Weitere Beispiele für die unterschiedlichen Verwendungen von `provide` finden Sie in den Codebeispiele zum Artikel (Datei: `Injector_tests.ts`).

Durchstarten

Die Methode `resolveAndCreate()` 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.

```
// app.ts
import {bootstrap} from 'angular2/angular2';
import Dashboard from './components/dashboard-component';

bootstrap(Dashboard);
```

> Listing X: Starten (bootstrapping) der Anwendung

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:

```
// app.ts
import {bootstrap} from 'angular2/angular2';
import Dashboard from './components/dashboard-component';
import GasService from './models/gas-service';

bootstrap(Dashboard, [GasService]);
```

> Listing X: Bootrapping mit Registrierung der Dependency GasService

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 wieder herstellen. 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.

```
{
  [...]

  "devDependencies": {
    "karma": "^0.13.15",
    "karma-chrome-launcher": "^0.2.1",
    "karma-jasmine": "^0.3.6"
  },
  "scripts": {
    "start": "live-server",
    "test": "karma start"
  }
}
```

> Listing X: Auszug aus der `package.json`

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 paar Pfade gemappt werden (siehe 1. Artikel). An dieser Stelle ist das Setup zum aktuellen Stand (Alpha-46) noch etwas unkomfortabel. Wir empfehlen Ihnen aktuell den "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 diesem 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 dieser Datei den Befehl `System.config()`. SystemJS haben wir bereits im 1. Artikel (Ausgabe 12/2015) kennen gelernt.

Unit-Tests mit Jasmine

Die Auswahl eines geeignete 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. X veranschaulicht die Syntax.

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

  var number;
```

```
  beforeEach(() => {
    number = 1;
  });

  it("contains spec with an expectation", () => {
    expect(number).toBeGreaterThan(0);
  });
});
```

> Listing Nr. X: Hello World mit Jasmine

Infobox: Die wichtigten Methoden in 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 die selben 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 Abhängkeiten in Tests vereinfacht und das Testen von asynchronem Code ermöglicht. Zudem erhält man auch gleich die passenden TypeScript type definitions:

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

Listing X: dies Zeile sollte in keinem Test fehlen

InfoBox: 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 X beweist zum Beispiel, dass die Dashbard-Komponente stets mit einem gefüllten Array initialisiert wird.

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

> Listing X: Verwendung von `beforeEachProviders()` und `inject()`

Beachten Sie auch die Verwendung der Methode `inject()`. Sie ist dazu gedacht, in einer `beforeEach()` oder `it()` eine Abhängigkeit anzufordern. Im den 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) => { ... }
```