Rich Internet Applications with JavaScript – Angular

12. Mai 2017

Studienarbeit im Fach Aktuelle Technologien zur Entwicklung Verteilter Java Anwendungen

Autor:
Sebastian Lausch
Student Informatik B.Sc.
slausch@hm.edu

Dozent:
Michael Theis
Lehrbeauftragter HM
michael.theis@hm.edu

Inhaltsverzeichnis

1 Einleitung **3**

2 Einführung in Angular **4**
2.1 Angular Modules 5
2.2 Components 6
2.3 Templates 8
2.4 Metadata 10
2.5 Data Binding 10
2.6 Directives 11
2.7 Services 12
2.8 Dependency Injection 12
2.9 Zusammenfassung 13

3 Stärken von Angular **15**

4 Schwächen von Angular **17**

5 Fazit **18**

Abbildungsverzeichnis **19**

Literaturverzeichnis **20**

Erklärung zur eigenständigen Anfertigung

Hiermit erkläre ich, dass ich die vorliegende Seminararbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Seminararbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

München, 12. Mai 2017

Kapitel 1
Einleitung

Rich Internet Applications

Unter Rich Internet Applications, kurz RIA (zu deutsch "reichhaltige Internet Anwendung") versteht man für gewöhnlich Internetanwendungen welche ihrem Benutzer über eine graphische Benutzeroberfläche eine große Anzahl von Möglichkeiten zur Interaktion bieten und mit dem Internet kommunizieren. Der Begriff einer RIA ist damit nicht eindeutig definiert und kann im Rahmen der genannten Eigenschaften unterschiedlich verwendet werden.
Auch wenn die Beschreibung von RIAs nicht die Technologie mit der sie geschrieben werden vorgibt werden werden für die Entwicklung von RIAs oft auf JavaScript Frameworks zurückgegriffen. Diese erleichtern die Entwicklung erheblich. Zusätzlich unterstützen alle modernen Browser JavaScript ohne die Einbindung zusätzlicher Plugins. Ein beliebtes Framework, das auch in der vorliegenden Arbeit verwendet wird, ist Angular.[1]

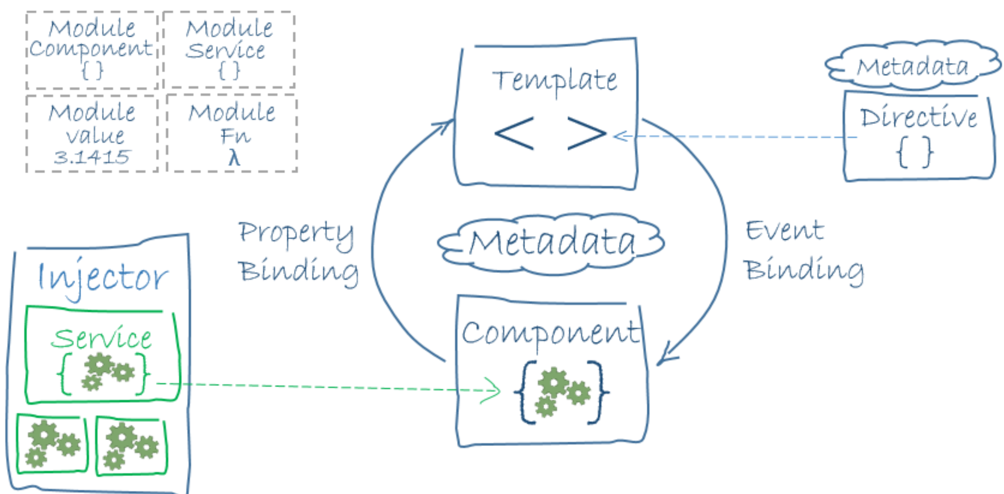
Thema dieser Arbeit

Die vorliegende Arbeit beschäftigt sich mit der Entwicklung einer Rich Internet Application unter Verwendung des Angular Frameworks. Die verwendete major Version des Frameworks ist 4.0 welche am 23. März 2017 veröffentlicht wurde. [2] Ziel der vorliegenden Arbeit ist es die Konzepte und Grundlagen von Angular zu vermitteln und die Stärken und Schwächen des Frameworks aufzuzeigen. Im nächsten Kapitel werden die grundlegenden Konzepte von Angular erläutert und eine Einführung in das Framework geboten. Anschließend beschäftigt sich dieses Dokument mit den Stärken von Angular. Im darauf folgenden Kapitel werden die Schwächen des Frameworks thematisiert. Abschließend folgt ein Fazit über die Verwendung von Angular für die Entwicklung einer Rich Internet Application.

[1] Vgl. [RIA-Wiki]
[2] Vgl. [CLogAng]

Kapitel 2
Einführung in Angular

Abbildung 2.1: Übersicht der Architektur von Angular 4.0



vgl. https://angular.io/resources/images/devguide/architecture/overview2.png; Google CC 4.0

Das Übersichtsdiagramm zeigt die wesentlichen Bestandteile einer Angularapplikation und deren Zusammenspiel. Im folgenden Abschnitt werden die einzelnen Teilkomponenten erklärt und anhand eines Beispielprojektes deren Funktionsweise verdeutlicht. Bei dem Beispielprojekt handelt es sich um eine übersichtliche Webapplikation die Helden verwaltet. Die *Tour of Heroes* ist ein Beispielprojekt, dass Angular als Tutorial anbietet und das den Einstieg in das Framework und TypeScript vereinfachen soll. Das Projekt des Tutorials wurde leicht angepasst und entspricht nicht vollständig den Beispielen auf der Website des Tutorials. Ein Link auf das Angular Tutorial und eine Live-Demonstration des Projekts ist im Literaturverzeichnis des Dokuments zu finden.[1]

[1] Vgl. [DokAngCode] und [DokAngTut]

2.1 Angular Modules

Die Module in Angular sind ein grundlegender Bestandteil des Frameworks. Sie werden auch NgModules genannt und sind nicht mit den JavaScript Modules zu verwechseln. NgModules sind ein eigenes Konstrukt und unabhängig von JavaScript Modules. Zu einer Verwechslung kann es leicht kommen, da sowohl NgModules als auch JavaScript Modules in einer Angularanwendung verwendet werden. Die NgModules sind an ihrem eindeutigen Decorator erkennbar (@NgModel) und somit von den JavaScript Modules zu unterscheiden. Jede Angularanwendung hat mindestens ein NgModule, das sog. Rootmodule. Das Rootmodule wird meist *AppModule* genannt und legt den Aufbau der Anwendung fest. Kleine Anwendungen, haben oft nur dieses eine Modul. Größere Anwendungen können z.B. Fachbereiche über Module auftrennen.
In unserem Beispiel sieht das Rootmodule wie folgt aus.

```
1  @NgModule({
2   declarations: [
3    AppComponent,
4    HeroDetailComponent,
5    HeroesComponent,
6    DashboardComponent,
7    HeroSearchComponent
8   ],
9   imports: [
10   BrowserModule,
11   FormsModule,
12   HttpModule,
13   InMemoryWebApiModule.forRoot(InMemoryDataService),
14   AppRoutingModule
15  ],
16  providers: [ HeroService ],
17  bootstrap: [AppComponent]
18 })
19 export class AppModule { }
```

Listing 2.1: Code - AppModule

Wie man erkennen kann, werden nahezu alle Konfigurationen über den Decorator *NgModule* gemacht. *Declarations* legen fest, welche Klassen zu diesem Modul gehören. Für dieses Beispiel werden nur ein paar Komponenten (s. 2.2) benötigt um die Webapplikation darzustellen.
Imports gibt die Abhängigkeiten von anderen Modulen an. In diesem Beispiel benötigen wir für die Darstellung im Browser die Module *BrowserModule, FormsModule* und *HttpModule*. Das *InMemoryWebApiModule* wird für die Simulation einer entfernten API benötigt. Das *AppRoutingModule* ist ein

weiteres NgModule und wird für das Routing auf der Website des Beispielprojektes benötigt.

```
1  const routes: Routes = [
2      {path:'', redirectTo:'/dashboard', pathMatch:'full'},
3      {path:'heroes', component: HeroesComponent},
4      {path:'dashboard', component: DashboardComponent},
5      {path:'detail/:id', component: HeroDetailComponent}
6  ];
7
8  @NgModule({
9      imports:[RouterModule.forRoot(routes)],
10     exports:[RouterModule]
11 })
```

Listing 2.2: Code - AppRoutingModule

Es enthält lediglich eine Routingtabelle *routes* welche den einzelnen Komponenten bzw. Seiten der Webapplikation URLs zuteilt. Die Routingtabelle hätte auch in *AppModule* deklariert werden können. Die Trennung von Routinglogik und Applogik ist jedoch empfehlenswert um die Trennung von Fachlichkeit aufrecht zu erhalten.
Unter *providers* werden Services (s. 2.7) die allen anderen Teilen der Applikation bekannt sein sollen festgelegt. Diese Services stehen somit auch anderen Modulen zur Verfügung.
Die *bootstrap*-Eigenschaft sollte nur in einem Modul gesetzt werden. Die hier gesetzte Komponente dient als Host für alle Views der restlichen Komponenten. Es ist vergleichbar mit einer Startseite. In dem Beispielprojekt ist dies *AppComponent*.[2]

2.2 Components

Die *Components* sind ein weiterer zentraler Bestandteil einer Angularapplikation. Components definieren die *Views* für unterschiedliche Sichten auf die Applikation. Sie sind damit in Kombination mit Templates (s. 2.3) für die gesamte Benutzererfahrung verantwortlich und sind der einzige Teil, mit dem der Benutzer direkt in Berührung kommt. Der Aufbau von Components soll hier am Beispiel der *HeroDetailComponent*, also der Detailansicht eines Helden erläutert werden.

```
1  @Component({
2      selector: 'hero-detail',
3      templateURL:'./hero-detail.component.html',
```

[2] Vgl. Bereich “Modules“ in [DokAngArch]

```
4     styleURLs:['./hero-detail.component.css']
5 })
6 export class HeroDetailComponent implements OnInit{
7  @Input() hero: Hero;
8
9  ngOnInit():void{
10  this.route.params.switchMap((params:Params)=>this.
       heroService.getHero(+params['id']))
11  .subscribe(hero=>this.hero = hero);
12  }
13  save():void{
14  this.heroService.update(this.hero)
15  .then(()=> this.goBack());
16 }
```

Listing 2.3: Code - HeroDetailComponent gekürzt

Wie man bereits bei den Modulen gesehen hat, wird auch hier ein Decorator verwendet um Components eindeutig zu markieren. Der Decorator @Component legt unter anderem die Eigenschaften *selector, templateURL und styleURLs* fest. Der Selector ist dabei ein eindeutiger Bezeichner für diese Komponente und dient zum Einbetten der Komponente in dem HTML-Code einer anderen Komponente. Die URL-Felder geben an in welchen Dokumenten der HTML-Code und wo der CSS-Code liegen. Weiter werden innerhalb der Component Methoden definiert. Diese Methoden repräsentieren dabei die Logik der View. Ein Beispiel ist die Methode *save()*. Die Hauptoperation wird in eine Serviceklasse ausgelagert. Der Aufbau dieser Methode ist typisch für Component-Methoden. Sie sollen möglichst wenig Logik enthalten und alle nicht-trivialen Aufgaben in andere Teile der Angularapplikation auslagern.
Neben einer Methode wie *save()*, die durch eine Nutzeraktion aufgerufen wird, gibt es auch sog. *Lifecycle-hooks*. Das sind Methoden, die beim Übergang der Component in einen anderen Lebenszyklus ausgeführt werden. Im vorliegenden Beispiel ist die Methode *ngOnInit()* implementiert, die bei Erstellung der Component ausgeführt wird. In diesem Fall setzt die Methode bei Erstellung der Component eine Route, damit die URL im Browser dem ausgewählten Hero entspricht. Wählt man beispielsweise den Hero *Mystique* mit der Id 16 aus, kann man die URL */detail/16* im Browser sehen. Die Lifecyclehooks sind notwendig, da Components je nach Bedarf des Benutzers einer Angularapplikation dynamisch erstellt und gelöscht werden. Die hooks sind damit der einzige Mechanismus mit denen der Entwickler auf diese dynamischen Veränderungen Einfluss nehmen kann.[3]

[3] Vgl. Bereich "Components" in [DokAngArch]

2.3 Templates

Templates definieren die sichtbare View einer Component. Hier wird festgelegt wie der Nutzer die Webapplikation sieht. Ein Template hat den gleichen Aufbau und eine ähnliche Syntax wie ein HTML Dokument. Allerdings gibt es einige Unterschiede zum regulären HTML.

```
1  <div *ngIf="hero">
2    <h2>{{hero.name}} details!</h2>
3    <div>
4      <label>id: </label>{{hero.id}}
5    </div>
6    <div>
7      <label>name: </label> <input [(ngModel)]="hero.name"
8        placeholder="name" />
9    </div>
10   <button (click)="save()">Save</button>
11   <button (click)="goBack()">Back</button>
12 </div>
```

Listing 2.4: Code - HeroDetailComponent HTML

Abbildung 2.2: Hero Detail - Mystique

Tour of Heroes
Dashboard Heroes
Mystique details!
id: 16
name: Mystique
Save Back

Wie in diesem Template für die Detailansicht eines Hero zu sehen ist, gibt es Codeschnipsel, die man in regulärem HTML nicht findet. So wird z.B. *ngIf* in Zeile 1 verwendet. *ngIf* ist Bestandteil der Angular Template Syntax und bedeutet, dass der darauf folgende Code nur angezeigt wird, falls ein hero ausgewählt ist.

```
1 <h3>Top Heroes</h3>
2 <div class="grid grid-pad">
```

```
3     <a *ngFor="let hero of heroes" [routerLink]="['/detail',
        hero.id]" class="col-1-4">
4       <div class="module hero">
5         <h4>{{hero.name}}</h4>
6       </div>
7     </a>
8   </div>
9   <hero-search></hero-search>
```

Listing 2.5: Code - DashboardComponent HTML

Abbildung 2.3: Tour of Heroes - Dashboard

In diesem Codeausschnitt sieht man ein weiteres Feature der Angular-Template-Syntax. In der letzten Zeile wird einfach *<hero-search></hero-search>* als HTML fremdes Tag eingebettet. Das bewirkt, dass in der Dashboardansicht an dieser Stelle der passende HTML-Code von HeroSearch eingeblendet wird. Als Nutzer bemerkt man diesen Unterschied nicht. Der Entwickler hat jedoch den Vorteil den HTML-Code von verschiedenen Komponenten mehrfach zu verwenden und die einzelnen Bereiche auch im Code sauber voneinander trennen kann. Außerdem kann man seine Webapplikation auf diese Weise modular gestalten und Blöcke mit wenig Aufwand beliebig vertauschen. [4]

[4] Vgl. Bereich “Templates“ in [DokAngArch]

2.4 Metadata

Metadaten sind Daten, welche Angular informieren, wie mit einer definierten Klasse zu verfahren ist. Damit Angular mit TypeScriptklassen arbeiten kann braucht es weitere Informationen, welche über Decorators angefügt werden. So kann mit dem Decorator *@Component* eine Klasse als Component definiert werden. Über die Eigenschaften des Decorator können weitere Einstellungen vorgenommen werden, die die Verhaltensweise der dekorierten Klasse beeinflussen. Einige dieser Eigenschaften wurden bereits in 2.2 vorgestellt. Weitere Beispiele für oft verwendete Decorators sind *@Injectable, @Input* und *@Output*.[5]

2.5 Data Binding

Data Binding ist ein Mechanismus von Angular, der dem Programmierer erheblich Arbeit abnimmt. Das Data Binding kümmert sich um die Kommunikation zwischen einer Component und dem dazugehörigen Template. Durch das Einfügen von *binding markup* werden Component und Template verbunden. Dabei gibt es 3 verschiedene Arten von Databinding. Von einer Component zum Domain Object Model (kurz DOM), vom DOM zu einer Component oder bidirektional.

```
1  <div *ngIf="hero">
2    <h2>{{hero.name}} details!</h2>
3    <div>
4      <label>id: </label>{{hero.id}}
5    </div>
6    <div>
7      <label>name: </label> <input [(ngModel)]="hero.name"
8        placeholder="name" />
9    </div>
10   <button (click)="save()">Save</button>
11   <button (click)="goBack()">Back</button>
12 </div>
```

Listing 2.6: Code - HeroDetailComponent HTML

In diesem Beispiel sehen wir zwei dieser Arten. In Zeile 2 und 4 wird über *{{hero.id}}* bzw. *{{hero.name}}* ein Wert aus der Component abgefragt und in den HTML-Code eingebettet. Diese Vorgehensweise nennt man *interpolation*. Das *property binding* ist eine Alternative zu *interpolation*.

```
1  <p><img src="{{heroImageURL}}"> is the <i>interpolated</i>
     image.</p>
```

[5] Vgl. Bereich “Metadata“ in [DokAngArch]

```
2 <p><img [src]="heroImageURL"> is the <i>property bound</i>
    image.</p>
```

Beide Schreibweisen führen zum gleichen Ergebnis. Aufgrund der Lesbarkeit ist aber für gewöhnlich *interpolation* zu bevorzugen.
In Zeile 10 und 11 wird ein Event Handler ausgeführt. Bei einem Klick auf den *Save*-Button wird die Methode *save()* in der Component ausgeführt. Analog dazu *back()* in Zeile 11. Es handelt sich um *event binding*.
Eine weitere Möglichkeit ist das *Two-Way-Databinding*. Es ist eine Kombination aus *event binding* und *interpolation* bzw. *property binding*. In Zeile 7 wird die Notation *<input=[(ngModel)]=hero.name>* verwendet. Durch das Two Way Data Binding wird der Inhalt des Feldes zunächst mit einem Wert aus der Component gefüllt (*property binding*). Ändert der User den Wert, so wird der neue Wert an die Component übermittelt (*event binding*).
Bindings dienen somit der Kommunikation zwischen Template und Component und erleichtern einem Entwickler die Arbeit.[6]

2.6 Directives

Directives sind Klassen mit dem @Directive Decorator. Sie geben Angular Anweisungen wie die dynamischen Templates im DOM aussehen sollen. Dabei sind Components eine Unterklasse von Directives. Da Components aber ein essentieller Bestandteil von Angular sind, werden sie getrennt von Directives betrachtet.
Es gibt zwei verschiedene Arten von Directives. Die *Structural Directives* und *Attribute Directives*. Structural Directives verändern das Aussehen einer Seite indem sie Elemente im DOM hinzufügen, verändern oder ausblenden. Ein Beispiel für eine solche Directive ist in HeroDetailComponent2.5 in Zeile 2 zu sehen. **ngIf="hero"* zeigt den restlichen HTML-Code des div nur an wenn ein hero ausgewählt wurde. Andernfalls wird der HTML-Code ausgeblendet.
Attribute Directives verändern das Aussehen oder das Verhalten eines existierenden Elements. Meist sehen sie im Code wie normale HTML Attribute aus. Ein Beispiel dafür ist in HeroDetailComponent2.5 in Zeile 7 zu finden. *<input=[(ngModel)]=hero.name>* ist eine Directive, die das Two Way Data Binding implementiert.
Angular bietet noch ein paar mehr Directives. Allerdings können Directives auch selbst geschrieben werden.[7]

[6] Vgl. Bereich "Data binding" in [DokAngArch]
[7] Vgl. Bereich "Directives" in [DokAngArch]

2.7 Services

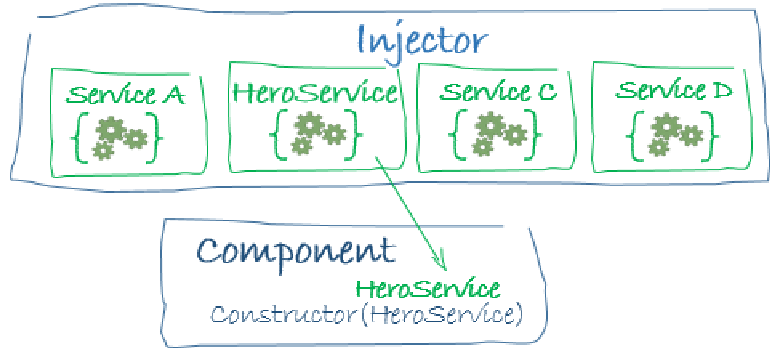
Services sind ein weiterer grundlegender Bestandteil der Angulararchitektur. Es ist nicht genau festgelegt, was oder wie ein Service in Angular arbeitet. Meist sind Services Klassen, die nur eine spezielle Aufgabe haben, die sie erfüllen müssen. In Angularanwendungen wird mit Services die komplexe Logik der Anwendung abgebildet. Components sind dabei lediglich die Vermittlungsschicht zwischen der View und dem Service. Sie lagern alles was nicht trivial ist an die Services aus. Im Modell einer 3-Schichten-Architektur wären Services im Businesslayer anzusiedeln.
In dem Beispielprojekt wird die Bereitstellung der Hero-Daten in einen Service ausgegliedert. Der HeroService bietet Components eine Anzahl von Methoden, mit denen die Daten von Heroes erfragt, verändert oder gelöscht werden können. Wie diese Daten manipuliert und gespeichert werden ist für die aufrufende Component intransparent. Diese Vorgehensweise bietet die Möglichkeit die Arbeitsweise von Services zu verändern ohne dass die Components angepasst werden müssen. So könnte der HeroService die Daten lokal aus einem Dokument auslesen oder auch von einer externen Ressource anfragen.
Damit Components auf einen Service zugreifen können wird dieser im Rootmodule unter *providers* eingetragen. Somit kann jede Klasse der Applikation auf die Methoden des HeroService zugreifen.
Das Prinzip der Trennung von Logik in Services und View in Components wird von Angular nicht erzwungen. Allerdings sollte man sich an dieses Designprinzip halten um wartbaren, übersichtlichen Code zu bekommen.[8]

2.8 Dependency Injection

Die Dependency Injection ist ein Mechanismus der eine neue Instanz einer Klasse mit allen notwendigen Abhängigkeiten versorgt. Für gewöhnlich werden dabei die Abhängigkeiten von Components zu Services aufgelöst. Die Liste der benötigten Injectables wird dabei aus der Parameterliste des Konstruktors der Component ausgelesen. Wird eine neue Instanz einer Component angelegt, fragt Angular den *Injector* nach den benötigten Services. Der Injector verwaltet einen Container mit allen *Injectables* und gibt die verlangten Services zurück.

[8] Vgl. Bereich “Services“ in [DokAngArch]

Abbildung 2.4: Prozess einer Dependency Injection von HeroService



vgl. https://angular.io/resources/images/devguide/architecture/injector-injects.png; Google CC 4.0

Sollte der Injector keine Instanz des angefragten Injectables haben, wird diese Instanz erstellt.

```
1 @Injectable()
2 export class HeroService {
3   //...all Methods of the service are defined here.
4 }
```

Listing 2.7: HeroService - Erstellung von HeroService

```
1 @NgModule({
2   declarations: [ ... ],
3   imports: [ ... ],
4   providers: [ HeroService ],
5   bootstrap: [ ... ]
6 })
7 export class AppModule { }
```

Listing 2.8: AppModule - Eintragen von HeroService in Injector

Damit der Injector alle Injectables kennt und diese erstellen kann, werden ihm diese über den Parameter providers[ ] bekannt gemacht. Damit alle Components innerhalb einer Applikation auf dieselbe Instanz eines Injectables zugreifen, werden die Injectables im Rootmodule eingetragen.[9]

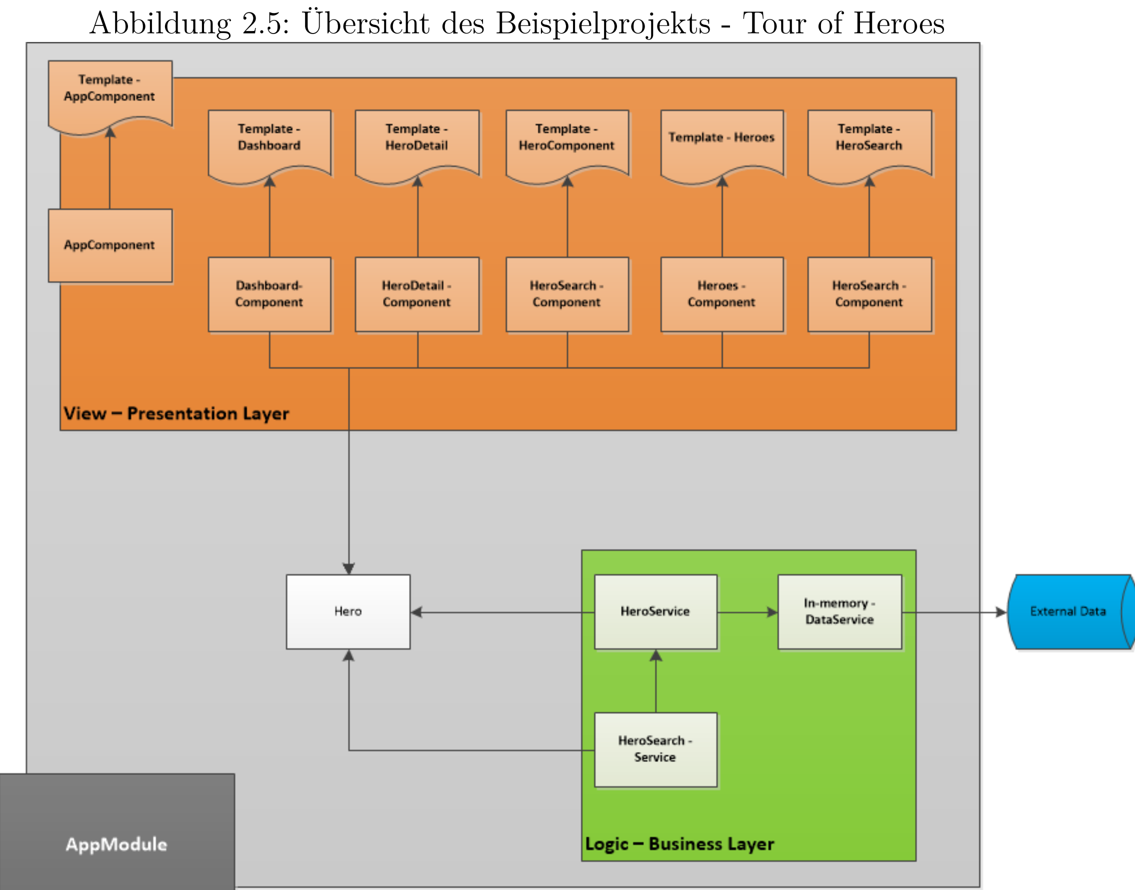
2.9 Zusammenfassung

In diesem Kapitel hat man einen Überblick über die Teilkomponenten von Angular bekommen und wie sie verwendet werden. An dem Beispielprojekt kann man nun die einzelnen Teile identifizieren und sortieren.

[9] Vgl. Bereich “Dependency Injection“ in [DokAngArch]

Auf der Abbildung kann man sehen, wie das AppModule als Wurzel alle anderen Teile der Applikation verwaltet. Die Services beinhalten die komplexe Logik der Anwendung. Die Components verwalten ihr Template und nutzen die Services um nicht triviale Aufgaben zu erledigen. Die AppComponent dient als Startpunkt für alle View-Elemente (s. bootstrap[ ] in AppModule). In einer größeren Anwendung würde dies nur ein Modul von vielen sein, welche alle eine eigene Fachdomäne repräsentieren. Auch in der Beispielanwendung gibt es ein nicht weiter besprochenes Angularmodule. Das Routingmodule ist ein eigenständiger Bereich in dem die Routen der Applikation verwaltet werden. Eine Route in der Applikation gibt unter anderem den Aufbau der URLs vor und dient zum Verlinken der einzelnen Anwendungsbereiche.
Möchte man einen Eindruck von der vorgestellten Beispielapplikation erhalten, kann man es unter [DokAngCode] als live Demonstration sehen.

Abbildung 2.5: Übersicht des Beispielprojekts - Tour of Heroes



Kapitel 3
Stärken von Angular

Plattformunabhängigkeit

Ein großer Vorteil von Angularanwendungen ist ihre Plattformunabhängigkeit. Das Framework sorgt für die dynamische Anpassung der Applikation auf allen Endgeräten. Der Entwickler kann sich voll auf das Programmieren der Logik konzentrieren und muss sich keine Gedanken über die Darstellung auf unterschiedlichen Endgeräten machen. Das hat außerdem den Vorteil, dass eine Applikation nur einmal geschrieben werden muss und nicht einmal für Desktopsysteme mit großem Bildschirm und einmal für mobile Geräte mit geringer Bildschirmdiagonale.
Angular nimmt dem Entwickler auch an dieser Stelle ein erhebliches Maß an Arbeit ab.

Breite Unterstützung neuer Browser

Angular wird von allen modernen Browsern unterstützt. Moderne Browser sind dabei die sog. Evergreenbrowser, die ständig mit Updates versorgt werden und bei einem Versionssprung nicht neu installiert werden müssen.

Chrome	Firefox	Edge	IE	Safari	iOS	Android	IE Mobile
latest	latest	14	11	10	10	Marshmallow (6.0)	11
		13	10	9	9	Lollipop(5.0, 5.1)	
			9	8	8	KitKat(4.4)	
				7	7	Jelly Bean(4.1, 4.2, 4.3)	

Abbildung 3.1: Angular unterstützende Browser

Diese Evergreenbrowser wie z.B. Mozilla Firefox oder Google Chrome unterstützen Angular nativ. Andere Browser wie die verschiedenen Versionen des Internet Explorer benötigen sog. Polyfills um die Inhalte darzustellen. Einige Funktionen können trotz Polyfills nicht benutzt werden.

Aufbau nach Model View Controller

Wie bereits im vorherigen Kapitel (s. 2.9) erläutert wurde, unterstützt Angular die Trennung von View und Business Layer durch Components und Services. Dieser modulare Aufbau orientiert sich dabei an der weit verbreiteten Model-View-Controller-Architektur (kurz MVC). Das bietet den Vorteil, dass die Logik der Angularapplikation unabhängig von der Darstellung ist, da beide nur über fest definierte Schnittstellen miteinander kommunizieren. Dieses Vorgehen erhöht die Wartbarkeit und Erweiterbarkeit von Code erheblich.
Außerdem ist die MVC-Architektur und ihre Funktionsweise bei vielen Entwicklern bekannt. Somit können sich Entwickler schneller in das Framework einarbeiten und können bereits angelerntes Wissen anwenden.

Data-Binding

Der Mechanismus von Databinding erleichtert einem Programmierer die Kommunikation zwischen View und Logik erheblich. Angular nimmt dem Entwickler hier die gesamte Implementierung einer push-pull-Konstruktion ab und stellt Bordmittel zur Verfügung.
Der Entwickler kann sich so auf die Anwendung konzentrieren und kümmert sich nicht um die Kommunikation zwischen den Components und ihren Templates.

Weiterentwicklung

Angular wird von einem erfahrenen Team, welches bereits für AngularJS verantwortlich war, entwickelt und kann von diesen Erfahrungen erheblich profitieren. Des Weiteren wird die Entwicklung des Frameworks von Google übernommen. Da Google als großes und finanzkräftiges Unternehmen über alle Mittel verfügt die Entwicklung von Angular zu unterstützen, können Nutzer davon ausgehen, dass das Framework für lange Zeit unterstützt und auch weiterentwickelt wird. Das schafft vor allem für Unternehmen Planungssicherheit.

Kapitel 4
Schwächen von Angular

Einarbeitung

Die erste Erstellung eines Projekts und der grundlegende Aufbau von Angular sind gut dokumentiert und einfach erklärt. Möchte man tiefer in die Materie einsteigen und eigene Anwendungen schreiben, muss man allerdings viel Zeit investieren. Das liegt zumeist daran, dass ein Entwickler die Prinzipien und den Aufbau der verschiedenen Features zunächst verstehen muss, um diese anwenden zu können.
Andere Frameworks, wie React, bieten eine höhere Abstraktion und erleichtern so die Verwendung des Frameworks für Einsteiger.

Unterstützung alter Browser

Angular wird lediglich von Evergreen Browsers nativ unterstützt. Ältere Browser wie sie zum Teil noch in Unternehmen verwendet werden, werden nur durch umständliche Polyfills unterstützt. Einige Features, wie Animationen können in diesen Browsern überhaupt nicht verwendet werden.
Das ist allerdings ein Nachteil, der auf viele JavaScript-Frameworks zutrifft. Abhilfe schafft dabei nur die stark eingeschränkte Nutzung von HTML in Kombination mit CSS.

Inkompatibilität

Ein großer Nachteil des neuen Angular ist die Inkompatibilität zu seinem Vorgänger AngularJS. Auch wenn sich die Namen der beiden Frameworks ähneln, wurde Angular ab Version 2 vollständig neu geschrieben. So müssen Anwendungen, welche in AngularJS geschrieben wurden, erst mühsam auf Angular portiert werden. Auch Entwickler müssen sich für die Verwendung des neuen Angular neu einarbeiten und können nur begrenzt auf ihr AngularJS-Wissen zurückgreifen.

Kapitel 5

Fazit

Als kurz gefasstes Fazit kann man sagen, dass Angular sich für die Entwicklung von Rich Internet Applications hervorragend eignet.
Insbesondere die ständige Interaktion mit dem Benutzer, welche für eine RIA typisch ist, wird in Angular einfach gelöst. Durch die Verwendung von Data Bindings (s. 2.5) fällt es dem Entwickler leicht die Interaktionen des Benutzers zu interpretieren und zu verarbeiten. Außerdem profitieren Single Page Anwendungen von den Routingmechanismen von Angular, welche zu einer besseren Übersicht in einem separierten Modul vorgenommen werden können.
Da Google die Entwicklung von Angular mit einem eigenen erfahrenen Team übernimmt, kann man sich auf eine konstante Entwicklung und regelmäßige Releases verlassen. Außerdem hat Google die Möglichkeit Angulars Bekanntheitsgrad zu erhöhen und so mehr Entwickler und Nutzer anzuziehen. Durch diese weite Verbreitung können Anwender von einer großen Community profitieren und Entwickler finden zu Problemstellungen schnelle und geprüfte Lösungen.
Für neue Anwendungen ist das Angular Framework daher durchaus interessant. Eine bestehende Anwendung sollte jedoch noch nicht unbedingt portiert werden. Angular ist in seiner aktuellen Struktur noch nicht sehr alt (Erste Veröffentlichung 14. September 2016) und daher nicht durchgehend erprobt. Auch die Portierung von AngularJS zu Angular ist nicht unproblematisch. Eine Empfehlung wäre es, mit einer Umstellung auf Angular bis zum nächsten Release (vermutlich Ende 2017) zu warten. Dadurch sollte es Entwicklern möglich sein, sich in das neue Framework einzuarbeiten und es sollte mehr Dokumentation und Best Practices zur Umstellung auf Angular geben.

Abbildungsverzeichnis

2.1 Übersicht der Architektur von Angular 4.0 4
2.2 Hero Detail - Mystique 8
2.3 Tour of Heroes - Dashboard 9
2.4 Prozess einer Dependency Injection von HeroService 13
2.5 Übersicht des Beispielprojekts - Tour of Heroes 14

3.1 Angular unterstützende Browser 15

Literaturverzeichnis

[DokAngArch] “Architecture Overview - The basic building blocks of Angular applications“,
https://angular.io/docs/ts/latest/guide/architecture.html,
zuletzt abgerufen am 10.05.2017

[DokAngTut] “Tutorial: Tour of Heroes“,
https://angular.io/docs/ts/latest/tutorial/,
zuletzt abgerufen am 10.05.2017

[DokAngCode] “Angular Example - Tour of Heroes“,
https://embed.plnkr.co/?show=preview,
zuletzt abgerufen am 10.05.2017

[RIA-Wiki] “Rich Internet Application“,
https://de.wikipedia.org/wiki/Rich_Internet_Application?oldid=157406000,
zuletzt abgerufen am 29.04.2017

[CLogAng] “Changelog - Angular“,
https://github.com/angular/angular/blob/master/CHANGELOG.md,
zuletzt abgerufen am 29.04.2017