Upgrade des Web-Frameworks AngularJS auf die neue Implementation Angular im Business-Umfeld

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

an der

Hochschule Anhalt - Anhalt University Of Applied Sciences
Fachbereich 5 - Informatik und Sprachen
Studiengang Angewandte Informatik - Digitale Medien und Spieleentwicklung (dual)

1. Prüfer:	Prof. Dr. Stefan Schlechtweg-Dorendorf
2. Prüfer:	Prof. Dr. Ursula Fissgus

Eingereicht von:	Toni Barth
Matrikelnummer:	4059906
Datum der Abgabe:	26. Juni 2018

Danksagung

Da an dieser Arbeit mehrere Personen essenziell unterstützend tätig waren, möchte ich hier meinen Dank aussprechen.

Dank gilt meinen Kollegen der GISA GmbH, welche für meine Probleme immer ein offenes Ohr hatten und mir halfen, wo immer es nötig war. Besonders hervorgehoben sei hier Marcel Tschullik, welcher die Betreuung meiner Bachelor-Arbeit übernahm und mich auch anfänglich in das Thema eingeführt hat.

Ich danke auch meiner Familie, welche mich mein Leben lang unterstützt und mir den Rücken gestärkt hat. Ohne das Durchsetzungsvermögen meiner Mutter und der soliden Robustheit meines Vaters wäre ich heute sicher nicht da, wo ich es jetzt bin.

Viel zu verdanken habe ich Ralf Höhrmann, denn ohne dich hätte ich vermutlich nie zur Programmierung und damit meiner Berufung gefunden. Danke für deine sprichwörtliche Engelsgeduld, ohne die mir der Einstieg sicher nicht geglückt wäre.

Vielen Dank auch an meine Freundin Lea. Du musstest oft wegen der Bachelor-Arbeit auf meine Anwesenheit verzichten und während der Telefonate meine oft ruppige Art nach einem anstrengenden Arbeitstag über dich ergehen lassen. Danke für dein Vertrauen und dein Durchhaltevermögen.

Der größte Dank gilt allerdings Max Haarbach, welcher, trotz seiner eigenen Bachelor-Arbeit und dem generellen Stress eines Abschlusssemesters, jede Woche mehrere

Stunden Zeit fand, um die großen und kleinen Probleme des Layouts einer Bachelor-Arbeit zu diskutieren und zu beheben. Unabhängig davon hätte ich heute nicht die Möglichkeit, meine Bachelor-Arbeit zu schreiben, wenn du mich im Studium nicht so umfangreich unterstützt hättest, um die Nachteile, welche mit der Blindheit einher kommen, auszugleichen. Ja, man kann sagen, dass du nicht unerheblich für meinen zukünftigen Werdegang Verantwortung trägst, und das in äußerst positivem Sinne. Dafür bin ich dir unglaublich dankbar.

Inhaltsverzeichnis

Kapitel 1

Motivation und Aufgabenstellung

Im Jahr 2017 häuften sich bei der GISA GmbH die Situationen, in denen Kunden eine Anpassung ihres Webauftrittes an aktuelle Gegebenheiten wie beispielsweise neue Browser, fortschrittlichere Anzeigetechnologien oder Bedienkonzepte forderten. Das von der GISA GmbH bis dato eingesetzte Entwicklungskonzept, welches auf AngularJS und dessen Erweiterungen basierte, stellte sich nach und nach als zu starr heraus. Erweiterungen, welche benötigt wurden, um die Forderungen umzusetzen, verlangten eine bestimmte Version des AngularJS-Frameworks, welche jedoch nicht verwendet werden konnte, weil ebenfalls notwendige andere Erweiterungen mit dieser Version nicht funktionieren würden.

Immer mehr Zeit floss in die Anpassung des AngularJS-Frameworks und seiner Erweiterungen, statt in die Entwicklung der Kundenauftritte selbst. Durch den größeren Zeitaufwand mussten die Preise gegenüber dem Kunden erhöht werden, was wiederum zu einem Nachteil im Wettbewerb um den Auftrag führte.

Auf der Suche nach Lösungen lag der Umstieg des veralteten AngularJS-Frameworks auf die neuere und erweiterte, zukunftssichere Version Angular nahe, welche deutlich moderner und flexibler, sowie sicherer und schneller ist. Die Entwicklung von Applikationen in Angular würde aufgrund der neuen etablierten Projektstruktur ebenfalls Zeitvorteile mit sich bringen. Diese Arbeit befasst sich daher mit dem Vorgang der Umstellung des AngularJS-Frameworks auf das modernere Angular-Framework.

Zu Beginn dieser Arbeit werden daher die Grundlagen erläutert, welche von Nöten sind, um die Portierung vorzunehmen. Im Kapitel „Umsetzung“ wird schließlich die eigentliche Umstellung erläutert, wobei die größten und umständlichsten Schritte möglichst detailliert aufgeschlüsselt werden. Zum Schluss wird dann das Ergebnis präsentiert und ein Fazit bezüglich der Aufgabenstellung gezogen und untersucht, ob die Umstellung für bereits etablierte Unternehmen im AngularJS-Umfeld von Vorteil ist.

Kapitel 2

Grundlagen

Dieses Kapitel zeigt die Grundlagen dieser Arbeit auf. Dazu gehört die Erklärung, wovon diese Arbeit handelt, aber auch die Notwendigkeit, warum genau die Problemlösung dieser Arbeit die Umstellung des einen Frameworks auf das Andere erfordert.

2.1 Frameworks

Dieser Abschnitt erklärt, worum es sich bei den beiden Frameworks AngularJS und Angular handelt und worin sie sich unterscheiden oder gleichen.

2.1.1 AngularJS

Von Google im Jahr 2009 das erste Mal veröffentlicht, handelt es sich bei AngularJS um ein Framework zur Entwicklung einer Single-Page-Webanwendung. Die Webanwendung wird dabei clientseitig ausgeführt, d.h. der Aufbau und die Funktionsweise der Webseite wird rein durch den Browser des Anwenders bestimmt. Somit grenzen sich mittels AngularJS entwickelte Webanwendungen gegen serverseitige Webanwendungen ab. Diese werden von einem Webserver generiert und anschließend nur noch an den Browser gesendet, welcher diese dann darstellt. Bei clientseitigen Lösungen wie AngularJS dagegen wird vom Code vorgegeben, was der Browser tun muss und welche Ressourcen er aus dem Internet beziehen muss, um die Seite korrekt darstellen zu können. Die Darstellung, das Herunterladen der notwendigen Dokumente und die Auswertung des Quellcodes übernimmt dann jedoch der Browser selbst.

AngularJS wird unter der MIT Lizenz [Ope] zur Verfügung gestellt, welche jegliche

Veränderung und Modifikation erlaubt. Da AngularJS quelloffen (Open Source) angeboten wird, bietet es die einfache Möglichkeit, Veränderungen vorzunehmen, ohne sich erst lizenzrechtliche Genehmigungen einzuholen oder gar Geld dafür zahlen zu müssen. Es sind also firmeninterne Anpassungen möglich, um das Framework bestmöglich in den Arbeitsablauf professioneller Firmen einzubetten.

Derzeit ist AngularJS in der Version 1.7.2 erhältlich, welche am 12.06.2018 veröffentlicht wurde. Seit einiger Zeit stellen diese Updates allerdings nur noch Fehlerbehebungen und Korrekturen dar - neue Funktionen werden hierbei nicht mehr hinzugefügt.

2.1.2 Angular

Angular ist eine neu entwickelte Version des AngularJS-Frameworks, wobei viele Schwachstellen, Engpässe und Schiefstände angefasst und von Grund auf erneuert wurden. Angular ist nicht abwärtskompatibel zu AngularJS, weshalb diese beiden Frameworks, obwohl sie namentlich ursprünglich nur mittels der Versionsnummer unterscheidbar waren, inzwischen in AngularJS (Version 1+) und Angular (Version 2+) unterschieden werden.

Wie bereits AngularJS wird Angular von mehreren Entwicklern, aber hauptsächlich Google, entwickelt und wurde erstmals Ende 2016 als stabile Version veröffentlicht. Inzwischen ist Angular in der Version 6.0.6 aktuell, welche am 21.06.2018 veröffentlicht wurde. Zu beachten ist hierbei, dass Angulars Versionierung immer unter der Premisse der generellen Version 2 zu betrachten ist. Betrachtet man also das Projekt „Angular“ als solches, zu welchem sowohl AngularJS (bis Version 1.6.7), aber auch Angular (Version 2+) gehören, so stellt Angular Version 4.3.6 im Großen und Ganzen Version 2.4.3.6 dar.

Bei der Entwicklung von Angular wurde jedoch nicht nur die Codestruktur neu durchdacht, sondern auch die dem Arbeitsprozess zu Grunde liegenden Werkzeuge. In AngularJS übliche Werkzeuge wurden entfernt und durch ein zentrales Werkzeug, die Angular CLI, abgelöst (siehe Abschnitt 2.4). Außerdem ändert sich die Codestruktur grundlegend. Google versuchte hierfür bereits in AngularJS nach und nach auf die neue Codestruktur umzustellen, um den Entwicklern im Nachhinein den Wechsel auf Angular zu erleichtern. Entwickler sehen sich allerdings gelegentlich mit teilweise

sehr alten Projekten konfrontiert, weshalb sie diesen Schritt noch selbst durchführen müssen (siehe Abschnitt 3.3).

2.2 Warum Angular statt AngularJS?

In der Aufgabenstellung wurden schon einige Gründe dafür genannt, warum ein Upgrade von AngularJS auf Angular empfehlenswert ist. Diese sollen hier noch einmal detailliert diskutiert werden.

Zunächst einmal besteht der offensichtliche Grund der Aktualität: AngularJS erhält keine Funktionsupdates mehr, hier werden nur noch Fehlerbehebungen nachgereicht, und auch dieser Support wird voraussichtlich in absehbarer Zukunft eingestellt. Angular wird von den Entwicklern beider Frameworks nun als Basis neuer Projekte empfohlen und erhält regelmäßige Updates, welche neue Funktionen beinhalten, zusätzlich zu den üblichen Fehlerbehebungen.

Die spürbar erhöhte Performance von Angular im Gegensatz zu AngularJS ist ebenfalls ein Grund, die Migration durchzuführen.

> The migration's main advantage is its boost in performance, since Angular is 5 times faster than Angular JS. [Har]

Die Entwickler haben bei dem Upgrade zusätzlich einige Komponenten isoliert und einzeln in Pakete ausgelagert. Dies resultiert in einer allgemein geringeren Größe des Angular-Frameworks selbst, außerdem wird dadurch die Sicherheit verbessert, da die Angriffsfläche minimiert wird.

Für Programmierer, welche Wert auf eine mobile Aufbereitung ihrer Webseite legen, ist der Sprung zu Angular ebenfalls empfehlenswert. AngularJS selbst besaß keinerlei Optimierung für mobile Webseiten, diese wurde über zusätzliche Pakete ermöglicht, welche dementsprechend zusätzliche Ressourcen verschlingen. Angular dagegen bringt nun von sich aus eine Optimierung für mobile Webseiten mit, ohne dabei unnötig viele zusätzliche Betriebsmittel zu beanspruchen - im Gegenteil, die bereits oben erwähnte Steigerung der Performance wirkt sich positiv auf die mobile Webseite aus.

Das neue Design des Angular-Frameworks bewirkt, dass Module eines Projektes bereits vor der Auslieferung an den Browser „kompiliert“ werden. Üblicherweise wird ein Projekt im Webumfeld als eine Mischung aus HTML-, CSS-, und JavaScript-Dateien ausgeliefert. Jede einzelne dieser Dateien muss bei Bedarf von dem Browser heruntergeladen und interpretiert werden, wobei die HTML-, und CSS-Dateien in letzter Konsequenz ebenfalls in JavaScript-Code überführt werden, da dieser Code vom Browser direkt ausführbar ist. Der größte Teil der Zeit geht dabei bei dem Aufbau der Verbindungen für den Download der einzelnen Dateien verloren. Während dies dennoch das Standardverfahren im AngularJS-Umfeld darstellte, sorgt bei Angular ein sogenannter Ahead-Of-Time-Compiler dafür, dass bereits bei der Bereitstellung der Dateien auf dem Webserver alle Dokumente in JavaScript-Dateien umgewandelt und in so wenig Dateien wie möglich zusammengefasst werden, um die Verarbeitungszeit auf Anwenderseite so weit wie möglich zu reduzieren. Diese Umstellung beschleunigt den Ladevorgang der Webseite, was gerade unterwegs von Vorteil ist.

Schließlich wurde die Code-Struktur deutlich vereinfacht, was es Neueinsteigern vereinfacht, in den Projektbetrieb einzusteigen (siehe Abschnitt 3.3). Die Entwicklungssprache wechselte von JavaScript (siehe Abschnitt 2.3.1) zu TypeScript (siehe Abschnitt 2.3.2), welches dem heutzutage eher üblichen Entwicklungsstil entspricht und die Entwicklung vereinfacht.

Es lässt sich also sagen, dass die Migration von AngularJS auf Angular generell zu empfehlen ist, weshalb die folgenden Kapitel versuchen, die einzelnen Schritte, Gefahren und Fallstricke aufzuzeigen und einen Weg zu finden, diese Hürden zu nehmen.

2.3 Relevante Sprachen

Im Folgenden werden die Skriptsprachen beschrieben, welche bei der Entwicklung im Webumfeld, und besonders mit AngularJS und Angular, zur Verwendung kommen.

2.3.1 JavaScript

JavaScript ist eine interpretierte Programmiersprache, welche hauptsächlich im Web-Umfeld eingesetzt wird, um das Design oder das Verhalten einer Webseite dynamisch zu verändern, weshalb sie auch größtenteils auf der Seite des Clients, also im Browser, zum Einsatz kommt. JavaScript kann dabei sowohl als Sprache für die Prozedurale Programmierung, aber auch für die Objektorientierte Programmierung (OOP) eingesetzt werden. Gerade aufgrund dieser Flexibilität und dank der Tatsache, dass JavaScript nur eine interpretierte Sprache ist, besitzt es allerdings einige Schwachstellen. Eine der Größten ist der Nachteil gegenüber stark typisierten Programmiersprachen, dass die Konsistenz des Codes nicht im Vornherein geprüft werden kann.

Da JavaScript schwach typisiert ist, d.h. in einer Variable prinzipiell jede Art von Daten liegen kann, kommt es hier schnell zu Fehlern während der Ausführung des Codes. Bei anderen Programmiersprachen, welche oft Variablen mit festen Typen besitzen, kann hierbei bereits vor der Ausführung des Codes geprüft werden, ob die Zugriffe der Variablen auch konform mit ihrem jeweiligen Datentyp sind. Außerdem können Eingabe- und Ausgabeparameter von Funktionen in JavaScript nicht fest typisiert werden. Diese Tatsache führt häufig zu Fehlern, welche erst im laufenden Betrieb festgestellt werden können. Diese Handhabung macht JavaScript-Code generell schwer wartbar, da Entwickler beim Lesen des Codes nicht sofort erkennen können, zu welchem Zweck und zur Aufbewahrung welcher Daten eine Variable definiert wurde. Hierfür ist intensiveres Einarbeiten notwendig, was sowohl Zeit, als auch Geld kostet.

Ein nicht zu missachtender Vorteil besteht dennoch in der einfachen Erlernbarkeit von JavaScript. Ein Hallo Welt Beispiel wie in Quellcodeauszug 2.3.1 ist sehr einfach geschrieben:

Quellcodeauszug 2.3.1: „Hallo Welt“-Beispiel in JavaScript

```
1 // Dieses Beispiel erfordert zwingend JavaScript
2 alert("Hallo Welt");
```

JavaScript verwendet eine sehr abstrakte Implementation von Objekten, was das Programmieren im heute üblichen objektorientierten Stil erschwert. Zwar ist in JavaScript prinzipiell jede Variable, egal ob String, Zahl oder Funktion ein Objekt,

allerdings existieren Konzepte wie Klassen, Vererbung und Zugriffsbeschränkung nicht. Zumindest Klassen und Vererbung können mittlerweile über Umwege realisiert werden.

Hierbei werden Klassen als Funktionen definiert, welche, da sie intern ebenfalls nur Objekte sind, wiederum Eigenschaften besitzen können. JavaScript selbst besitzt bereits eine rudimentäre Unterstützung für Zugriffsräume, was dazu führt, dass mithilfe des „this“-Schlüsselwortes innerhalb eines Objektes auch auf die spezifischen Eigenschaften dieses Objektes zugegriffen werden kann, womit schließlich Eigenschaften und Methoden einer Klasse abgebildet werden. Diese Funktionen können nun, statt einfach aufgerufen zu werden, wie es eigentlich üblich ist, mittels des „new“-Schlüsselwortes initialisiert werden und somit ein Objekt der Pseudoklasse erzeugen. Eine zusätzliche Schwierigkeit entsteht, sobald Methoden vererbt, oder gar den Klassen statische Methoden mitgegeben werden sollen. Hier muss das „prototype“-Objekt, welches jede Funktion in JavaScript besitzt, neu erzeugt werden, da sonst in der erbenden Klasse ausversehen Funktionen angelegt werden, welche auch für die Elternklasse zugänglich sind.

Das Beispiel im Quellcodeauszug 2.3.2 demonstriert diese Techniken:

Quellcodeauszug 2.3.2: Klassen in JavaScript

```
1  // Klassen in JavaScript sind nur Funktionen
2  var Person = function(name) {
3    // Diese Funktion ist mit dem Konstruktor
4    // gleichzusetzen
5    this.name = name;
6  };
7
8  // um der Klasse Methoden zu verleihen, werden sie
9  // im Folgenden definiert
10 Person.prototype.halloSagen = function() {
11   console.log("Hallo, ich bin "+this.name);
12 };
13
14 // es werden Objekte erzeugt
15 var toni = new Person("Toni Barth");
```

```
16 // resultiert in "Hallo, ich bin Toni Barth"
17 toni.halloSagen();
18
19 // Vererbung ist allerdings noch deutlich umständlicher
20 function Student(name, lieblingsmodul) {
21   // den Konstruktor der Elternklasse aufrufen
22   Person.call(this, name);
23   this.lieblingsmodul = lieblingsmodul;
24 }
25
26 // es muss ein neues prototype Objekt erstellt werden,
27 // da es sonst zu Problemen mit den Zuweisungen kommt
28 Student.prototype = Object.create(Person.prototype);
29 Student.prototype.constructor = Student;
30
31 // die frühere halloSagen() Methode wird überlagert
32 Student.prototype.halloSagen = function() {
33   console.log("Hallo, ich bin "+this.name+", und mein
     ↪  Lieblingsmodul ist "+this.lieblingsmodul);
34 };
35
36 // Toni wird als Student erzeugt
37 var student_toni = new Student("Toni Barth",
    ↪  "Programmierung");
38 // er begrüsst uns
39 student_toni.halloSagen();
40
41 // toni und student_toni sind allerdings nicht
42 // eindeutig typisiert
43
44 // verursacht keinen Fehler und ist durchaus legal
45 toni = 0815;
46
47 // ist ebenfalls kein Problem
48 student_toni = new Person("Toni Barth");
49
```

```
50 // ist ebenfalls machbar
51 student_toni = new Student("Toni Barth", 0815);
```

Dennoch gibt es in der objektorientierten Programmierung Ansätze, welche in reinem JavaScript nicht abgebildet werden können, wozu auch die Zugriffsbeschränkung gehört. Das bedeutet, dass in Pseudo-Klassen enthaltene Variablen, wie die Variable „name“ im obigen Beispiel, in jedem Fall von außen zugänglich sind und man diese nicht als privat, readonly oder Ähnliches definieren kann.

JavaScript ist in dieser Arbeit von Relevanz, da AngularJS (siehe Abschnitt 2.1.1) in JavaScript entwickelt wurde und die Projekte, welche AngularJS verwenden, üblicherweise ebenfalls in dieser Sprache verfasst werden. Eine vollständige Referenz für JavaScript findet man bei JavaScript Kit [Jav].

2.3.2 TypeScript

TypeScript ist eine von Microsoft entwickelte Sprache, welche 2012 das erste mal stabil veröffentlicht wurde. TypeScript ist ein Ansatz, welcher es Entwicklern ermöglichen soll, zwar in JavaScript zu entwickeln, allerdings dennoch einem besseren Entwicklungsablauf folgen zu können. Dabei gilt der Grundsatz, dass zwar JavaScript gleichzeitig auch gültiges TypeScript ist, allerdings TypeScript nicht automatisch gültiges JavaScript ist. Um TypeScript in gültiges JavaScript zu übersetzen, kommt ein sogenannter Transpiler zum Einsatz. Da man in TypeScript-Code die Typen von einzelnen Variablen definieren kann, eine Funktion, welche JavaScript nicht unterstützt, wird hierbei gleich überprüft, ob im TypeScript-Code ausschließlich gültige Operationen auf die definierten Variablentypen ausgeführt werden, und ob auch die richtigen Parametertypen an aufgerufene Funktionen weitergegeben werden.

Allein schon diese Funktion ist ein großer Pluspunkt für TypeScript, denn diese Funktionalität erleichtert und verbessert den Entwicklungsablauf immens. Zusätzlich bietet TypeScript vereinfachte Schreibweisen für Funktionen, zusätzliche Schlüsselwörter und ergänzende Operatoren. Alle diese sind allerdings nur für den Transpiler relevant, denn nach Abschluss der Verarbeitung wird jeglicher TypeScript-Code in gültigen JavaScript-Code abgebildet.

Das TypeScript-Beispiel für „Hallo Welt“ ist daher identisch zu dem Beispiel in JavaScript. Erst, wenn es um Klassendefinitionen geht, beginnt TypeScript damit, seine Stärken auszuspielen, denn TypeScript bringt native Unterstützung für OOP mit, weshalb eine Klasse hier mithilfe des „class“-Schlüsselwortes erzeugt wird. Sämtliche Fallstricke, welche in JavaScript zu beachten sind, werden von dem Transpiler automatisch beachtet und sind für den Entwickler von nun an irrelevant, zu sehen in Quellcodeauszug 2.3.3.

Quellcodeauszug 2.3.3: Klassen in TypeScript

```
1  // eine Klasse definieren ist eingängig
2  class Person {
3    // es werden die Eigenschaften einer Klasse definiert
4    // der Name darf ausschließlich ein String sein
5    private name: string;
6
7    // es folgt der Konstruktor
8    constructor(private name: string) {
9      // es muss nichts weiter getan werden, weil die
10     // Anweisung private den Transpiler bereits anweist,
11     // den Funktionsparameter in die gleichnamige
12     // Klasseneigenschaft zu übernehmen.
13     // Theoretisch müsste name außerhalb des
14     // Konstruktors nicht einmal definiert werden,
15     // diese Art von Parameterdefinition erledigt das
16     // bereits. Allerdings trägt es zur
17     // Übersichtlichkeit bei, die Definition wie
18     // oben vorzunehmen.
19   }
20
21   // die halloSagen methode
22   halloSagen () -> void {
23     console.log("Hallo, ich bin "+this.name);
24   }
25 }
26
27 // Toni wird erzeugt
```

```
28 var toni: Person = new Person("Toni Barth");
29 // das Hallo Sagen funktioniert gleichermaßen
30 toni.halloSagen();
31
32 // einen Student als Erweiterung der Person erzeugen
33 // ist ungemein leichter
34 class Student extends Person {
35   private lieblingsmodul: string;
36
37   constructor(name: string, private lieblingsmodul:
     ↪  string) {
38     // ruft den Konstruktor der Elternklasse auf
39     super(name);
40   }
41
42   halloSagen () -> void {
43     console.log("Hallo, ich bin "+this.name+" und mein
       ↪  Lieblingsmodul ist "+this.lieblingsmodul);
44   }
45 }
46
47 // Toni wird als Student erzeugt
48 var student_toni: Student = Student("Toni Barth",
   ↪  "Programmierung");
49 // wenn die Variable student_toni mit Student als
50 // Datentyp definiert wird, kann diese Variable auch
51 // nur einen Studenten enthalten, aber keine
52 // andere Person, oder Anderes
53 student_toni.halloSagen();
54
55 // da ein Student aber gleichzeitig eine Person ist,
56 // ist folgendes möglich:
57 toni = new Student("Toni Barth", "Programmierung");
58 toni.halloSagen();
59
60 // folgendes würde allerdings beispielsweise
```

```
61 // nicht funktionieren:
62 // toni = new Student("Toni Barth", 0815);
63 // TypeScript würde dann beim transpilieren Fehler melden
```

Wie in diesem Beispiel bereits festgestellt, ist es in TypeScript auch möglich die Eigenschaften einer Klasse mit Zugriffsbeschränkungen zu versehen. So können hier die Modifikatoren public, private, protected und readonly gesetzt werden. Jede Variable in einer Klasse ist standardmäßig public und folgt damit dem Verhalten von JavaScript. Eine private Variable ist nur von ihrer eigenen Klasse verwendbar, während protected Variablen zumindest noch von Kindklassen zugänglich sind. Readonly-Eigenschaften sind schließlich zumindest von äußeren Klassen oder Objekten lesbar, aber können nicht verändert werden. Readonly ist zusätzlich gemeinsam mit den anderen Zugriffsmodifikatoren einsetzbar, um den Zugriff weiter zu beschränken. So kann man Eigenschaften von Klassen sowohl als private, als auch als readonly definieren, damit die so definierte Eigenschaft ausschließlich von dieser Klasse eingesehen werden kann. Schreibende Aktionen werden hierbei allerdings vom Transpiler verhindert.

Diese Eigenschaften machen TypeScript zu einer sehr soliden Lösung im Webumfeld und ermöglichen einen geregelteren Entwicklungsablauf, weshalb auch Angular (siehe Abschnitt 2.1.2) in TypeScript entwickelt wird und es daher naheliegt, alle Projekte, welche mittels Angular realisiert werden, ebenfalls in TypeScript umzusetzen. Die Mehrheit aller Pakete, welche beispielsweise via NPM (siehe Abschnitt 2.4.1) angeboten werden, sind daher in TypeScript verfügbar, sofern sie für Angular entwickelt wurden. Pakete, welche sowohl für Angular, als auch für AngularJS entwickelt wurden, bringen meistens Bindings für TypeScript mit, damit ein in TypeScript entwickeltes Projekt weiß, welche Typen und Berechtigungen die Komponenten dieses Paketes besitzen.

Eine vollständige Referenz für TypeScript findet man auf den Webseiten der Sprache selbst [Micb].

2.4 Werkzeuge

Bei der Entwicklung von Anwendungen mithilfe von AngularJS (siehe Abschnitt 2.1.1) oder Angular (siehe Abschnitt 2.1.2) sind mehrere Werkzeuge von Nöten. Einige davon werden hier beschrieben, da diese bei der Migration speziell behandelt werden müssen.

2.4.1 NPM

NPM (ehemals Node Package Manager) ist ein Paketmanager für JavaScript Pakete. Ein Paketmanager ist dabei eine Software, welche es ermöglicht, sogenannte Pakete herunterzuladen und in ein Projekt zu importieren. Ein Paket ist ein Projekt, welches von einem Entwickler bereitgestellt wurde, um von anderen Entwicklern in deren Projekten genutzt zu werden. Auf diese Art und Weise können Entwickler ihre Arbeit miteinander teilen und sich gegenseitig Zeit sparen, indem man bereits vorhandene Projekte verwendet, statt alles von Grund auf neu entwickeln zu müssen.

In der Registry, also der Paketzentrale von NPM, sind über 600.000 große und kleine Pakete zu finden. NPM ermöglicht es dabei, eine bestimmte Version eines Paketes in ein Projekt zu importieren. Diese Funktion erlaubt es einem Entwickler, selbst mit veralteten Versionen eines Projektes weiter zu arbeiten, was es beispielsweise möglich macht, heutzutage noch mit AngularJS (siehe Abschnitt 2.1.1) zu entwickeln, obwohl es bereits die deutlich neuere Version des Frameworks gibt.

NPM ist hierbei nur einer von mehreren möglichen Paketmanagern, hat sich allerdings im JavaScript-Umfeld zu dem Standardwerkzeug für die Paketverwaltung herausgebildet. NPM ist dabei Bestandteil der Node.js-Softwaresammlung, welche zur lokalen Ausführung von JavaScript-Code außerhalb von Browsern dient und NPM gleich zusätzlich installiert.

2.4.2 Grunt

Grunt bezeichnet sich selbst als „The JavaScript Task Runner“ [Gru]. Grunt ist ein Hilfsmittel, welches es ermöglicht, verschiedene Programme in einer vordefinierten

Reihenfolge auszuführen. Dies ist vor allem dann wichtig, wenn das Projekt ausgeliefert werden soll. Ein Projekt ist beispielsweise in mehrere Dateien untergliedert, um eine leichte Wartbarkeit zu gewährleisten. Je mehr Dateien das Projekt jedoch enthält, wenn es dem Anwender im Browser präsentiert werden soll, desto länger dauert es, die Webseite im Browser des Anwenders aufzubauen. Der Entwickler ist also bestrebt, das Projekt bei der Auslieferung so kompakt wie möglich zu strukturieren, ohne dabei jedoch auf die einfachere Wartbarkeit verzichten zu müssen.

Ein weiteres übliches Einsatzgebiet von Grunt ist die sogenannte Obfuscation. Hierbei wird der Quellcode bei der Auslieferung verzerrt, indem beispielsweise Variablen abstrakt umbenannt werden. Dies soll die Lesbarkeit durch andere Menschen behindern, damit essenzieller Code von anderen Menschen schlechter gelesen und somit eventuelle Schwachstellen im Code deutlich schwerer erkannt und ausgenutzt werden können.

Zu diesen Zwecken können mit Grunt sogenannte Tasks definiert werden, welche diese Schritte abbilden und am Ende ein fertig bearbeitetes Projekt erzeugen, welches dann an den Kunden ausgeliefert werden kann. Grunt wird hauptsächlich in AngularJS-Projekten verwendet, da es hier noch kein anderes, von den AngularJS-Entwicklern empfohlenes Werkzeug gibt.

2.4.3 Angular CLI

Die Angular CLI (Angular Command Line Interface) wurde zeitgleich mit Angular (siehe Abschnitt 2.1.2) veröffentlicht, da die Entwickler von Angular ein zentrales Werkzeug zur Verfügung stellen wollten, um die gängigsten Aufgaben eines Projektes ausführen zu können. Die Angular CLI löst daher in Angular Grunt ab und bietet sogar noch mehr Funktionen. So benötigt man beispielsweise üblicherweise zur Bereitstellung einer Webseite einen Server, welcher die Dokumente an den Browser sendet. Angular CLI bringt auch einen solchen Server mit, wodurch ein weiteres zusätzliches Werkzeug entfällt.

Eine der wichtigsten Funktionen besteht jedoch darin, dass es Angular CLI dem Entwickler erleichtert, neue Projekte anzulegen oder bereits bestehende Projekte zu erweitern. So wird mittels der Befehle aus Quellcodeauszug 2.4.1 ein neues Projekt angelegt und gleich eine neue Komponente hinzugefügt:

Quellcodeauszug 2.4.1: Projekt anlegen mit Angular CLI

```
1 $ ng new TestProjekt
2 $ ng generate component NavBar
```

Ein auf diese Art und Weise erstelltes Projekt kann anschließend mittels des Befehls

```
$ ng serve
```

gestartet werden, woraufhin ein Server auf Port 4200 läuft. Sobald die Dateien des Projektes verändert werden, aktualisiert sich der Server entsprechend. Die Aufgaben von Grunt, also das Ausführen des Build-Vorganges, übernimmt Angular CLI mittels:

```
$ ng build
```

Kapitel 3

Umsetzung

In der Umsetzung wird nach und nach der Umstieg von AngularJS auf Angular beschrieben. Dabei wird zuerst die Integration von TypeScript in ein bereits bestehendes AngularJS-Projekt beschrieben, wobei auch die bislang verwendeten Werkzeuge angepasst werden müssen. Anschließend kann optional ein Upgrade auf AngularJS Version 1.5 durchgeführt werden, um den Umstieg zu erleichtern. Die größte Arbeit verlangt anschließend die Aufrüstung mittels ngUpgrade, um unter Beibehaltung der Lauffähigkeit des Projektes nach und nach Portierungen nach Angular vornehmen zu können. Zum Schluss wird das vollständig unter Angular lauffähige Projekt in ein vollständiges Angular-Projekt migriert, woraufhin der Vorgang abgeschlossen ist.

3.1 Aufbau des AngularJS-Projektes

Ein typisches AngularJS-Projekt, wie es in diesem Kapitel nach Angular portiert werden soll, folgt üblicherweise einem immer gleichen Aufbau. Dieser sieht die Speicherung aller Komponenten, End-to-End-Tests, Unittests und etwaiger anderer Ressourcen in separaten Ordnern vor. Ein solches Projekt wird dabei in mindestens diese Ordner gegliedert:

- /app: Der App-Ordner beinhaltet alle Module des Projektes, welche wiederum deren Komponenten, Direktiven und Templates beinhalten. Diese liegen in „js“-Dateien vor, welche gültiges JavaScript beinhalten. HTML-Bestandteile können entweder in HTML-Dateien gelagert, oder eventuell als Template-String direkt im JavaScript-Code hinterlegt werden, wohingegen CSS-Dateien prinzipiell immer ausgelagert werden. Die Dateien werden zusammenhängend gegliedert - alle Dateien, welche beispielsweise zu einem Model gehören, heißen also gleich, nur die Dateiendung ist, je nach Inhalt, unterschiedlich.

- /build: Dieser Ordner ist nicht immer notwendig und kann während des Build-Prozesses entstandene Dateien beinhalten, welche für diverse Zwischenschritte notwendig sind. In der Regel kann er nach Abschluss des Build-Prozesses wieder gelöscht werden, weshalb diese Aktion meistens automatisiert wird.

- /dist: Dieser Ordner beinhaltet die für den produktiven Einsatz vorgesehenen finalen Dateien des Projektes. Diese liegen nur noch in möglichst wenigen aber kompakten JS, CSS und HTML-Dateien vor, um viele HTTP-Requests zu vermeiden.

- /e2etests und /tests: Diese Ordner beinhalten die Dateien zur Steuerung der End-to-End-Tests und Unittests des Projektes. Diese sind üblicherweise ausschließlich in JavaScript geschrieben.

- /bower_components und /node_modules: Diese Ordner beinhalten Abhängigkeiten des Projektes, welche über Paketmanager installiert wurden (siehe Abschnitt 2.4.1).

Je nach Wunsch können die Templates, also die HTML-Vorlagen zur Abbildung der einzelnen Teile der Webseite, aus dem /app Ordner ausgelagert werden, und auch andere Änderungen sind möglich. Zur einfacheren Einarbeitung neuer Mitarbeiter und auch zugunsten der besseren Wartbarkeit sollte jedoch davon abgesehen werden.

Ein derart strukturiertes AngularJS-Projekt soll im Laufe dieses Kapitels nach Angular portiert werden. Es wird daher davon ausgegangen, dass in diesem Projekt eine AngularJS-Version < 1.5 verwendet wird.

3.2 Umstieg auf TypeScript

Da Projekte, welche mithilfe von AngularJS arbeiten, noch in JavaScript entwickelt werden, Angular jedoch in TypeScript entwickelt und man daher Angular-Projekte ebenfalls in TypeScript entwickeln sollte, befasst sich dieses Kapitel mit der möglichst reibungslosen Integration von TypeScript in bereits bestehende Projekte. Um den Übergang so fließend wie möglich zu gestalten, wird hier beschrieben, wie man TypeScript in ein (noch) AngularJS-Projekt integriert und dieses Projekt kom-

plett nach TypeScript umstellt. AngularJS-Projekte benötigen TypeScript eigentlich nicht, jedoch ist es für den fließenden Übergang empfehlenswert, TypeScript vor der eigentlichen Umstellung des Codes nach Angular einzubinden.

3.2.1 Installation von TypeScript und Type-Definitionen

Zuerst muss der TypeScript-Transpiler heruntergeladen und in die Ordnerstruktur der Abhängigkeiten integriert werden. Außerdem muss die Abhängigkeit für Entwickler in der NPM-Konfigurationsdatei hinterlegt werden. Der Befehl in Quellcodeauszug 3.2.1 erledigt beide Aufgaben:

Quellcodeauszug 3.2.1: Installation von TypeScript

```
$ npm install --save-dev typescript
```

Anschließend befindet sich das TypeScript-Paket im Projekt und wird als Abhängigkeit erkannt. Jetzt steht der Befehl

```
$ tsc
```

zur Verfügung, um den Transpiler auszuführen. Allerdings spielt TypeScript seine Stärken erst dann aus, wenn der Transpiler weiß, welche Variablen, Funktionen und Objekte welche Typen besitzen. Da AngularJS aber nicht in TypeScript entwickelt wird, werden seine Type-Definitionen nicht standardmäßig mit installiert. Diese müssen daher nachträglich wie in Quellcodeauszug 3.2.2 installiert werden:

Quellcodeauszug 3.2.2: Installation von Type-Definitionen für AngularJS

```
1 $ npm install --save-dev @types-angular
2 # Hier müssen auch alle Typen für andere
3 # AngularJS-Abhängigkeiten installiert werden
4 # Beispiel: npm install --save-dev @types/angular-mocks
```

Der TypeScript-Transpiler wird über eine Konfigurationsdatei gesteuert, welche üblicherweise im Wurzelverzeichnis des Projektes liegt und alle möglichen Informationen

beinhaltet, wie der Transpiler sich verhalten soll. So beinhaltet die Datei Eingabe- und Ausgabeordner, aber auch die Striktheit des Transpilers, also bei welchen Fällen Fehler oder nur Warnungen ausgelöst werden sollen. Hier kann man experimentelle Funktionen ein- oder abschalten. Interessant ist dabei auch, dass man hier festlegen kann, in welche Version von JavaScript der TypeScript-Code übersetzt werden soll. Möchte man beispielsweise ein Projekt entwickeln, welches sogar noch mit Internet Explorer 6 lauffähig sein soll, kann man hier eine ältere JavaScript-Version angeben, damit dies dann tatsächlich auch der Fall ist. Eine solche „tsconfig.json“-Datei könnte wie im Folgenden Quellcodeauszug 3.2.3 aussehen:

Quellcodeauszug 3.2.3: Beispiel einer „tsconfig.json“-Datei

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "outDir": "./build",
    "rootDir": "./src/",
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types",
      "types"
    ],
    "lib": [
      "es2017",
      "dom"
    ]
  }
}
```

3.2.2 TypeScript-Integration in Grunt

In AngularJS ist Grunt (siehe Abschnitt 2.4.2) ein Hilfsmittel, um häufig auftretende Vorgänge, sogenannte Tasks, effizient zu implementieren. Grunt führt hierbei Tasks wie den Build-Prozess aus, welcher die Übersetzung und Bündelung der Quellcode-Dateien vornimmt, aber auch Aufräumaufgaben (clean) oder die Veröffentlichung neuer Projektversionen. Grunt kann man hier mit den Targets von make vergleichen.

Nachdem nun der TypeScript-Transpiler fertig eingerichtet wurde, kann auch TypeScript in den Build-Prozess mit Grunt integriert werden. Dafür muss das Grunt-Plugin für die TypeScript-Ausführung wie in Quellcodeauszug 3.2.4 installiert werden:

Quellcodeauszug 3.2.4: Installation des TypeScript Grunt-Plugins

```
$ npm install --save-dev grunt-ts
```

Danach wird in der Grunt-Konfigurationsdatei „Gruntfile.js“ im Wurzelverzeichnis des Projektes eine neue Konfiguration für grunt-ts angelegt. Diese legt einen Task an, welcher den TypeScript-Transpiler mit einer hier festgelegten „tsconfig.json“-Datei aufruft, zu sehen in Quellcodeauszug 3.2.5.

Quellcodeauszug 3.2.5: Grunt-Task für die Ausführung von grunt-ts

```
1 ts: {
2   default: {
3     tsconfig: 'tsconfig.json'
4   }
5 },
```

Nun muss die Konfiguration im Build-Task aufgerufen werden, indem der ts-Task aufgerufen wird. Der Build-Task ist dabei nur eine Liste kleinerer Tasks, welche in einer vordefinierten Reihenfolge ausgeführt werden und als Resultat das fertig erstellte Projekt beinhaltet, welches nun publiziert werden kann. Ein solcher Task könnte wie in Quellcodeauszug 3.2.6 definiert werden:

Quellcodeauszug 3.2.6: Beispielhafter Build-Task in Grunt

```
1 grunt.registerTask('build', [
2   'clean:dist',
3   'ts',
4   'jshint',
5   'copy:dist'
6 ]);
```

Ein paar letzte Modifikationen müssen noch durchgeführt werden. Zu beachten ist, dass alle fertig transpilierten JavaScript-Dateien jetzt nicht mehr im Ordner „src“, sondern im Ordner „build“ zu finden sind. Wenn beispielsweise Yeoman für die Vereinheitlichung der Organisation genutzt werden soll, sähe der Yeoman-Eintrag in dem Gruntfile wie in Quellcodeauszug 3.2.7 aus:

Quellcodeauszug 3.2.7: Yeoman-Konfiguration für Grunt

```
1 yeoman: {
2   dist: 'dist',
3   app: 'build'
4 },
```

Außerdem kann nun der Ordner „build“ zu einem eventuellen Clean-Task hinzugefügt werden. Anschließend sollte die Ausführung des Build-Tasks via

```
$ grunt build
```

die TypeScript-Dateien im Ordner „src“ automatisch nach JavaScript transpilieren und das Ergebnis im Ordner „build“ ablegen, wo sie von den Grunt-Tasks weiter verarbeitet werden.

3.2.3 JavaScript nach TypeScript portieren

Das zu portierende Projekt befindet sich derzeit in einem instabilen Zustand. In dem „src“-Ordner des Projektes befinden sich nur noch .js-Dateien, der Build-Prozess kann

dort allerdings nur mit .ts-Dateien etwas anfangen, da nur diese vom TypeScript-Transpiler übersetzt werden. Daher müssen alle diese Dateien mit der neuen Endung .ts versehen werden.

Der Build-Task des Projektes wird nun wieder erfolgreich ausgeführt werden können und das Projekt befindet sich jetzt im Prinzip wieder im Ausgangsstadium - das Endergebnis sieht genau so aus wie vorher und funktioniert auch genau so, da JavaScript auch gültiges TypeScript ist.

Allerdings gilt es jetzt nach und nach, die einzelnen Objekte des Projektes nach TypeScript umzuschreiben. Quellcodeauszug 3.2.8 ist ein Beispiel für eine zu übersetzende Komponente:

Quellcodeauszug 3.2.8: Ausgangsbeispiel JavaScript

```
1  (function () {
2    'use strict';
3
4    angular.module('appModule').component('testComponent', {
5      bindings: {
6        textBinding: '@',
7        dataBinding: '<',
8        functionBinding: '&'
9      },
10     controller: TestComponentController,
11     templateUrl: 'test-component.html'
12   });
13
14   function TestComponentController () {
15     var _this = this;
16
17     _this.add = function (){
18       _this.functionBinding();
19     };
20   }
21 })();
```

Von hier an wird Programmiererfahrung und im Bestfall sogar Erfahrung mit AngularJS zwingend notwendig.

Spätestens jetzt fällt auf, dass ebenfalls, wie in anderen Programmiersprachen (beispielsweise Java) üblich, Interfaces definiert werden müssen, um die Typisierung konsequent durchführen zu können. Hierfür wurde bereits vorgesorgt (siehe Abschnitt 3.2.1), indem in der „tsconfig.json“-Datei vermerkt wurde, dass der Transpiler im Ordner „types“ nach gültigen Type-Definitionen suchen und diese dann automatisch in jeder Quellcode-Datei zugänglich machen soll. Es muss nun also ein Ordner „types“ im Wurzelverzeichnis erstellt werden und dort eine Datei angelegt werden, die häufig einfach nur „index.d.ts“ benannt wird. Hier werden die Interfaces für die Bindings der Komponente definiert (siehe Quellcodeauszug 3.2.9):

Quellcodeauszug 3.2.9: Type-Definitionen für das Beispielprojekt

```
1  interface ITestComponentBindings {
2    textBinding: string;
3    dataBinding: number;
4    functionBinding: () => any;
5  }
6
7  interface ITestComponentController extends
   ↪  ITestComponentBindings {
8    add(): void;
9  }
```

Anschließend kann die Component als eine Klasse in TypeScript wie in Quellcodeauszug 3.2.10 fertig definiert werden:

Quellcodeauszug 3.2.10: Beispielkomponente in TypeScript

```
1  class TestComponentController implements
   ↪  ITestComponentController {
2
3    public textBinding: string;
4    public dataBinding: number;
5    public functionBinding: () => any;
```

```
6
7     constructor() {
8       this.textBinding = '';
9       this.dataBinding = 0;
10    }
11
12    add(): void {
13      this.functionBinding();
14    }
15  }
```

Auf diese Weise können nun nach und nach alle Bestandteile eines Projektes typisiert und damit in TypeScript übersetzt werden. Soll sicher gestellt werden, dass dabei auch keine Stelle zu typisieren vergessen wurde, kann in der „tsconfig.json“ Datei noch der Eintrag

```
noImplicitAny: true
```

hinterlegt werden. Wenn der Transpiler nun ausgeführt wird und eine implizite Deklaration einer Variable mit dem Typ any bemerkt, wird er eine Warnung ausgeben, statt diese einfach zur Kenntnis zu nehmen.

3.3 Upgrade auf AngularJS 1.5 (optional)

Alle AngularJS-Projekte, aber auch Angular-Projekte haben eine Modul-Struktur gemeinsam. Ein Modul ist dabei ein Bereich einer Webseite, der strikt von anderen Bereichen abgegrenzt werden kann. Üblicherweise sind das Sektionen wie das Gästebuch, ein Newsbereich, die Administrationsoberfläche und Ähnliches. Diese sind voneinander unabhängig und können daher lose verzahnt werden, wie es für das Projekt erforderlich ist. Diese Module werden anschließend wieder in Unterbestandteile gegliedert, welche sich von Projektarchitektur zu Projektarchitektur unterscheiden.

Um den folgenden, optionalen Upgrade-Vorgang eines AngularJS-Projektes, welches auf einer AngularJS-Version < 1.5 entwickelt wurde, auf das neuere Konzept

ab Version 1.5 nachvollziehen zu können, ist es dementsprechend notwendig, die Architekturunterschiede zwischen beiden Versionen zu verstehen, weshalb diese im Folgenden aufgezeigt werden.

3.3.1 Model-View-Controller Architektur (MVC)

Die Model-View-Controller Architektur [Kow] beschreibt ein Software-Design, bei dem Teile einer Applikation in drei Komponenten unterteilt werden:

Die View ist dabei ein Bestandteil, welche sich hauptsächlich auf die Aufbereitung interner Informationen in eine für Menschen erfassbare Darstellung kümmert. Im Gegensatz dazu steht das Model, welches sämtliche internen Daten eines Programmes beinhaltet. Der Controller ist schließlich eine Abstraktionsschicht zwischen Model und View, welcher Schnittstellen für die jeweilige Seite zur Verfügung stellt, um die Daten zu verwalten, welche die jeweils andere Seite für die Arbeit benötigt. Er steuert den Datenfluss zwischen beiden Bestandteilen und sorgt somit für die Kommunikation im Programm.

Dieses Paradigma existiert in mehreren Varianten, welche sich zumeist minimal in der internen Arbeitsweise unterscheiden. Im Webumfeld hat sich dabei die CVM-Variante etabliert ([Kow]). Hierbei kann der Controller direkt mit View und Model kommunizieren, allerdings kann auch die View direkt mit einem Model kommunizieren, um beispielsweise die in einem Formular eingegebenen Daten direkt abzulegen.

In Abbildung 3.1 wird ein Sequenzdiagramm dargestellt, welches dieses Paradigma visualisiert.

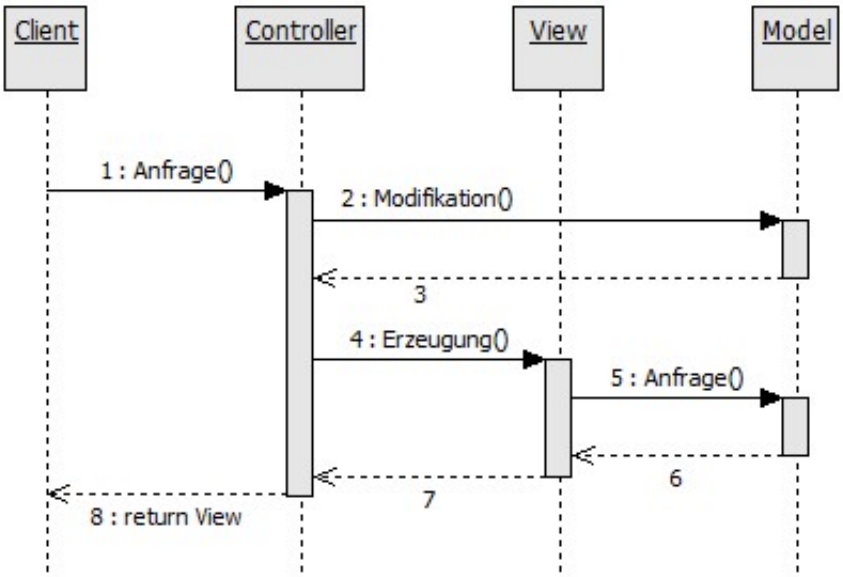

Abbildung 3.1: Darstellung CVM

Dieses Prinzip wurde vor allem wegen seiner strikten Trennung zwischen Anwendungslogik und grafischer Aufbereitung beliebt und häufig implementiert.

3.3.2 Component Architektur

> The main primitive is this idea of a „component“. I think everyone has some notion of what a component is. The idea is that it should be an atomic UI piece that is composable and reusable, and should work with other pieces in a way that's performant.
>
> Brian Ford

Bei der Component Architektur [Kir] sind die Aufgaben immernoch strikt getrennt, allerdings wird dies nicht mehr forciert. Statt Model, View und Controller gibt es hier nur noch Komponenten, welche sich wie eine View oder wie ein Model verhalten. Controller entfallen hier, da die Komponenten über Events zum Datenfluss angeregt werden.

Ob eine Komponente nun die Funktionen einer View oder eines Models übernimmt ist nicht mehr festgelegt. Theoretisch könnte eine Komponente sogar beides gleichzeitig übernehmen, wovon allerdings zugunsten besserer Wartbarkeit abgeraten wird. Im Angular-Umfeld haben sich daher die Begriffe Container Component für Komponenten, welche sich wie Models verhalten und Presentational Component für Komponenten, welche sich wie Views verhalten herausgebildet. Es wird hier dringend das Single-Responsibility-Prinzip ([Ell]) empfohlen, d.h. das eine Komponente ihren eingeschränkten Arbeitsbereich hat und auch immer nur für eine Aufgabe zuständig ist. Dies ermöglicht ein einfaches Austauschen von Komponenten bei Bedarf, ohne die Integrität des Projektes zu beeinträchtigen. Die einzelnen Komponenten stellen zueinander nur Blackboxes dar, sie wissen, über welche Methoden sie Daten eingeben und was sie als Ergebnis erhalten, sie sollen jedoch nicht wissen, wie diese Daten intern verarbeitet werden.

Die Komponenten werden nun untereinander verbunden. Eine Webseite des sozialen Netzwerkes Twitter beispielsweise könnte mehrere Komponenten beinhalten, wie die Anzeige von Nutzerinformationen oder die Darstellung von Tweets. Twitter besitzt jedoch mehrere Seiten, welche Tweets aufbereiten und anzeigen wollen. So gibt es ein Widget, welches Twitter zur Verfügung stellt, um auf externen Webseiten einzelne Tweets oder die Timeline eines bestimmten Nutzers einzublenden. Auch diese Widgets hängen von der selben Komponente ab, welche bereits auf der Twitter-internen Webseite die Tweets eines Nutzers darstellt.

Da Komponenten aber nicht nur Daten ausgeben, sondern auch Daten aufnehmen und sie an in der Hierarchie weiter oben angeordnete Komponenten weiterleiten, kann dieses Prinzip mithilfe einer Baumstruktur visualisiert werden. Ein gutes Beispiel für die Weiterleitung von Daten ist ein Gästebuch. Das Formular, welches einen neuen Beitrag aufnimmt, leitet diese Daten an die Container Komponente weiter, welche zum Ersten das Abspeichern in einer Datenverwaltung veranlasst, und anschließend eine neue Komponente zur Anzeige des neuen Eintrages erzeugt und darstellt.

Bei der Visualisierung eines solchen Baumes entsteht ein sogenannter Component Tree, welcher in Abbildung 3.2 dargestellt wird.

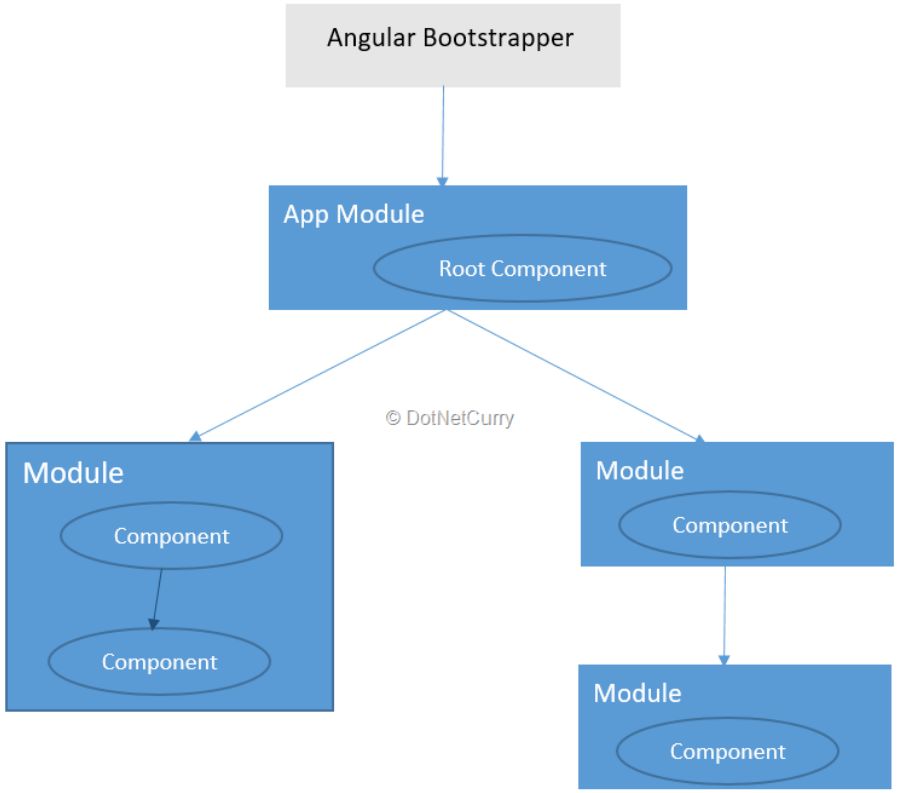

Abbildung 3.2: Darstellung Component Architektur

3.3.3 MVC und Komponenten im Angular-Umfeld

Der Aufbau eines AngularJS-Projektes wurde strikt nach der MVC Architektur geführt. Diese Architektur war damals neu und erfreute sich großer Beliebtheit. Nach einigen Jahren der massiven Anwendung stellte sich dieses Konzept jedoch als zu steif heraus, nicht immer kann man klar zwischen Model und View unterscheiden und häufig mussten Controller entwickelt werden, die im Endeffekt nichts tun, als Variablen von einem Model auch einer View zur Verfügung zu stellen und umgekehrt.

Als schließlich mit der Entwicklung von Angular begonnen wurde stand fest, dass hier eine neue Architektur eingeführt werden musste, welche die Übergänge etwas fließender gestaltet. Google entschied sich hierbei für die Component Architektur. Um den Übergang so fließend wie möglich zu gestalten, führten sie eben diese Architektur ebenfalls in AngularJS ab Version 1.5 ein, allerdings war sie hier nur

optional. Dies sollte den Entwicklern ermöglichen, ihre Models und Views nach und nach in Komponenten umzuschreiben, ohne dabei die Funktionalität des Projektes zu beeinträchtigen.

In Angular wurde schließlich die Component Architektur zur Pflicht, es gibt schlicht keine Views und Models mehr, und erst recht keine Controller. Falls also Projekte portiert werden müssen, welche noch auf der alten MVC Architektur basieren und allerdings eine AngularJS-Version 1.5 oder neuer verwendet wird, kann es von Vorteil sein, erst die Architektur intern umzustellen und zu sehen, dass das Projekt anschließend noch funktioniert, bevor man den großen Umstieg auf Angular beginnt.

3.3.4 Refactoring zur Einführung von Komponenten in AngularJS 1.5+

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. With refactoring you can take a bad design, chaos even, and rework it into well-designed code. Each step is simple, even simplistic. [...] Yet the cumulative effect of these small changes can radically improve the design. It is the exact reverse of the normal notion of software decay.
>
> Martin Fowler

Dieser Abschnitt erfordert tiefgreifendes Wissen über AngularJS und dessen Funktionsweise im Speziellen, um die Umstellung ohne größere Komplikationen durchführen zu können.

Zur Einführung sei gesagt, dass Komponenten in Angular und AngularJS aus drei wichtigen Bestandteilen bestehen:

- Ein Interface, welches bestimmt, auf welche Eigenschaften der Komponente von einer anderen Komponente aus lesend zugegriffen werden kann und außerdem, in welche Eigenschaften der Komponente Werte von einer anderen Komponente abgelegt werden können.

- Ein Template, welches den HTML- und CSS-Code zur Darstellung der Komponente bereitstellt. Dieses kann entweder direkt im Quellcode der Komponente liegen. Besser ist es jedoch, wenn dieser Code in eine separate HTML-Datei und eventuell auch CSS-Dateien ausgelagert wird.

- Die Logik der Komponente, welche festlegt, wie sich die Komponente in speziellen Fällen verhält.

Während eine Komponente in Angular später eine Klasse sein wird, muss eine Komponente in AngularJS noch in zwei Klassen und das Template aufgeteilt werden. Die Funktionsweise ist allerdings identisch zu einer Komponente in Angular. In AngularJS übernimmt eine Direktive die Aufgabe der Verwaltung der Eingabe- und Ausgabeeigenschaften, wohingegen ein Controller die Funktionsweise und Logik der Komponente abbildet.

Der Übersichtlichkeit halber sollte jede Komponente in einen extra Ordner geschachtelt werden, wobei in diesem Ordner eine Datei für den Controller, eine für die Direktive und sämtliche Dateien für das Template liegen sollten. Diese Dateien sind einheitlich nach dem Namen der Komponente benannt, beispielsweise „meine-testkomponente.controller.ts“, „meine-testkomponente.directive.ts“, „meine-testkomponente.html“.

Als Erstes kann an jeder Stelle, wo die View, welche nun von der Komponente abgelöst werden soll, zu finden war, stattdessen die Komponente eingebunden werden. Üblicherweise wurde hier häufig der ng-include Befehl genutzt. Dieser kann nun einfach durch den Namen der Direktive als HTML-Tag ersetzt werden. Die Anbindung einer Komponente namens „meine-testkomponente“ sähe beispielsweise wie in Quellcodeauszug 3.3.1 aus.

Quellcodeauszug 3.3.1: Anbindung meine-testkomponente in AngularJS 1.5+

```
1 <!-- vorher -->
2 <div ng-include="'meine-testkomponente.html'">
3 </div>
4 <!-- nachher -->
5 <meine-testkomponente></meine-testkomponente>
```

Als nächstes müssen Input- und Output-Eigenschaften definiert werden. Dies ist eine Neuerung, welche mit Angular eingeführt wurde. In AngularJS gab es einen globalen Scope, in dem man sämtliche Services finden konnte, und üblicherweise wurden diese auch unkontrolliert durch alle Controller und Views angesteuert. In Angular wurde auf eine Kontrolle umgestellt, welche definiert, welche Eigenschaften der jeweiligen Komponente an andere Komponenten weitergeleitet werden, welche Eigenschaften von anderen Komponenten beschrieben werden können und welche Dienste eine Komponente genau benötigt, um zu funktionieren. Diese Dienste werden dann von Angular korrekt weitergeleitet. Alle anderen Eigenschaften und Dienste als diejenigen, die zur Verwendung durch andere Komponenten freigegeben wurden, sind nicht ansteuerbar, die Komponenten werden tatsächlich voneinander isoliert.

In AngularJS-Projekten wurde dieses Konzept nur lose umgesetzt, weshalb Projekte erst darauf umgestellt werden müssen. In Quellcodeauszug 3.3.2 wird ein Text in der vom Nutzer gewählten Sprache ausgegeben, welche dafür allerdings erst abgefragt werden muss. Zudem wird dies auch nur dann getan, wenn die Komponente tatsächlich auch angezeigt werden soll:

Quellcodeauszug 3.3.2: Beispiel für unisolierte Zugriffe unter AngularJS < 1.5

```
1 <div ng-show="globalCtrl.isMeineTestkomponenteVisible()">
2     <p>
3         {{ctrl.getText(globalCtrl.getLanguage())}}
4     </p>
5 </div>
```

Nun müssen alle Zugriffe auf Controller und Dienste außerhalb der Komponente selbst in den Controller der Komponente verlegt und zusätzlich Eigenschaften angelegt werden, welche als Input- bzw. Output-Eigenschaften deklariert werden. Unter Angular geschieht dies mittels der „@input“ bzw. „@output“-Dekoratoren, in AngularJS muss noch eine Eigenschaft „bindToController“ in der Direktive angelegt werden, welche die Eingabe- und Ausgabeeigenschaften beinhaltet und mittels dafür vorgesehener Zeichen „=“ und „&“ diese als Eingabe- bzw. Ausgabeeigenschaften definiert.

In Quellcodeauszug 3.3.3 wird dargestellt, wie so eine Definition in der Direktive aussieht.

Quellcodeauszug 3.3.3: Eingabe- und Ausgabeeigenschaften unter AngularJS ab Version 1.5

```
module.directive('meine-testkomponente', function() {
  return {
    scope: true,
    controller: 'MeineTestkomponenteCtrl',
    controllerAs: 'ctrl',
    bindToController: {
      visible: '=',
      language: '='
    },
    templateUrl: 'meine-testkomponente.html'
  };
});
```

Diese Eingabeeigenschaften müssen nun nur noch mit den Werten gefüllt werden, woraufhin sie direkt aus der Komponente abgefragt werden können und nicht mehr auf die externen Dienste zugreifen müssen. Dafür werden die Eigenschaften an der Stelle, an der die Komponente im Haupt-HTML-Dokument (meistens index.html) eingebunden wird, befüllt. In Quellcodeauszug 3.3.4 ist dies zu sehen.

Quellcodeauszug 3.3.4: Korrekte Verwendung von Input- und Output-Eigenschaften unter AngularJS 1.5

```
<!-- index.html
     das zentrale Modul zur Darstellung der Webseite
     hat als Einziges noch Zugriff auf den
     globalen Controller -->
<meine-testkomponente
  visible="globalCtrl.isMeineTestkomponenteVisible()"
  language="globalCtrl.getLanguage()">
</meine-testkomponente>
<!-- meine-testkomponente.html -->
<div ng-show="ctrl.visible">
  <p>{{ctrl.getText(ctrl.language)}}</p>
</div>
```

Können Aktionen nicht mit Variablen, sondern müssen mit Funktionen abgebildet werden, können diese im Komponenten-eigenen Controller definiert werden und, wie oben, auch aus der View bzw. dem Template heraus aufgerufen werden. Wichtig ist, das kein Dienst mehr aufgerufen wird, welcher nicht explizit in der Komponente benötigt wird. Eine Ausgabe-Eigenschaft wird entsprechend mit „&“ im bindToController-Objekt vermerkt.

Sind schließlich alle direkten Zugriffe auf externe Komponenten und Dienste entsprechend gekapselt und umgestellt, kann der Hinweis „scope: true“ in der Direktive auf „scope: “ umgestellt werden. Dies isoliert den Scope der Komponente. Wenn das Projekt dann noch wie vorgesehen funktioniert, wurden alle Umstellungen richtig vorgenommen.

Nachdem diese Schritte umgesetzt wurden, ist das Projekt ein gültiges AngularJS-Projekt mit Komponenten, welches nun verhältnismäßig unkompliziert nach Angular migriert werden kann. Sicherlich können auf dem Weg noch weitere Probleme auftreten. Je nach Komplexität des Projektes können immer Fälle auftreten, welche vorher nur schwer bedacht werden können. Jedoch sind diese Probleme voraussichtlich immer auf nicht vollständig erfolgte Umsetzung der oben beschriebenen Schritte zurückzuführen.

3.4 Integration von ngUpgrade

Bei ngUpgrade handelt es sich um ein Modul, welches die parallele Verwendung von AngularJS-Komponenten und Services neben den neu von Angular eingeführten Komponenten und Services erlaubt. Dieses Modul ermöglicht erst die zeitunabhängige Migration von AngularJS nach Angular. Mit ngUpgrade wird es daher möglich, jedes Modul und jede Komponente nach und nach zu migrieren, ohne das das Projekt dadurch unproduktiv wird.

Bei der Ausführung des UpgradeModule, welches ngUpgrade zur Verfügung stellt, werden sowohl Angular, als auch AngularJS ausgeführt, es findet also keine Emulierung statt, alle Funktionen beider Frameworks werden wie erwartet ausgeführt. Außerdem wird die Kommunikation zwischen den Komponenten und Diensten beider Framework-Versionen gewährleistet, wobei auch die Abhängigkeiten aufgelöst werden

und das Document Object Model (DOM) korrekt abgebildet wird. Wichtig dabei ist, dass die AngularJS-Projekte auf Komponenten umgestellt wurden, da ngUpgrade nur die Eingabe- und AusgabeSchnittstellen überbrückt, nicht aber die Zugriffe über den globalen Scope.

Ein wichtiger Punkt ist, dass eine Darstellung im DOM nur in einem Framework existieren kann. Ein AngularJS-Template kann daher ein Angular-Template verwenden und umgekehrt. Allerdings kann ein AngularJS-Template nicht auf eine Angular- und eine AngularJS-Komponente gleichzeitig zugreifen, was aber ohnehin keinem guten Angular-Programmierstil entspricht. Ein Template sollte immer nur an eine Komponente gebunden sein. Ein AngularJS-Template, welches auf eine Angular-Komponente zugreift, bleibt weiterhin ein AngularJS-Objekt, nur die Komponente wird von Angular betrieben.

Die Change Detection von AngularJS bzw. Angular dagegen wird mittels ngUpgrade vereinfacht. Während in AngularJS noch regelmäßig

```
scope.$apply()
```

aufgerufen werden musste, um Änderungen an Variablen auch auf der Webseite ersichtlich zu machen, ist dies in Angular nicht mehr nötig. ngUpgrade übernimmt diese Neuerung auch auf AngularJS-Templates und Komponenten, der manuelle Aufruf der Apply-Methode ist nicht mehr nötig, verursacht jedoch auch keine Fehler.

3.4.1 Einbindung von ngUpgrade

Zunächst muss ngUpgrade installiert werden, was unter Verwendung von NPM schnell geschehen ist:

```
$ npm install @angular/upgrade --save
```

Anschließend wird die Einbindung der AngularJS-App an der Wurzel des Projektes umgestellt. Hier wurde üblicherweise das oberste Modul bzw. die App eingebunden, welche dann wiederum alle nachfolgenden Module einbindet, wie es das Projekt vorsieht. Die Einbindung übernimmt nun nicht mehr AngularJS selbst, sondern ngUpgrade, mittels der UpgradeAdapter Klasse. Dies bedeutet allerdings, dass das

AngularJS-Projekt weiterhin als AngularJS-Projekt läuft, was zur Folge hat, dass alle Angular-Komponenten automatisch durch das UpgradeModule nach AngularJS herabgestuft werden. Dies muss zwar während des Upgrade-Vorgangs beachtet werden, bringt allerdings keine Nachteile mit sich.

Üblicherweise existiert an der Wurzel des Projektes ein App-Objekt, welches beispielsweise so aussehen kann:

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

Diese App wird üblicherweise mittels des ng-app-Attributes in die Hauptseite des Projektes eingebunden. Bei der Verwendung von ngUpgrade wird diese Einbindung in der HTML-Datei des Projektes jedoch entfernt und stattdessen die Einbindung über den UpgradeAdapter vorgenommen.

Quellcodeauszug 3.4.1: Einbindung der Projekt-App mittels UpgradeAdapter

```
1 import { UpgradeAdapter } from '@angular/upgrade';
2
3 var adapter = new UpgradeAdapter();
4 var app = angular.module('meine-app', []);
5
6 adapter.bootstrap(document.body, ['meine-app']);
```

Dieser Adapter sollte natürlich in eine separate Datei ausgelagert werden, damit er von mehreren Stellen ohne Weiteres zugänglich ist, das obige Beispiel dient daher nur der Veranschaulichung.

3.4.2 Migration einer Komponente von AngularJS nach Angular

Von nun an ist ngUpgrade ein wichtiger Bestandteil der App und es kann mit dem Upgrade-Vorgang begonnen werden. Dafür wird nun eine Komponente von AngularJS nach Angular migriert. Eine Angular-Komponente sieht dabei fertig migriert wie in Quellcodeauszug 3.4.2 aus.

Quellcodeauszug 3.4.2: Eine beispielhafte Angular-Komponente

```
@Component({
  selector: 'meine-komponente',
  template: `
    <h2>{{welcome}}</h2>
  `
})
class MeineKomponente {
  @Input()
  welcome: string;
}
```

Das Template wird hierbei meist in eine separate HTML-Datei ausgelagert, welche dem bereits beschriebenen Namensschema folgt (siehe Abschnitt 3.1). Der Upgrade-Vorgang der AngularJS-Komponenten nach Angular sollte, sofern bereits vorher auf die in AngularJS 1.5 verwendeten Komponenten umgestellt wurde, nur noch eine Sache der Anpassung an die neuen Schlüsselwörter sein. Dabei werden Komponenten mit dem Dekorator @Component, sowie alle Eingabe- und Ausgabe-Eigenschaften mit den Dekoratoren @Input und @Output versehen. Zudem handelt es sich bei der Komponente nun tatsächlich um eine echte Klasse.

Nun gilt es zu beachten, dass dies zwar eine gültige Angular-Komponente ist, unser Projekt allerdings, wie oben bereits erwähnt, noch unter AngularJS läuft, weshalb diese Komponente von ngUpgrade einem downgrade unterzogen werden muss, damit sie sich zumindest der App gegenüber wieder wie eine AngularJS-Komponente verhält. Um dies zu bewerkstelligen, wird die Direktive der Komponente entsprechend angepasst.

Quellcodeauszug 3.4.3: Downgrade einer Angular-Komponente nach AngularJS

```
app.directive('meineKomponente',
  adapter.downgradeNg2Component(MeineKomponente));
```

3.4.3 Upgrade der Services von AngularJS nach Angular

Services stellen im Angular-Umfeld Klassen dar, welche bestimmte Funktionen zentralisieren, welche von mehreren Komponenten benötigt werden, die allerdings keine Darstellung auf der Webseite erfordern. Ein gutes Beispiel stellt dafür der von Angular bereitgestellte $http Service dar, welcher die Kommunikation mittels des HTTP bzw. HTTPS-Protokolls ermöglicht. Solche Services werden üblicherweise global verwaltet und von der Komponente zur Ladezeit dynamisch angefordert.

Auch solche Services existieren natürlich sowohl im AngularJS- als auch im Angular-Umfeld und müssen daher migriert werden.

Ein Service folgt hierbei keiner speziellen Semantik und wird daher einfach als formlose Klasse implementiert. Allerdings muss hierbei natürlich auch erst einmal von AngularJS nach Angular portiert werden, indem der Service in eine Klasse umgeschrieben wird, was allerdings keine großen Hürden aufwerfen dürfte.

Anschließend muss der Service dem UpgradeAdapter bekannt gemacht werden, da alle Services in Angular global verwaltet werden, wohingegen sie in AngularJS noch von jeder Komponente separat angefordert wurden. Dabei hilft der Aufruf

```
adapter.addProvider(MeineKomponenteService);
```

Nun stellt sich die Frage, wie vorgegangen werden soll, wenn der Service, der bereits nach Angular portiert wurde, von einem weiteren Service abhängt, welcher bislang allerdings nur in AngularJS existiert und daher dem Angular-Framework nicht bekannt ist. Dafür stellt der Adapter wiederum eine Methode zum Aufwerten des AngularJS-Service zu einem Angular-Service bereit. Im Quellcodeauszug 3.4.4 wird dazu ein Beispiel angeführt:

Quellcodeauszug 3.4.4: Aufwertung eines AngularJS-Services nach Angular

```
1  class MeineKomponenteService {
2
3    // hier wird @Inject verwendet, da dieser Dienst
4    // bereits in Angular laeuft, der benötigte
```

```
5     // Dienst aber noch in AngularJS läuft und es
6     // daher noch keine Typisierung in Angular gibt.
7     constructor(@Inject('DataService') dataService) {
8     }
9   }
10
11  // hier folgt der in AngularJS entwickelte Dienst,
12  // von dem der obenstehende Dienst abhaengt
13  app.service('DataService', () => {
14  });
15
16  // und so wird er dem UpgradeAdapter bekannt gemacht
17  adapter.upgradeNg1Provider('DataService');
18
19  // noch eleganter geht es allerdings, wenn auch
20  // der DataService bereits als TypeScript-Klasse
21  // angelegt wurde. In dem Fall waere dies bevorzugt.
22
23  class DataService {
24  }
25
26  app.service('DataService', DataService);
27
28  adapter.upgradeNg1Provider('DataService', {asToken:
    ↪  DataService});
29
30  // in diesem Fall kann die MeineKomponenteService
31  // Klasse wie folgt definiert werden
32
33  @Injectable()
34  class MeineKomponenteService {
35
36    constructor(dataService: DataService) {
37    }
38  }
```

Nun ist natürlich auch die umgekehrte Situation denkbar, ein AngularJS-Service benötigt also einen Angular-Service als Abhängigkeit. Dies wird mittels zweier kurzer Zeilen bewerkstelligt, denn wiederum stellt ngUpgrade bereits alle Mittel zur Verfügung:

Quellcodeauszug 3.4.5: Herabstufung eines Angular-Service nach AngularJS

```
1 app.factory('MeineKomponenteService',
2   adapter.downgradeNg2Provider(MeineKomponenteService));
```

3.5 Erstellen eines neuen Angular-Projektes

Da nun alle Bestandteile des AngularJS-Projektes nach Angular überführt wurden und das Projekt im Prinzip nur noch durch den UpgradeAdapter des ngUpgrade Paketes lauffähig ist, können die einzelnen Bestandteile nun in ein frisches Angular-Projekt überführt werden. Dies stellt nunmehr nur noch eine Fleißarbeit dar, stellt allerdings sicher, dass auch Hilfsmittel wie ein Modulmanager und die neuesten Werkzeuge für den Build-Prozess eingesetzt werden.

Hierfür wird mittels

```
$ ng new MeinProjekt
```

ein neues Projekt angelegt. Anschließend wird für jede Komponente des Projektes

```
$ ng g c MeineKomponente...
```

ausgeführt. Die Angular CLI legt dabei alle Komponenten selbstständig an, es müssen nur noch die Platzhalter-Dateien, welche von der Angular CLI angelegt wurden, durch die Komponenten aus dem alten Projekt ausgetauscht werden.

Gleichermaßen verfährt man mit den Modulen des alten Projektes, und auch mit den Services verhält es sich nicht anders.

Das Ergebnis ist das fertig migrierte Angular-Projekt, welches nun mittels

```
$ ng serve
```

im Browser ersichtlich sein sollte.

Kapitel 4

Ergebnisse

Als Ergebnis der Umstellung des Projektes von AngularJS nach Angular erhält man schließlich ein vollwertiges Angular-Projekt. Ob man dabei zuerst die optionale Umstellung nach AngularJS durchgeführt hat, oder sogar ein komplett neues Projekt in Angular angelegt und alle Komponenten neu entwickelt hat, spielt dabei keine Rolle.

Das hierbei entstandene Projekt wurde nicht nur seitens des Quellcodes auf Angular umgestellt, sondern auch hinsichtlich der Werkzeuge, welche bei der Installation oder beim Erzeugen des finalen Projektes benötigt werden.

Eigene Betrachtungen bezüglich der Performanceunterschiede zwischen AngularJS und Angular wurden hierbei nicht getätigt, jedoch fühlen sich mittels Angular entwickelte Webseiten für den Nutzer häufig flüssiger an. Ob dies aber an Angular liegt steht nicht fest.

Der Zeitaufwand bei der Umstellung ist dabei jedoch nicht von der Hand zu weisen. Je nach Größe des Projektes erfordert dies einigen Aufwand, und auch bei größeren Projekten kann es, wenn genügend Zeit zur Verfügung steht, für das Projekt letztendlich von Vorteil sein, es komplett neu in Angular zu entwickeln. Generell gilt allerdings der Grundsatz: Never touch a running system. Ein dringender Grund zur Umstellung besteht erst dann, wenn eine gravierende Sicherheitslücke in den AngularJS-Paketen, welche in einem Projekt verwendet werden, gefunden wird, oder benötigte Funktionen als deprecated vermerkt oder sogar ganz ausgebaut werden, denn in diesem Fall wird die Instandhaltung des Projektes erschwert, vielleicht sogar verhindert und eine Umstellung nach Angular sollte in Betracht gezogen werden.

Kapitel 5

Zusammenfassung

Nachdem wir bei der GISA GmbH mehrere Projekte in Angular komplett neu entwickelt haben und auch einige kleinere Projekte von AngularJS nach Angular überführt haben, stellte sich die Frage, wie man ein größeres Projekt überführen könnte, ohne dabei dessen Lauffähigkeit zu beeinträchtigen. Während kleine Projekte in einem Anlauf übertragen werden können, ist dies bei größeren Projekten nicht möglich. Diese Arbeit ist ein Resultat dieser Fragestellung.

In dieser Arbeit wurden die Ansätze erläutert, welche wir dabei verfolgt haben. Die Umstellung erfordert in jedem Fall einen größeren Aufwand, sei es nun durch eine vollständige Neuentwicklung oder eine Umstellung, allerdings sind auch die positiven Aspekte nicht von der Hand zu weisen. Bereits früher gab es immer Paketunstimmigkeiten, da die GISA GmbH in bereits bestehenden Projekten veraltete Paketversionen verwendete, deren Aktualisierung Zeit in Anspruch genommen hätte, die vom Auftraggeber nicht bezahlt wurde, oder ihm einfach nicht wichtig genug war. Die Umstellung von AngularJS nach Angular löste die meisten dieser Probleme auf und sichert eine nachhaltige Stabilität des Projektes, sowohl in Hinsicht auf zukünftige Paketversionen, als auch auf Angular selbst.

Sehr wichtig ist hierbei auch die Aktualität im Hinblick auf den derzeitigen Standard. Bei der Ausschreibung und Werbung wurde oft auf Projekte referenziert, welche noch in AngularJS entwickelt wurden. In Zeiten, in denen Angular allerdings standardisiert ist, stellt sich dem Auftraggeber dann die Frage, warum hier nicht auf Angular gesetzt wird. Die Umstellung auf Angular kommt also nicht nur der Kompatibilität zu Gute, sondern auch der Wirkung gegenüber potenziellen Interessenten.

Auch verbessert hat sich die generelle Verfügbarkeit auf mobilen Endgeräten. Früher floss viel Arbeitszeit in explizite Fehlermeldungen oder Neuanforderungen in Hinsicht auf mobile Verfügbarkeit, da zu AngularJS-Zeiten noch auf zusätzliche Pakete mit zusätzlich benötigtem Code gesetzt wurde. Diese Funktionalität ist in Angular

allerdings bereits nativ vorhanden, weshalb viele der früher nötigen Funktionen bereits ohne mehr Aufwand vorhanden sind, was langfristig Entwicklungszeit einspart.

Eine größere Umstellung erfordert die Umstellung auf TypeScript. Zwar bringt TypeScript als Solche bereits enorme Hilfestellungen für Entwickler mit, wie beispielsweise starke Typisierung oder Objektorientierung, jedoch stellt die Umstellung von JavaScript nach TypeScript eine Hemmschwelle dar, welche es zu übertreten gilt und erneut Einarbeitungszeit fordert. Gerade die Umstellung von JavaScript, welches keine Typen besitzt, zu TypeScript gestaltet sich als kompliziert, da viele Angular-unabhängige Paketabhängigkeiten häufig auch gar keine Typendefinitionen für TypeScript mitbringen und man diese entweder aufwändig selbst erstellen muss, oder hier lieber auf die Absicherung durch TypeScript verzichtet. Im Webumfeld sind stark typisierte Sprachen eher selten, und Entwickler, welche von der Pike an im Webumfeld aufgewachsen sind, haben mit stark typisierten Ansätzen eine ganz besondere Herausforderung gefunden, welche in unserem Fall bislang jedoch gut gemeistert wurde.

Nach einer dreitägigen Schulung in unserem Haus, welche erfreulicherweise genau in den Zeitraum dieser Arbeit fiel, stellten wir auch fest, dass Angular viele Möglichkeiten bietet, welche AngularJS nicht implementiert hatte, oder einige Funktionen, die in AngularJS zwar schon vorhanden waren, allerdings nicht so zuverlässig funktionierten, auf die wir daher zugunsten besserer Lösungen verzichtet haben. Um eine höhere Entwicklungseffizienz und Projektleistung zu erreichen, gilt es, diese Möglichkeiten zu entdecken und allen Entwicklern des Teams näher zu bringen, um eine konsistente Codestruktur zu gewährleisten. Dies wird ebenfalls viel Zeit und Übung erfordern.

Ebenso gab es allerdings bereits Fälle, in denen beschlossen wurde, ein kleines Angular-Projekt als Lösung einzusetzen, wobei allerdings im Nachhinein festgestellt wurde, dass eine Lösung auf einer ganz anderen Grundlage wie ASP.Net schneller und effizienter gewesen wäre. Hier gilt es zu differenzieren, und das mangelnde Wissen um all die Möglichkeiten, welche das noch verhältnismäßig neue Framework Angular bietet, zu ergänzen, um solche Entscheidungen zielsicherer treffen zu können.

Generell wurde die Schulung zu Angular allerdings positiv aufgenommen und zeitnah bereits einige Projekte in Angular implementiert, wobei einige Entwickler gefallen an der neuen Art der Projektentwicklung gefunden haben. Bei Neuentwicklungen von Projekten wird Angular nun häufiger in Betracht gezogen, und nach und nach wird den

Auftraggebern auch die Aufrüstung von AngularJS nach Angular vorgeschlagen, wobei es von Projekt und Auftraggeber abhängt, ob dies realistisch oder eventuell sogar unnötig ist. In jedem Fall hat sich das Arbeitsgebiet der Entwickler im Webumfeld jedoch stark von AngularJS nach Angular verschoben und die Einfachheit der in den vorherigen Kapiteln beschriebene Schritte zur Umstellung nach Angular trägt sicherlich positiv dazu bei.

Die oben ausführlich beschriebenen Schritte wurden bereits erfolgreich eingesetzt, um Teilprojekte nach Angular zu migrieren, ohne großen Zeit- oder Entwicklungsaufwand zu erzeugen, weshalb bei uns bereits mehrere Projekte nach und nach überführt werden. Größere Probleme sind dabei bislang nicht aufgetreten und auch das Ergebnis spricht für sich, sowohl der Kunde, als auch die Entwickler sind damit zufrieden.

In jedem Fall stellen die Schritte dieser Bachelor-Arbeit bereits jetzt für die GISA GmbH einen Leitfaden dar und repräsentieren anschaulich den Aufstieg von AngularJS nach Angular.

Literaturverzeichnis

[Böh] Robin Böhm. *Was ist Angular und AngularJS?* URL: https://angularjs.de/artikel/was-ist-angular/ (besucht am 12. 12. 2017).

[Com] Computer Hope. *What is Google?* URL: https://www.computerhope.com/jargon/g/google.htm (besucht am 12. 12. 2017).

[Ell] James Ellis-Jones. *Think you understand the Single Responsibility Principle?* URL: https://hackernoon.com/you-dont-understand-the-single-responsibility-principle-abfdd005b137 (besucht am 13. 04. 2018).

[Gooa] Google. *Angular - The Ahead-of-Time (AOT) Compiler.* URL: https://angular.io/guide/aot-compiler (besucht am 15. 06. 2018).

[Goob] Google. *Upgrading from AngularJS.* URL: https://angular.io/guide/upgrade (besucht am 03. 01. 2018).

[Gru] Grunt Team. *Grunt: The JavaScript Task Runner.* URL: https://gruntjs.com/ (besucht am 03. 01. 2018).

[Har] Bruno Hartmann. *AngularJS vs Angular: 4 reasons why and 2 simple ways to migrate.* URL: https://www.uruit.com/blog/2017/07/04/angular-1-vs-2-migrate/ (besucht am 08. 01. 2018).

[ITWa] ITWissen.info. *Framework.* URL: http://www.itwissen.info/Framework-framework.html (besucht am 11. 12. 2017).

[ITWb] ITWissen.info. *Objektorientierte Programmierung.* URL: http://www.itwissen.info/Objektorientierte-Programmierung-object-oriented-programming-OOP.html (besucht am 15. 12. 2017).

[ITWc] ITWissen.info. *Prozedurale Programmierung.* URL: http://www.itwissen.info/Prozedurale-Programmierung-procedural-programming.html (besucht am 15. 12. 2017).

[Jav] JavaScript Kit. *JavaScript Kit- JavaScript Reference.* URL: http://www.javascriptkit.com/jsref/ (besucht am 14. 03. 2018).

[Kir] Ravi Kiran. *Angular application - Architecture overview.* URL: http://www.dotnetcurry.com/angularjs/1400/angular-application-architecture (besucht am 04. 01. 2018).

[Kow] Wolfgang Kowarschick. *Model-View-Controller-Paradigma*. URL: https://glossar.hs-augsburg.de/Model-View-Controller-Paradigma#CVM-Prozess.2C_Model_2 (besucht am 05. 04. 2018).

[Lau] Rob Lauer. *What is TypeScript?* URL: https://developer.telerik.com/topics/web-development/what-is-typescript/ (besucht am 13. 12. 2017).

[Mica] Microsoft. *Classes*. URL: https://www.typescriptlang.org/docs/handbook/classes.html (besucht am 08. 01. 2018).

[Micb] Microsoft. *Documentation*. URL: https://www.typescriptlang.org/docs/home.html (besucht am 14. 03. 2018).

[Micc] Microsoft. *Facts About Microsoft*. URL: https://news.microsoft.com/facts-about-microsoft/ (besucht am 03. 01. 2018).

[Moza] Mozilla. *Über das Document Object Model*. URL: https://developer.mozilla.org/de/docs/DOM/Ueber_das_Document_Object_Model (besucht am 15. 06. 2018).

[Mozb] Mozilla. *Was ist JavaScript?* URL: https://developer.mozilla.org/de/docs/Web/JavaScript/About_JavaScript (besucht am 13. 12. 2017).

[Mül] David Müllerchen. *Einführung in Angular-CLI*. URL: https://angularjs.de/artikel/angular-cli-einfuehrung/ (besucht am 03. 01. 2018).

[NPM] NPM Inc. *What Is NPM?* URL: https://docs.npmjs.com/getting-started/what-is-npm (besucht am 03. 01. 2018).

[Ope] Open Source Initiative. *The MIT License*. URL: https://opensource.org/licenses/MIT (besucht am 15. 12. 2017).

[Pav] Tero Paviainen. *Refactoring Angular Apps to Component Style*. URL: https://teropa.info/blog/2015/10/18/refactoring-angular-apps-to-components.html (besucht am 13. 04. 2018).

[Pre] Pascal Precht. *Upgrading Angular apps using ngUpgrade*. URL: https://blog.thoughtram.io/angular/2015/10/24/upgrading-apps-to-angular-2-using-ngupgrade.html#upgrading-using-ngupgrade (besucht am 04. 06. 2018).

[Sen] Peleke Sengstacke. *JavaScript Transpilers: What They Are & Why We Need Them*. URL: https://scotch.io/tutorials/javascript-transpilers-what-they-are-why-we-need-them (besucht am 03. 01. 2018).

[Ste] Almero Steyn. *Creating an AngularJS 1.5 component in TypeScript without losing your hair.* URL: http://almerosteyn.com/2016/02/angular15-component-typescript (besucht am 08. 01. 2018).

Quellcodeverzeichnis

Glossar

Document Object Model Das Document Object Model ist eine API für HTML- und XML-Dokumente. Es bildet die strukturelle Repräsentation des Dokumentes und ermöglicht dir, dessen Inhalt und visuelle Darstellung zu verändern. Im Wesentlichen verbindet es Webseiten mit Scripts oder Programmiersprachen. Alle Eigenschaften, Methoden und Events, die dem Webentwickler zum Manipulieren und Erstellen von Webseiten zur Verfügung stehen, sind organisiert in Objekten (z. B. dem Document-Objekt, welches das Dokument selbst repräsentiert, dem Table-Objekt, welches HTML table-Elemente repräsentiert, usw.). Auf solche Objekte kann in modernen Webbrowsern mit Scriptsprachen zugegriffen werden. Das DOM wird meistens in Verbindung mit JavaScript verwendet. Das bedeutet, dass der Code in JavaScript geschrieben ist und das DOM benutzt, um auf die Webseite und dessen Elemente zuzugreifen. [Moza]

Framework Im Software-Engineering ist ein Framework ein modernes Rahmenwerk, das dem Programmierer den Entwicklungsrahmen für seine Anwendungsprogrammierung zur Verfügung stellt und damit die Software-Architektur der Anwendungsprogramme bestimmt. Das Framework wird vorwiegend in der objektorientierten Programmierung eingesetzt und umfasst Bibliotheken und Komponenten wie Laufzeitumgebungen und stellt die Designgrundstruktur für die Entwicklung der Bausteine zur Verfügung. Diese Basisbausteine existieren in Form von abstrakten und konkreten Klassen und unterstützen das Erstellen von Applikationen. Das komplette Framework besteht aus mehreren Klassen, die zusammenarbeiten und wieder verwendbare Entwürfe darstellen. Das Framework spezifiziert den Datenfluss und die Schnittstellen zwischen den Klassen und bildet die generelle, umfangreiche Rahmenstruktur, die in sich unterteilt ist und in die die Programme und Komponenten eingebettet sind. [ITWa]

Google Google LLC ist eine 1998 von Larry Page und Sergey Brin gegründete Firma mit Sitz in den USA, welche durch die gleichnamige

Suchmaschine weltbekannt wurde. Heutzutage spezialisiert sich Google LLC auf Internetdienstleistungen, Internethandel, Werbung und Softwareentwicklung und ist eine der drei wertvollsten Marken der Welt.

Microsoft Microsoft, gegründet in 1975, ist der weltweit führende Anbieter von Geräten und Software für die Heimanwendung. So bietet Microsoft beispielsweise das von Heimanwendern bevorzugte Betriebssystem Windows (derzeit in Version 10), oder die Videospiel-Konsole XBOX an.

MIT Lizenz Die MIT-Lizenz erlaubt den Verwendern der lizenzierten Software und beinhalteter Dokumentation die vollständige Nutzung, Verbreitung, Veröffentlichung etc. dieser. Im Folgenden findet sich ein Auszug aus jener Lizenz:

> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: [...] [Ope]

Objektorientierte Programmierung Bei der objektorientierten Programmierung werden Programme in Einheiten unterteilt, die Objektegenannt werden. Jedes Objekt besitzt einen Zustand, der durch dessen Eigenschaften (Objektattribute) beschrieben wird. Nur die im Objekt selbst vorhandenen Funktionen (Methoden genannt), können dessen Daten manipulieren und so den Zustand verändern. Objekte können anderen Objekten Botschaften senden (indem sie deren Methoden aufrufen) und sie damit auffordern, ihren Zustand zu ändern. Letztendlich bleibt es aber dem Objekt selbst überlassen, ob es der Aufforderung nachkommt. Somit befindet sich das Objekt immer in einem wohldefinierten, selbstkontrollierten Zustand. Man fasst in der OOP-Programmierung also Daten und Funktionen zu Objekten zusammen. Diese Objekte können auf vielfältige Weise miteinander

in Verbindung stehen, indem sie gegenseitig ihre Methoden aufrufen oder ein Objekt andere Objekte enthält. So bilden die Objekte einer Software ein sehr flexibles Gesamtsystem. [ITWb]

Prozedurale Programmierung Bei der prozeduralen Programmierung wird die Gesamtaufgabe, die eine Software lösen soll, in kleinere Teilaufgaben aufgelöst. Jede Teilaufgabe für sich ist einfacher zu beschreiben, programmieren und testen. Außerdem kann der entstehende Programmcode in anderenProgrammen wieder verwendet werden, ein sehr wichtiger Aspekt in der Softwaretechnik. Die bei der Aufgabenlösung entstehenden Programm-Module werden Prozeduren bzw. Funktionen genannt, wobei zu beachten ist das prozedurale Programmierung keineswegs mit funktionaler Programmierung gleichzusetzen ist - beide folgen unterschiedlichen Programmierparadigmen. Prozeduren und Funktionen werden üblicherweise nach Aufgabengebieten gruppiert zu Bibliothekenzusammengefasst, die dann verteilt und in beliebig viele andere Programme eingebunden werden können. [ITWc]

Single-Page-Webanwendung Eine Single-Page-Webanwendung ist eine Webseite, welche möglichst unabhängig von einem Server im Browser des Clients ausgeführt wird und dabei nur aus einer Seite besteht. Ladezeiten beim Anklicken von Links oder Öffnen von Formularen fallen meist kürzer aus, da hierbei nicht die gesamte neue Seite, sondern nur restliche Daten nachgeladen werden müssen.

Transpiler Ein Transpiler (oder auch Source-to-Source Compiler) übersetzt den Quellcode einer Programmiersprache in den Quellcode einer anderen Sprache. Meist wurde die Quelle der Übersetzung dabei nachträglich entwickelt, um die Programmierung in der Zielsprache zu vereinfachen oder Funktionen hinzuzufügen, welche die Zielsprache nicht mitbringt, und die durch einen Transpiler bei der Ausführung abgearbeitet werden können.

Eigenständigkeitserklärung

Ich versichere, dass ich die Arbeit selbständig ohne fremde Hilfe verfasst, in gleicher oder ähnlicher Fassung noch nicht in einem anderen Studiengang als Prüfungsleistung vorgelegt habe und keine anderen als die angegebenen Hilfsmittel und Quellen, einschließlich der angegebenen oder beschriebenen Software, verwendet habe.

Halle (Saale), den 26. Juni 2018