

**Analyse des JavaScript-Frameworks Angular und Konzeption der Migration  
einer Webapplikation**

**Bachelorarbeit**

für die Prüfung zum

Bachelor of Science

des Studiengangs Informatik  
Studienrichtung Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Marc Vornetran

29. August 2016

Bearbeitungszeitraum	12 Wochen
Matrikelnummer	6106705
Kurs	TINF13B4
Ausbildungsfirma, Ort	SMARTCRM GmbH, Kandel
Betreuer der Ausbildungsfirma	Herr B. Sc. Alexander Rupp
Gutachter der Studienakademie	Herr Prof. Dr. Jörn Eisenbiegler

**Erklärung der Eigenleistung**

gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

---

Ort, Datum

Unterschrift

## Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich im Laufe der Durchführung meiner Bachelorarbeit begleitet haben.

Mein besonderer Dank gilt:

- Christian Blinn und Alexander Rupp für die betriebliche Betreuung, sowie die konstruktiven Beiträge für das Gelingen der Arbeit.
- Prof. Dr. Jörn Eisenbiegler für die Betreuung seitens der Dualen Hochschule Karlsruhe, sowie die konstruktiven Vorschläge zur Verbesserung der Arbeit.
- Stephanie Schommers und Michael Vornetran für das ausführliche Korrekturlesen der Arbeit.
- Philipp Schemel für die hilfreichen Vorschläge und die tatkräftige Unterstützung.

Vornetran, Marc

Karlsruhe, den 29. August 2016

## Inhaltsverzeichnis

<b>Inhaltsverzeichnis .....</b>	<b>IV</b>
<b>Abbildungsverzeichnis.....</b>	<b>VII</b>
<b>Tabellenverzeichnis.....</b>	<b>VIII</b>
<b>Listingverzeichnis .....</b>	<b>IX</b>
<b>Abkürzungsverzeichnis .....</b>	<b>XI</b>
<b>1. Einleitung .....</b>	<b>1</b>
1.1 Vorstellung des Unternehmens.....	1
1.2 Ausgangslage.....	1
1.3 Ziel.....	2
1.4 Motivation.....	3
1.5 Abgrenzung.....	3
1.6 Angular .....	4
<b>2. Grundlagen.....</b>	<b>5</b>
2.1 App-Varianten.....	5
2.1.1 Web-App .....	5
2.1.2 Hybride App.....	6
2.1.3 Native App.....	8
2.1.4 Progressive Web App .....	9
2.2 Vergleich der App-Varianten .....	11
2.3 Web-Komponenten .....	12
2.3.1 Templates.....	14
2.3.2 HTML-Imports.....	14
2.3.3 Custom Elements.....	15
2.3.4 Shadow DOM.....	16
<b>3. Stand der Technik.....</b>	<b>18</b>
3.1 Trend der Webentwicklung.....	18
3.2 Angular .....	22
3.2.1 Geschichte.....	23
3.2.2 Grundlagen .....	24
3.2.3 Programmiersprachen.....	25
3.2.4 Vergleich der Programmiersprachen.....	29

3.2.5	Applikationszyklus .....	31
3.2.6	Komponenten.....	34
3.2.7	Templates.....	35
3.2.8	Services.....	43
3.2.9	Dependency Injection .....	45
3.2.10	Formulare .....	47
3.2.11	HTTP.....	51
3.2.12	Routing.....	56
3.2.13	Pipes.....	59
3.3	Alternative Frameworks.....	61
3.3.1	Ember.....	63
3.3.2	React .....	65
3.3.3	Aurelia.....	66
3.3.4	Vue.....	67
3.4	Vergleich der Frameworks.....	68
<b>4.</b>	<b>Migration der Webapplikation .....</b>	<b>71</b>
4.1	Projektanalyse.....	72
4.2	Vorbereitung.....	74
4.2.1	Projektstruktur.....	75
4.2.2	Services.....	76
4.2.3	Komponenten.....	77
4.2.4	Programmiersprache.....	78
4.2.5	Decorators .....	79
4.2.6	Styleguides .....	80
4.3	Strategien.....	81
4.3.1	Big Bang .....	81
4.3.2	Inkrementell .....	82
4.3.3	Neuentwicklung.....	83
4.4	Auswahl der Strategie .....	84
4.5	Durchführung.....	86
4.5.1	Bootstrap.....	86
4.5.2	Setup .....	87
4.5.3	Hybride Ausführung.....	89

4.5.4	Komponenten.....	92
4.5.5	Dependency Injection .....	94
4.5.6	Abhängigkeiten.....	96
4.6	Ergebnis.....	101
<b>5.</b>	<b>Schlussbetrachtung.....</b>	<b>102</b>
5.1	Zusammenfassung .....	102
5.2	Ausblick.....	103
5.3	Reflexion.....	104
	<b>Literaturverzeichnis.....</b>	<b>105</b>
	<b>Bibliografie.....</b>	<b>109</b>
	<b>Anhang .....</b>	<b>110</b>

## Abbildungsverzeichnis

Abbildung 1: Vergleich zwischen Chrome 51 Desktop und Android .....	6
Abbildung 2: Native vs. Hybrid vs. Web.....	7
Abbildung 3: Cross Platform Development mit Native Script .....	8
Abbildung 4: Status der Web-Component-Standardisierung .....	13
Abbildung 5: Shadow-DOM-Technik bei den Progress- und Meter-Elementen ..	16
Abbildung 6: Verbreitung von Programmiersprachen auf GitHub.....	19
Abbildung 7: Google Trend Analyse – jQuery, AngularJS & React.....	20
Abbildung 8: Model-View-ViewModel-Entwurfsmuster .....	21
Abbildung 9: Google-Suchanfragen-Trend nach AngularJS .....	23
Abbildung 10: Google-Trend-Analyse – TypeScript & Dart .....	30
Abbildung 11: Angular-Bindings.....	37
Abbildung 12: Bidirektionale Datenbindung.....	40
Abbildung 13: Anzahl der Frameworks nach Programmiersprache .....	62
Abbildung 14: Ember.js Architektur Übersicht.....	64
Abbildung 15: Aufrufe der AngularJS- und Angular-Seiten.....	71
Abbildung 16: Projektstruktur von SMARTCRM.Web .....	72
Abbildung 17: Ablauf der Migration.....	101

## Tabellenverzeichnis

Tabelle 1: Überblick über die Eigenschaften der App Varianten .....	11
Tabelle 2: Vergleich von TypeScript, JavaScript und Dart für Angular .....	31
Tabelle 3: Property Binding Direktiven in AngularJS und Angular .....	39
Tabelle 4: Event Bindings in AngularJS und Angular.....	39
Tabelle 5: Bidirektionale Datenbindung in AngularJS und Angular .....	40
Tabelle 6: Struktur-Direktiven in AngularJS und Angular .....	41
Tabelle 7: Syntaktischer Zucker bei <i>ngIf</i> und <i>ngFor</i> .....	42
Tabelle 8: CSS-Klassen in einem Angular Formular.....	47
Tabelle 9: Matrix-Klassifikation von Werten.....	55
Tabelle 10: URL-Arten im Browser.....	56
Tabelle 11: Ergebnis der Routen-Definition in Angular.....	59
Tabelle 12: Vergleich von Filter und Pipes.....	60
Tabelle 13: Vergleich der JavaScript-Frameworks.....	70
Tabelle 14: Analyse des Projekts in Bezug auf die Migration.....	74
Tabelle 15: Bewertung der Migrationsstrategien.....	85
Tabelle 16: Abhängigkeiten von Angular .....	87
Tabelle 17: Template Syntax für AngularJS-Bindings.....	94
Tabelle 18: Angular spezifische Abhängigkeiten des Projekts.....	96

## **Listingverzeichnis**

Listing 1: Einbinden der Google-Maps-Bibliothek .....	12
Listing 2: Einbinden von Google Maps als Webkomponente.....	13
Listing 3: „Hello World“-Beispiel für HTML Templates.....	14
Listing 4: Verwendung von HTML Imports .....	15
Listing 5: Registrierung eines Custom Elements .....	16
Listing 6: HTML Aufbau einer App mit Komponenten.....	21
Listing 7: Angular Komponente in TypeScript.....	27
Listing 8: Angular Komponente in JavaScript .....	27
Listing 9: Angular Komponente in Dart.....	29
Listing 10: Config- und Run-Block eines AngularJS Moduls .....	32
Listing 11: <i>OnInit</i> Lifecycle Hook in Angular.....	33
Listing 12: Komponente in AngularJS 1.5 .....	34
Listing 13: Angular Komponente mit Input- und Output-Attributen .....	35
Listing 14: Einfache Interpolation in Angular .....	36
Listing 15: Angular-Interpolation mit Komponente.....	36
Listing 16: Ausführliche Syntax für bidirektionale Datenbindung in Angular .....	41
Listing 17: Service Definition in AngularJS.....	44
Listing 18: Service-Definition in Angular.....	44
Listing 19: Dependency Injection Varianten in AngularJS.....	45
Listing 20: Dependency Injection in Angular .....	46
Listing 21: Formular in AngularJS.....	48
Listing 22: Template Driven Form in Angular .....	49

Listing 23: Model Driven Form in Angular.....	50
Listing 24: \$http Service in AngularJS.....	51
Listing 25: Registrierung der HTTP Service Provider in Angular.....	52
Listing 26: Verwendung des HTTP-Client in Angular.....	53
Listing 27: Observable mit Filterung der Ergebnismenge .....	54
Listing 28: Routing-Konfiguration in AngularJS.....	57
Listing 29: Konfiguration des Angular Component Router.....	58
Listing 30: Hierarchischer Routen-Aufbau in Angular .....	59
Listing 31: Filter-Definition in AngularJS .....	61
Listing 32: Pipe-Definition in Angular .....	61
Listing 33: Component-Hilfsfunktion in AngularJS 1.5 .....	78
Listing 34: Decorators im SMARTCRM.Web-Projekt .....	80
Listing 35: Manueller Bootstrap in AngularJS .....	86
Listing 36: Import der Angular Abhängigkeiten .....	88
Listing 37: Bootstrap durch den Angular Upgrade Adapter .....	89
Listing 38: Downgrade einer Angular-Komponente .....	92
Listing 39: Upgrade einer AngularJS-Direktive.....	93
Listing 40: Upgrades eines AngularJS-Service .....	95
Listing 41: Downgrade eines Angular-Service .....	96
Listing 42: Migration der Routing Definition.....	100
Listing 43: Routing Definition der Webapplikation.....	110
Listing 44: Bootstrap-Prozess einer Angular-Anwendung.....	111

## Abkürzungsverzeichnis

.NET	Microsoft .NET – Plattform für Anwendungsentwicklung
API	Application Programming Interface – Schnittstelle zur Kommunikation zwischen Programmen
ARIA	Accessible Rich Internet Applications – Initiative für Barrierefreiheit dynamischer Webanwendungen und HTML-Seiten im Allgemeinen
COM	Component Object Model – Microsoft-Technologie zur Kommunikation zwischen Prozessen unabhängig von den eingesetzten Programmiersprachen
CRM	Customer Relationship Management – Strategischer Ansatz zur vollständigen Erfassung, Planung und Steuerung des Kundenlebenszyklus
CSS	Cascading Style Sheets – Design und Gestaltungsanweisungen für Web-Dokumente in HTML
CTI	Computer Telephony Integration
DI	Dependency Injection – Entwurfsmuster, nach dem die Abhängigkeiten der Klassen an einer zentralen Stelle verwaltet werden. Die Klassen selbst kümmern sich nicht um die Erzeugung ihrer Abhängigkeiten und bekommen sie „ <i>injiziert</i> “
DMS	Dokumentenmanagementsystem
DOM	Document Object Model – Eine Baumstruktur zur Repräsentation von HTML-, XML- oder XHTML-Dokumenten
DSL	Domain-specific language – Formale Sprache spezifisch für eine Problemdomäne
ERP	Enterprise Resource Planning

<b>ES</b>	ECMAScript – Standardisierte Spezifikation von JavaScript
<b>HTML</b>	Hypertext Markup Language – Dient der Strukturierung von Dokumenten im Web
<b>HTTP</b>	Hypertext Transfer Protocol – Zustandsloses Protokoll zur Übertragung von Daten im Web
<b>HTTPS</b>	Hypertext Transfer Protocol Secure – Kommunikationsprotokoll zur abhörsicheren Übertragung von Daten im Web. Zusätzlich wird die Vertraulichkeit und Integrität gesichert
<b>IDE</b>	Integrated development environment – Eine umfassende Entwicklungsumgebung zur Softwareentwicklung
<b>IIFE</b>	Immediately Invoked Function Expression – Eine JavaScript-Funktion, welche direkt nach ihrer Definition ausgeführt wird. Als Design Pattern wird sie zum Schutz des globalen Scopes eingesetzt
<b>JSON</b>	JavaScript Object Notation – Für Menschen lesbare Format zum Austausch von Daten
<b>JSONP</b>	JSON with Padding – Methode um die Same-Origin-Policy in Browsern zu umgehen
<b>JSX</b>	JavaScript Syntax Extension – Optionale Syntax-Erweiterung für React, angelehnt an XML
<b>MVC</b>	Model View Controller – Entwurfsmuster zur Erhöhung der Wiederverwendbarkeit von Komponenten
<b>MVVM</b>	Model-View-ViewModel – Variante des MVC-Entwurfsmusters zur besseren Trennung der Geschäftslogik von der Darstellung
<b>npm</b>	Node Package Manager – Package Manager für Node.js
<b>OS</b>	Operating System – Betriebssystem, Verwaltet die Systemressourcen eines Computers und stellt Programme bereit

<b>PIM</b>	Persönlicher Informationsmanager – SMARTCRM-Modul zur Termin- und Aufgabenplanung
<b>RC</b>	Release Candidate – Meilenstein während der Softwareentwicklung vor der Veröffentlichung
<b>RxJS</b>	Reactive Extensions – Bibliothek für asynchrone Programmierung mit Observables
<b>Sass</b>	Syntactically Awesome Stylesheets – Syntaxerweiterung für CSS zur leichteren Erstellung von Stylesheets
<b>SEO</b>	Search engine optimization – Methoden zur Verbesserung der Sichtbarkeit einer Website innerhalb von Suchmaschinen-Ergebnissen
<b>SPA</b>	Single-page application – Die Inhalte der Webanwendung werden dynamisch nachgeladen. Durch den Verzicht auf Browser-Reloads wird eine flüssige Bedienung der Anwendung ermöglicht
<b>SRP</b>	Single Responsibility Principle – Entwurfsrichtlinie, nach der jede Klasse nur eine einzige Aufgabe besitzen sollte
<b>UI</b>	User Interface – Schnittstelle für Mensch-Maschine-Interaktion
<b>UX</b>	User Experience – Alle Aspekte, die zur Nutzererfahrung beitragen
<b>VM</b>	Virtual Machine – Emulation eines speziellen Computer-Systems
<b>XHR</b>	XMLHttpRequest – Objekt, um HTTP-Anfragen aus JavaScript zu versenden
<b>XML</b>	Extensible Markup Language – Sprache zur Strukturierung hierarchischer Daten

## 1. Einleitung

Diese Arbeit befasst sich mit der neu entwickelten Version des JavaScript-Frameworks Angular. Dabei werden Unterschiede zum Vorgänger herausgearbeitet, sowie potenzielle Vorteile für das vorliegende Projekt untersucht. Im zweiten Teil wird die Migration der Webapplikation konzipiert. Diese umfasst Strategien für die Vorbereitung und Umsetzung der Umstellung.

### 1.1 Vorstellung des Unternehmens

Die SMARTCRM GmbH mit Sitz in Kandel entwickelt seit 1992 CRM-Lösungen für mittelständische Betriebe aus Industrie und Großhandel. Durch den modularen Aufbau der gleichnamigen Software lassen sich Kundenprojekte frei konfigurieren und individuell umsetzen. Das System kann vollständig an das entsprechende Einsatzszenario angepasst werden. Ergänzt wird das Produkt durch Analyse, Beratung und Schulungsangebote. Unter Verwendung diverser Schnittstellen ist es möglich, die CRM-Software mit Drittssystemen zu verbinden, um Daten aus verschiedenen Quellen zu bündeln. Hierzu zählen beispielsweise ERP- und DMS-Systeme, CTI für ein verbessertes Informationsmanagement bei Telefonaten oder eine Integration von Microsoft-Office-Produkten.<sup>1</sup>

### 1.2 Ausgangslage

Implementiert wurde die CRM-Lösung SMARTCRM hauptsächlich in der Programmiersprache Visual FoxPro von Microsoft. Einige der neuen Komponenten wurden in C# unter Verwendung des .NET-Frameworks umgesetzt. Hierzu zählt auch das Modul „Persönlicher Informationsmanager“ (kurz PIM), das der Termin- und Aufgabenplanung, sowie dem schnellen internen Kommunikationsfluss dient. Unter Verwendung von COM-Schnittstellen können diese Komponenten mit dem Kern der Software kommunizieren.

Durch die verwendeten Technologien entsteht eine Bindung an eine Windows-Umgebung, um die Software ausführen und verwenden zu können. Um einen flexibleren und mobilen Einsatz zu gewährleisten, wurde eine iOS-App entwickelt. Hiermit haben Kunden über iPhone oder iPad Zugriff auf die wichtigen CRM-Funktionalitäten.

---

<sup>1</sup> Vgl. SMARTCRM GmbH (2016)

Um einen größeren Markt zu erreichen, folgt der Schritt zur hybriden Webanwendung. Damit verfolgt das Unternehmen dem aktuellen Trend, von plattformgebundenen Applikationen abzusehen und den Fokus auf einheitliche Anwendungen für alle Plattformen zu legen.<sup>2</sup> Der immer wiederkehrende Kundenwunsch, die Software über einen Browser oder unter Android verwenden zu können, sowie die positive Resonanz auf die iOS-App sind weitere treibende Faktoren in Richtung Webentwicklung.

In Kooperation mit einer Webagentur wurde die neue Web-App entwickelt und an die SMARTCRM GmbH übergeben. Das Projekt setzt dabei auf das Framework Ionic zur Entwicklung von hybriden, mobilen Apps.<sup>3</sup> Zur Umsetzung können die Webtechnologien HTML, CSS und JavaScript verwendet werden.

Das clientseitige JavaScript Framework AngularJS ist ein Open-Source-Projekt, welches von Google entwickelt wird. Innerhalb von Ionic-Projekten wird Angular für die Architektur der Anwendung eingesetzt. Diese befolgt das „Model View Controller (MVC)“-Entwurfsmuster.

Bereits im Dezember 2015 wurde die öffentliche Beta-Phase von Angular angekündigt.<sup>4</sup> Die neue Version wurde von Grund auf neu entworfen und bildet damit einen harten Schnitt zum Vorgänger. Dabei soll der neue Entwurf Performance-Probleme beheben und Schwachpunkte an AngularJS beseitigen.

### 1.3 Ziel

Diese Arbeit soll eine detaillierte Analyse des JavaScript-Frameworks Angular bieten. Miteinbezogen werden dabei aktuelle Standards der Webentwicklung. Darüber hinaus soll ein Vergleich zu anderen Frameworks erfolgen, um Angular für den vorliegenden Einsatzzweck einzurordnen. Innerhalb der Untersuchung werden Unterschiede zwischen den verschiedenen Framework Versionen hervorgehoben. Dadurch soll ein erster Aufschluss darüber gegeben werden, welche Änderungen für das Webprojekt relevant sind.

Zusätzlich wird eine Strategie zur Migration der Anwendung erarbeitet. Diese enthält klare

---

<sup>2</sup> Vgl. Mobile Zeitgeist (2015)

<sup>3</sup> Siehe Kapitel 2.1.2

<sup>4</sup> Vgl. Green (2015)

Empfehlungen darüber, welche Aspekte zu beachten und welche Vorbereitungen zu treffen sind. Auf Basis des aktuellen Status von Angular soll dadurch eine fundierte Grundlage für eine potenzielle Umstellung in der Zukunft geschaffen werden.

Mit Abschluss der Arbeit sollen daher eine Anleitung, sowie Handlungsempfehlungen für die Migration der Webapplikation entstehen.

### 1.4 Motivation

Die neue Webanwendung von SMARTCRM basiert auf AngularJS, der ursprünglichen Version des Frameworks. Für das Unternehmen ist es wichtig, die Applikation zukunftsfähig und performant zu halten. Daher gilt es, die aktuellen Trends der Webentwicklung im Auge zu halten und Framework Upgrades hinsichtlich ihres Mehrwerts zu untersuchen. Um potenzielle Vorteile zu erkennen, ist eine Untersuchung des neuen Frameworks hilfreich.

Eine Umstellung auf die neue Version des Frameworks wird ab einem gewissen Punkt unausweichlich sein, selbst wenn dieser aktuell mehrere Jahre in der Zukunft liegen sollte. Deshalb sollten das Sammeln von Ressourcen, sowie die Umsetzung vorbereitender Maßnahmen möglichst früh beginnen. Dadurch lässt sich die Migration effizienter durchführen und auf diesem Weg Zeit sparen.

### 1.5 Abgrenzung

Diese Arbeit behandelt thematisch zwei große Bausteine. Als erstes wird das JavaScript-Framework Angular untersucht, unter Miteinbeziehung der modernen Webentwicklung. Zusätzlich werden weitere Frameworks für einen Vergleich herangezogen, um einen Kontext für die diversen Einsatzzwecke der Projekte aufzuzeigen. Im nächsten Teil der Arbeit wird eine Grundlage für die Migration des Webprojekts geschaffen. Diese enthält eine auf das vorliegende Projekt bezogene Strategie zur effizienten Umstellung auf die neue Version von Angular. In separaten Entwicklungs-Rewpositories können praktische Schritte der Migration nachvollzogen werden. Allerdings kann eine vollständige Umstellung des Projekts an dieser Stelle noch nicht realisiert werden. Ein Grund dafür ist die ausstehende Veröffentlichung von Angular. Das Framework befindet sich in der aktiven Entwicklung und erfährt rapide Änderungen. Dadurch ist die Dokumentation noch nicht vollständig und viele Drittanbieter können ihre eigenen

Produkte noch nicht vollends anpassen. Des Weiteren benötigt das Webprojekt aufgrund seiner Komplexität einen hohen Zeitaufwand und viele Ressourcen zur erfolgreichen Umstellung.

### 1.6 Angular

Im Verlauf dieser Arbeit bezeichnet der Begriff *AngularJS* die erste Version des Frameworks, *Angular* den Nachfolger.

Diese Arbeit bezieht sich auf Release Candidate 4 von Angular, welches am 30. Juni 2016 veröffentlicht wurde.<sup>5</sup>

---

<sup>5</sup> Abgerufen am 12. August 2016 von <http://angularjs.blogspot.de/2016/06/rc4-now-available.html>

## 2. Grundlagen

Für das Verständnis dieser Arbeit sind fortgeschrittene Kenntnisse im Bereich der Webentwicklung notwendig. Insbesondere neue Technologien in HTML und JavaScript spielen eine wichtige Rolle bei der Realisierung von Apps, sowie innerhalb der untersuchten Frameworks.

Spezielle Begriffe, Technologien, Bibliotheken und Frameworks, die im Verlauf dieser Arbeit genannt werden oder zur besseren Verständlichkeit beitragen, werden im Folgenden erklärt.

### 2.1 App-Varianten

Heutzutage sind Apps der Grundstein für mobile Endgeräte und werden über entsprechende App Stores vertrieben. Auch für Desktop-Systeme ist ihre Bedeutung immer größer geworden.

In klassischer Hinsicht bezeichnet der Begriff App eine Applikation für Smartphones oder Tablets. Diese wird über den entsprechenden App Store heruntergeladen und installiert.

Mittlerweile lässt sich der Begriff jedoch weiter differenzieren. Durch die stetige Weiterentwicklung der Browser ist es möglich, das Verhalten von Apps auch im Web-Kontext abzubilden. Um eine möglichst native Benutzererfahrung zu ermöglichen, wurden verschiedene Konzepte und Techniken entwickelt, die im Folgenden kurz erläutert werden.

#### 2.1.1 Web-App

Bei dieser Art von Anwendungen befinden sich der Code und die zugehörigen Ressourcen auf einem Server. Aufgerufen und ausgeführt wird die App mithilfe eines beliebigen Browser. Hier liegt der größte Unterschied zu traditionellen Apps, welche nur einmal heruntergeladen werden und anschließend dauerhaft auf dem Gerät verfügbar sind.

Die Anwendung kann aufgrund der Browser-Laufzeitumgebung auf allen Desktop- und mobilen Betriebssystemen verwendet werden. Die Entwicklung ist nur einmal notwendig und erreicht dabei alle Plattformen mit Internet-Browsern. Allerdings gilt es, browserspezifische Unterschiede zu beachten. Gerade mobile Browser verfügen oft nicht über die gleiche Funktionalität wie ihr Desktop-Pendant. Bedingt ist diese Tatsache schon durch die grundlegenden Unterschiede der Hardware.

Zusätzlich spielt die Performance für viele Apps eine bedeutende Rolle. Mobile Geräte können vor allem aufgrund der Akkulaufzeit nicht die gleiche Leistung wie Desktop-Systeme erbringen.

Abbildung 1: Vergleich zwischen Chrome 51 Desktop und Android

Mixed support	Chrome 51	Chrome 51 for Android
CSS3 Cursors (original values)	Yes	No
CSS3 Cursors: zoom-in & zoom-out	Yes	No
KeyboardEvent.code	Yes	No
Custom protocol handling	Yes	No
Drag and Drop	Yes	No
PointerLock API	Yes	No
Web Notifications	Yes	No
Shared Web Workers	Yes	No
FIDO U2F API	Yes	No
Opus	Yes	No
Ogg/Theora video format	Yes	No
Network Information API	No	Yes
CSS font-smooth	Partial	-webkit-

Quelle: Can I Use – Browser Comparison<sup>6</sup>

Ein weiterer Unterschied zu den anderen App-Varianten ist die Abhängigkeit von einer dauerhaften Internetverbindung. Für das Starten einer Web-App ist eine Verbindung zum Server notwendig. Neue Technologien in HTML versuchen diese Barriere durch die Nutzung des Application Cache oder den Einsatz von Service Workern zu beseitigen. In beiden Fällen werden die Ressourcen der Web-App nach dem ersten Aufruf im lokalen Cache des Browsers bereitgehalten. Alle folgenden Anfragen werden direkt aus dem Cache bedient.

Durch das Sandbox-Verfahren von Browsern ist es JavaScript-Anwendungen nicht möglich, native Funktionen des Betriebssystems aufzurufen. Stattdessen stellt der Browser eigene Schnittstellen für GPS, Kamera, Dateisystem oder Tonwiedergabe zur Verfügung. Die Unterstützung verschiedener Browser-Versionen variiert an dieser Stelle.<sup>7</sup>

### 2.1.2 Hybride App

Hybride Apps bündeln die Vorteile nativer Apps und Web-Apps. Die Entwicklung findet weiterhin im Web-Kontext unter Verwendung von HTML, CSS und JavaScript statt. Allerdings

<sup>6</sup> Abgerufen am 12. Juli 2016 von <http://caniuse.com/#compare=chrome+51,android+51>

<sup>7</sup> Vgl. Riepe (2015)

wird der Browser nicht mehr zur Ausführung und Anzeige der Anwendung eingesetzt. Stattdessen wird eine Web View in einer nativen App verpackt. Anschließend kann die hybride App über die jeweiligen Apps Stores bezogen werden. Für den Benutzer ist der Unterschied zu einer nativen Anwendung in den meisten Fällen nicht erkennbar.<sup>8</sup>

Durch die lokale Installation der App wird die Ausführung ohne Internetverbindung möglich. Alle benötigten Ressourcen befinden sich bereits auf dem Gerät und erlauben einen schnellen Start der App. Darüber hinaus ist ein Zugriff auf Hardware-Funktionen möglich. Diese werden durch das verwendete hybride Framework als JavaScript API zur Verfügung gestellt.

Abbildung 2: Native vs. Hybrid vs. Web



Quelle: Webscope<sup>9</sup>

Einer der größten Vorteile ist eine zentrale Codebasis für alle Plattformen. Für die unterschiedlichen Systeme wird derselbe Code jeweils in einem nativen Container eingebettet. So sinken die Entwicklungskosten mit jeder zusätzlich berücksichtigten Plattform.<sup>10</sup>

Der zentrale Nachteil hybrider Apps ist die Performance im Vergleich zu nativen Apps. Aufgrund der Webansicht sowie der Leistung heutiger Geräte ist eine hybride App langsamer als eine native Anwendung. Das Rendering des Document Object Model (DOM), der zugrundeliegenden Technik von HTML-Dokumenten, erfordert eine höhere Rechenleistung und beeinträchtigt zugleich die Akkulaufzeit mobiler Geräte.

<sup>8</sup> Vgl. Svanidze (2014)

<sup>9</sup> Abgerufen am 13. Juli 2016 von <https://webscope.co.nz/hybrid-apps-could-your-business-be-in-the-app-store-sooner-than-you-think>

<sup>10</sup> Vgl. Svanidze (2014)

### 2.1.3 Native App

Eine nativ entwickelte Applikation ist in der Programmiersprache ihrer Plattform geschrieben. Im Fall von iOS ist dies Objective-C oder die neuere Sprache Swift, bei Android-Applikationen hingegen Java. Dadurch können die Programme auf systemeigene Komponenten für das User Interface (UI) zugreifen.<sup>11</sup> Aufgrund der technischen Umsetzung sind native Apps an ihre Plattform gebunden und müssen für jedes mobile Betriebssystem neu entwickelt werden.

Der Vorteil nativer Apps liegt in der optimalen Performance durch die Systemnähe. Außerdem hat die Anwendung vollständigen Zugriff auf das Application Programming Interface (API) des Betriebssystems. Hardware-Funktionen wie GPS oder die Kamera können dadurch benutzt werden. Die Offline-Verfügbarkeit ist aufgrund der Installation über den App Store gewährleistet.

Mithilfe neuer Technologien wie Native Script<sup>12</sup> oder React Native<sup>13</sup> versucht man, die plattformunabhängige Entwicklung zu vereinfachen. Hierbei unterscheiden sich die Projekte in ihrer Herangehensweise, native UI-Komponenten anzusprechen.

Abbildung 3: Cross Platform Development mit Native Script



Quelle: Native Script - About<sup>14</sup>

Native Script folgt dabei dem Ansatz, eine einzige Codebasis in JavaScript für alle Plattformen zu ermöglichen. Hierfür bietet das Framework durch JavaScript-Aufrufe direkten Zugriff auf native APIs. Diese Funktionsaufrufe werden durch die Native Script Runtime in beide Richtungen übersetzt und koordiniert.

<sup>11</sup> Vgl. Abed (2015)

<sup>12</sup> Siehe <https://www.nativescript.org>

<sup>13</sup> Siehe <https://facebook.github.io/react-native/>

<sup>14</sup> Abgerufen am 13. Juli 2016 von <https://www.nativescript.org/about>

Im Gegensatz hierzu erfordert React Native plattformspezifische Kenntnisse zur UI-Gestaltung. Für iOS müssen die Komponenten in Objective-C/Swift angesprochen werden, während auf Android eine Java-Implementierung nötig ist. Lediglich die Geschäftslogik kann extrahiert und in JavaScript umgesetzt werden.

Beide Frameworks erlauben die Entwicklung nativer Apps unter Verwendung bereits bekannter Webtechnologien. Dabei ist es möglich, auf Webview-Container zu verzichten. Durch den Einsatz nativer Komponenten wird die User Experience (UX) verbessert. Der Benutzer benötigt keine über seinen Wissensstand hinausgehenden Kenntnisse, kann Betriebssystem-eigene Gesten wiederverwenden oder erkennt spezifische Bedienelemente wieder.

### 2.1.4 Progressive Web App

Mit der neusten Herangehensweise zur Entwicklung von Apps wird versucht, die Vorteile von Web Apps und nativen Apps zu kombinieren. Dabei werden moderne Web-Technologien genutzt, um eine App-ähnliche Erfahrung im Browser zu schaffen. Aufgrund ihrer Eigenschaften gilt es, Progressive Web Apps von herkömmlichen Web Apps zu differenzieren.<sup>15</sup>

#### Progressive

Jeder Benutzer kann die Anwendung ausführen und benutzen. Die Wahl des Browsers sollte dabei keine Rolle spielen. Bei nicht unterstützten Features muss die App auf Polyfills oder Fallbacks zugreifen. Polyfills ergänzen fehlende Funktionen durch JavaScript und machen diese für ältere Browser verfügbar.<sup>16</sup>

#### Responsive

Das Layout der Anwendung passt sich automatisch der Größe des Bildschirms an. Dadurch wird die Bedienung auf Smartphones, Tablets und Desktop-Geräten gewährleistet.

#### Netzwerkunabhängigkeit

Durch den Einsatz der Service Worker Technologie in aktuellen Browsern können die Ressourcen der App lokal im Cache gespeichert werden. Bei fehlenden, unstabilen oder

---

<sup>15</sup> Vgl. LePage (2016)

<sup>16</sup> Vgl. <https://developer.mozilla.org/de/docs/Glossary/Polyfill> (18. Juli 2016)

langsamen Netzwerkverbindungen kann ein Großteil der Anfragen direkt aus dem Cache verarbeitet werden.

### **Appähnliche UX**

Bekannte Interaktions- und Navigationselemente aus dem App-Umfeld werden verwendet. Der Benutzer benötigt keine Kenntnisse, die über seinen Wissensstand hinausgehen, und kann die Applikation intuitiv bedienen.

### **Aktualität**

Der Service Worker Ansatz erlaubt die Aktualisierung der Progressive Web App im Hintergrund. So befindet sich die Anwendung jederzeit auf dem neusten Stand.

### **Sicherheit**

Zur abhörsicheren Übertragung im Web wird das Hypertext Transfer Protocol Secure (HTTPS) eingesetzt. Darüber hinaus wird mit HTTPS auch die Integrität der Daten gewährleistet, um Man-in-the-Middle-Attacken zu verhindern. Diese Vorteile sind im Grunde unabdingbar für jede Anwendung. Außerdem schreibt die Spezifikation der Service Worker eine Verwendung von HTTPS vor.<sup>17</sup>

### **Search Engine Optimization (SEO)**

Die Auffindbarkeit der Progressive Web App wird für Suchmaschinen und soziale Netzwerke optimiert. Dabei lassen sich die Inhalte und die Struktur der Anwendung indexieren.

### **Nutzerbindung**

Push-Benachrichtigungen erreichen den Benutzer auch bei geschlossenem Browser. Hierdurch wird ein schneller Wiedereinstieg im richtigen Kontext ermöglicht.

### **Installierbarkeit**

Im Gegensatz zu klassischen Apps ist eine Installation über einen App Store nicht notwendig. Der Browser speichert alle Ressourcen der App im Cache und hält sie für die folgenden Aufrufe bereit. Darüber hinaus kann der Benutzer dazu angeleitet werden, die Progressive Web App als Verknüpfung auf dem Desktop oder Home-Bildschirm abzulegen.

---

<sup>17</sup> Vgl. Russell, Song, & Archibald (2015)

## Referenzierbarkeit

Sofern die App die Anzeige der Adressleiste des Browsers zulässt, kann der aktuelle Zustand der Anwendung geteilt werden. Die URL kann einfach kopiert und an Andere weitergegeben werden.

## 2.2 Vergleich der App-Varianten

In der folgenden Tabelle wird eine Zusammenfassung über die vorangegangenen App-Varianten geboten. Dabei werden die beschriebenen App-Varianten anhand definierter Kriterien miteinander verglichen. Diese Übersicht dient der verschiedenen Herangehensweisen. Stattdessen sollen die unterschiedlichen Eigenschaften hervorgehoben werden, nicht aber deren Bewertung.

Tabelle 1: Überblick über die Eigenschaften der App Varianten

	 Web App	 Hybrid App	 Native App	 Progressive Web App
<b>Performance</b>	(-)	(+)	(++)	(+)
<b>Offline-Nutzbarkeit</b>	(-)	(+)	(++)	(+)
<b>Native Funktionen<sup>1</sup></b>	(-)	(+)	(+)	(-)
<b>Installation</b>	(+)	(-)	(-)	(+)
<b>Erreichbarkeit</b>	(+)	(-)	(-)	(+)
<b>Entwicklungskosten</b>	(+)	(-)	(--)	(-)
<b>Plattformunabhängigkeit</b>	(++)	(+)	(--)	(++)
<b>Updates</b>	(+)	(-)	(-)	(+)

<sup>1</sup> Bezieht sich auf native Funktionen eines Endgeräts, welche nicht durch den Browser zur Verfügung gestellt werden.

Quelle: Eigene Darstellung, Vgl. Svanidze (2014)

### 2.3 Web-Komponenten

Mit der Standardisierung von Web-Komponenten soll ein einfacher Weg geschaffen werden, neue Funktionalitäten auf einer Website einzubinden. Web-Komponenten sind in sich selbst gekapselt und können leicht wiederverwendet werden.

Aktuell gibt es keinen einheitlichen Weg, um Bibliotheken zu einer Seite hinzuzufügen. In manchen Fällen reicht es aus, ein externes Skript einzubinden. Allerdings sind oft Stylesheets oder weitere Ressourcen notwendig. Bei der Verwendung verschiedener Bibliotheken und Frameworks spielt zusätzlich die Reihenfolge der JavaScript-Dateien eine Rolle.<sup>18</sup>

In Listing 1 ist der Code zur Einbindung einer Google-Maps-Karte abgebildet. Für die korrekte Einbindung muss zuerst ein Platzhalter-Element mit einer eindeutigen ID versehen werden. An dieser Stelle wird später der Inhalt der Komponente eingefügt. Des Weiteren wird die externe Google-Maps-Bibliothek geladen. Der letzte Schritt ist die Initialisierung der Karte durch einen JavaScript-Aufruf der Google-Maps-API.

Listing 1: Einbinden der Google-Maps-Bibliothek

```
1  <div id="map"></div>
2  <script src="http://maps.googleapis.com/maps/api/js?key=APIKEY "></script>
3  <script>
4      new google.maps.Map(document.getElementById('map'), {
5          center: new google.maps.LatLng(49.026, 8.385),
6          zoom: 8,
7          mapTypeId: google.maps.MapTypeId.ROADMAP
8      });
9  </script>
```

Quelle: Eigene Darstellung

Die Vorgehensweise ist hierbei spezifisch für die eingesetzte Bibliothek. Andere Libraries und Frameworks erfordern unterschiedliche Anordnungen der Ressourcen, und es kann zu Inkompabilität zwischen den Komponenten kommen.

---

<sup>18</sup> Vgl. Kröner (2014)

Ein standardisiertes Plugin-System wird durch Web-Komponenten bereitgestellt.

Listing 2: Einbinden von Google Maps als Webkomponente

```
1 <link rel="import" href="google-maps-plugin.html">
2 <google-map latitude="49.026" longitude="8.385" zoom="8" type="roadmap" />
```

Quelle: Eigene Darstellung

Die Komponente wird zu Beginn als externes HTML-Element importiert. Alle benötigten HTML-Templates, CSS-Styles und JavaScript-Dateien sind innerhalb des externen Elements zusammengefasst. Anschließend lässt sich die neue Komponente wie ein gewöhnlicher HTML-Tag einsetzen und über Attribute konfigurieren. Darüber hinaus können Web-Komponenten eigene JavaScript-APIs oder DOM-Events exportieren.

Es gilt zu beachten, dass Web-Komponenten aus vier voneinander unabhängigen Standards definiert werden. Aufgrund des jungen Alters dieser Technologien befinden sich die Spezifikationen noch im Zustand des Working Drafts, mit Ausnahme der als Living Standard definierten HTML-Templates.<sup>19</sup> In allen Fällen wird aktiv über die Spezifikation diskutiert, wodurch Veränderungen relativ leicht vorgenommen werden können.

Abbildung 4: Status der Web-Component-Standardisierung

	Specged	Implementation				
		Polyfill	Chrome / Opera	Firefox	Safari	Edge
Templates			Stable	Stable	8	13
HTML Imports			Stable	On Hold	No Active Development	Vote
Custom Elements			Stable	Flag	Prototype	Vote
Shadow DOM			Stable	Flag	10	Vote

Quelle: Are We Componentized Yet?<sup>20</sup>

Für einen produktiven Einsatz von Web-Komponenten sind Polyfills notwendig. Selbst in den

---

<sup>19</sup> Vgl. Rimmer (2015)

<sup>20</sup> Rimmer (2015)

sogenannten Evergreen-Browsern, zu denen alle Browser mit automatischen Hintergrund-Updates zählen, ist die Implementierung noch nicht abgeschlossen. Lediglich Browser auf Basis des Open-Source-Chromium-Projekts und der Blink Rendering Engine unterstützen die Standards in ihrer stabilen Version. Zu diesen Browsern gehören Google Chrome und Opera.

### 2.3.1 Templates

Diese Spezifikation umfasst die Erweiterung von HTML durch einen Template Tag. Im Gegensatz zu anderen Elementen wird der Inhalt dieses Tags beim Aufruf der Seite nicht angezeigt. Das Element dient als Fragment zur Wiederverwendung in JavaScript.<sup>21</sup>

Listing 3: „Hello World“-Beispiel für HTML Templates

```
1 <div id="container"></div>
2 <template id="content">Hello World!</template>
3 <script>
4   const template = document.querySelector('#content');
5   const container = document.querySelector('#container');
6   const templateClone = document.importNode(template.content, true);
7   container.appendChild(templateClone);
8 </script>
```

Quelle: Eigene Darstellung

In Listing 3 soll der Inhalt des Templates zur Anreicherung des Container-Elements genutzt werden. Der Text des Templates wird über die Funktion `document.importNode()` geklont und anschließend an den Inhalt des Container-Elements angehängt.

### 2.3.2 HTML-Imports

Während es für alle Arten von Web-Ressourcen einfache Möglichkeiten zur Einbindung auf eine Seite gibt, fehlt diese Funktionalität für HTML selbst.

Durch den neuen HTML-Import-Standard wird diese Situation verbessert. Über einen Import lassen sich beliebige HTML-Dateien nachladen. Innerhalb der importierten Ressource können sich beliebige Inhalte eines regulären HTML-Dokumentes befinden.

---

<sup>21</sup> Vgl. Mozilla Developer Network (2015)

Um den Browser über einen Import zu informieren, wird wie bei Stylesheets der Link-Tag verwendet. Die Anweisung befindet sich üblicherweise im Head-Bereich der Seite.

Listing 4: Verwendung von HTML Imports

```
1  <head>
2      <link rel="import" href="/path/to/imports/stuff.html">
3      <link rel="import" href="http://example.com/elements.html">
4  </head>
```

Quelle: Eigene Darstellung

Ein wichtiger Anwendungsfall für HTML-Imports über Web-Komponenten hinaus, ist die Zusammenfassung mehrerer Ressourcen zu einer Datei. Besonders Bibliotheken mit diversen Stylesheets, Scripts, Templates und weiteren Abhängigkeiten können von diesem Standard profitieren. Um alle benötigten Ressourcen zu importieren genügt das einfache Einfügen einer einzigen Zeile.<sup>22</sup>

### 2.3.3 Custom Elements

Bisher litten HTML-Seiten unter einem eingeschränkten semantischen Ausdruck. Denn die normalen Tags drücken nur begrenzt ihren Sinn und Zweck aus. Mit HTML5 wurde eine Reihe neuer Elemente eingeführt, um die Semantik einer Seite zu verbessern. Hierzu gehören beispielsweise: `<header>`, `<nav>`, `<section>`, `<article>`, `<aside>` und `<footer>`. Diese Elemente verdeutlichen durch ihren Namen, wofür sie eingesetzt werden.

Allerdings ist es für Entwickler nicht möglich, eigene HTML-Tags zu definieren. Um dies zu ändern, bietet die „Custom Elements“-Spezifikation neue Schnittstellen und wird damit zur Grundlage für die gesamte Web-Komponenten-Funktionalität. Es ergeben sich folgende neue Funktionen:<sup>23</sup>

- Registrierung neuer HTML-Elemente
- Vererbung zwischen Elementen
- Zusammenfassung von Funktionalitäten zu einem Element
- Erweiterung der bestehenden API eines Elementes

---

<sup>22</sup> Vgl. Bidelman, HTML Imports - Include for the web (2013)

<sup>23</sup> Vgl. Bidelman, Custom Elements - Defining new elements in HTML (2013)

Über die Funktion `document.registerElement()` lässt sich ein neues Element registrieren. Der Name muss einen Bindestrich enthalten, um die Kompatibilität mit kommenden HTML-Standards zu sichern.

Listing 5: Registrierung eines Custom Elements

```

1  const customButton = document.registerElement('custom-button', {
2      prototype: Object.create(HTMLElement.prototype),
3      extends: 'button'
4  });
5  document.body.appendChild(new customButton());

```

Quelle: Eigene Darstellung

Über den zweiten Parameter lassen sich verschiedene Optionen zum neuen Element angeben. Sofern das Objekt nicht übergeben wurde, fällt der Browser auf Standardwerte zurück.

### 2.3.4 Shadow DOM

Dieser Standard sorgt für die Kapselung von Teilen des DOMs innerhalb eines sogenannten Shadow Roots. Ohne diese Kapselung können Inhalte einer Web-Komponente unabsichtlich durch globale Stylesheets oder JavaScript-Manipulationen beeinflusst werden. Durch den Aufruf der Funktion `createShadowRoot()` an einem Element wird dieses zu einem Shadow Host. Die Inhalte des Shadow Roots sind im DOM der Seite nicht sichtbar und dadurch losgelöst.

Abbildung 5: Shadow-DOM-Technik bei den Progress- und Meter-Elementen

The screenshot shows the element tree of an HTML document. It highlights two specific elements: a `<progress>` element and a `<meter>` element. Both of these elements have a child node with the text '#shadow-root (user-agent) == \$0'. Expanding this reveals the internal structure of the Shadow Root. The `<progress>` element's shadow root contains three nested div elements with pseudo-classes: `pseudo="-webkit-progress-inner-element"`, `pseudo="-webkit-progress-bar"`, and `pseudo="-webkit-progress-value"`. The `<div>` under `-webkit-progress-value` has a style attribute setting its width to -100%. The `<meter>` element's shadow root also contains three nested div elements with pseudo-classes: `pseudo="-webkit-meter-inner-element"`, `pseudo="-webkit-meter-bar"`, and `pseudo="-webkit-meter-optimum-value"`. The `<div>` under `-webkit-meter-optimum-value` has a style attribute setting its width to 40%.

```

<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    <progress>
      <#shadow-root (user-agent) == $0>
        <div pseudo="-webkit-progress-inner-element">
          <div pseudo="-webkit-progress-bar">
            <div pseudo="-webkit-progress-value" style="width: -100%;"></div>
          </div>
        </div>
      </progress>
      <meter value="0.4">
        <#shadow-root (user-agent)>
          <div pseudo="-webkit-meter-inner-element">
            <div pseudo="-webkit-meter-bar">
              <div pseudo="-webkit-meter-optimum-value" style="width: 40%;"></div>
            </div>
          </div>
        </meter>
      </body>
    </html>

```

Quelle: Eigene Darstellung

Abbildung 5 zeigt die Struktur des DOM mit den HTML5-Elementen Progress und Meter. Der Aufbau dieser Elemente lässt sich in den Entwicklertools des Browsers betrachten, sofern die Anzeige des Shadow DOMs aktiviert wurde.

## 3. Stand der Technik

Im Folgenden wird der aktuelle Trend der Webentwicklung, der sich in Richtung des komponentenbasierten Designs bewegt, untersucht. Anschließend wird Angular analysiert und mit alternativen Frameworks verglichen.

### 3.1 Trend der Webentwicklung

Aufgrund der stetig wachsenden Anforderungen an Webseiten befindet sich die Webentwicklung in einem rasanten Wandel. Die hohe Anzahl der Endgeräte mit unterschiedlichen Bildschirmgrößen und Auflösungen führte bereits zu einer Etablierung des Responsive Webdesigns. Mithilfe dieses Ansatzes passen sich Webseiten flexibel an die Größe des Bildschirms an und bieten eine optimale Benutzererfahrung.

Allerdings hat sich nicht nur der Bereich des Webdesigns verändert. Die klassische Herangehensweise des server-side Renderings in Programmiersprachen wie PHP oder C# mit ASP.NET wird zunehmend um JavaScript ergänzt. Durch Bibliotheken zur Abstrahierung der Browser-Unterschiede ist es möglich, die Inhalte einer HTML-Seite im Browser zu manipulieren oder dynamisch weitere Daten zu laden. Populärstes Beispiel ist die JavaScript-Bibliothek jQuery.<sup>24</sup>

Der Einfluss von JavaScript wirkt sich zunehmend auf Server und Browser aus.

*Atwood's Law:* „*Any application that can be written in JavaScript, will eventually be written in JavaScript.*“ – Jeff Atwood, 2007

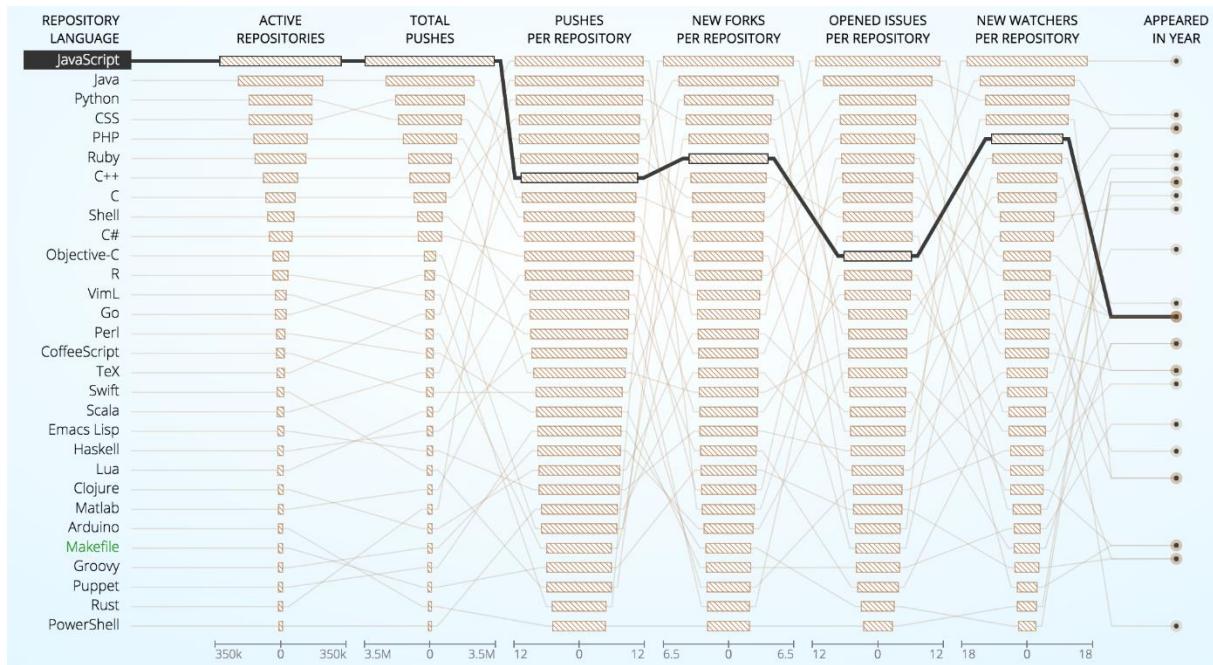
In 2009 wurde Node.js als Open-Source-Projekt zur Entwicklung serverseitiger Anwendungen in JavaScript veröffentlicht.<sup>25</sup> Als Laufzeitumgebung wird die V8 JavaScript Engine von Google verwendet, welche bereits im Browser Google Chrome eingesetzt wird.

---

<sup>24</sup> Vgl. Williams (2016)

<sup>25</sup> Abgerufen am 19. Juli 2016 von <https://github.com/nodejs/node-v0.x-archive/tags?after=v0.0.4>

Abbildung 6: Verbreitung von Programmiersprachen auf GitHub



Quelle: GitHut<sup>26</sup>

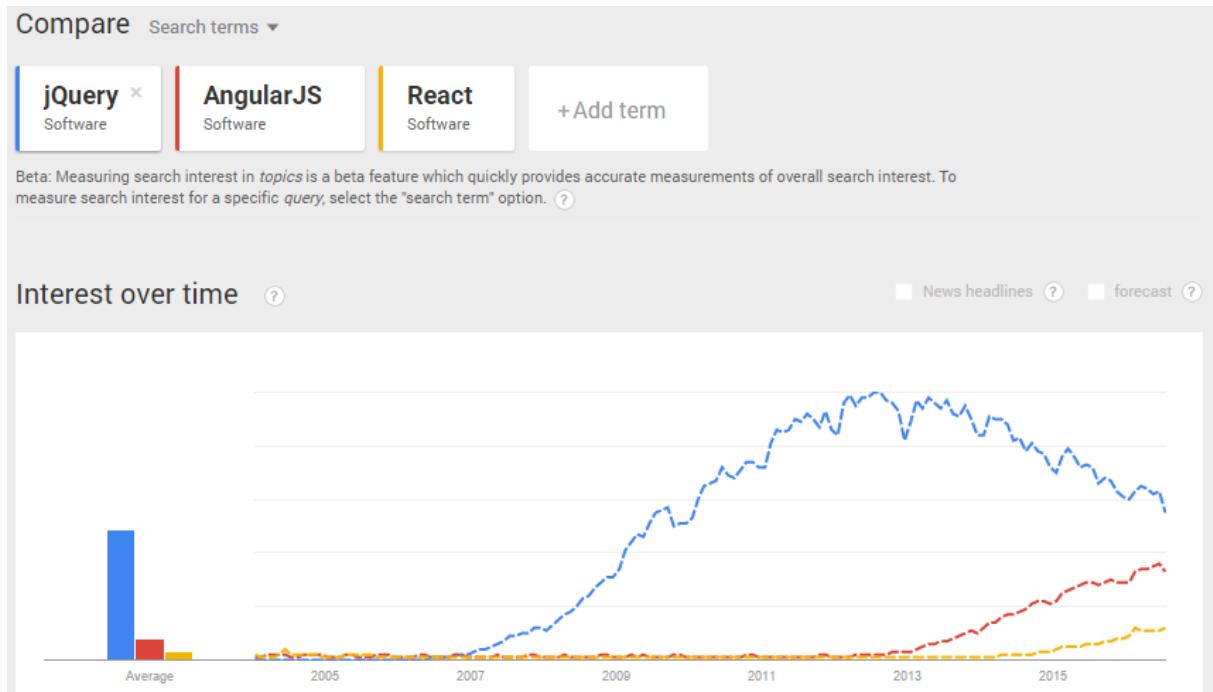
Darüber hinaus ist die Popularität von client-side JavaScript Frameworks gestiegen.<sup>27</sup> Als Alternative zum server-side Rendering bietet sich die Entwicklung von Single-page Applications (SPA) an. Bei dieser Art von Webanwendungen erfolgen nach dem initialen Aufruf der Seite keine weiteren Browser Reloads. Alle weiteren Daten und Navigationsaufrufe werden dynamisch nachgeladen. Für den Benutzer ergibt sich dadurch eine flüssige Bedienung der Anwendung, vergleichbar mit Desktop-Applikationen.

<sup>26</sup> Abgerufen am 20. Juli 2016 von <http://githut.info/>

<sup>27</sup> Vgl. Sayar (2015)

In Abbildung 7 zeichnet sich ein sinkendes Interesse an der Bibliothek jQuery ab dem Jahr 2013 ab. Zur gleichen Zeit nimmt die Anzahl der Suchanfragen nach dem Framework AngularJS zu.

Abbildung 7: Google Trend Analyse – jQuery, AngularJS & React



Quelle: Google Trends<sup>28</sup>

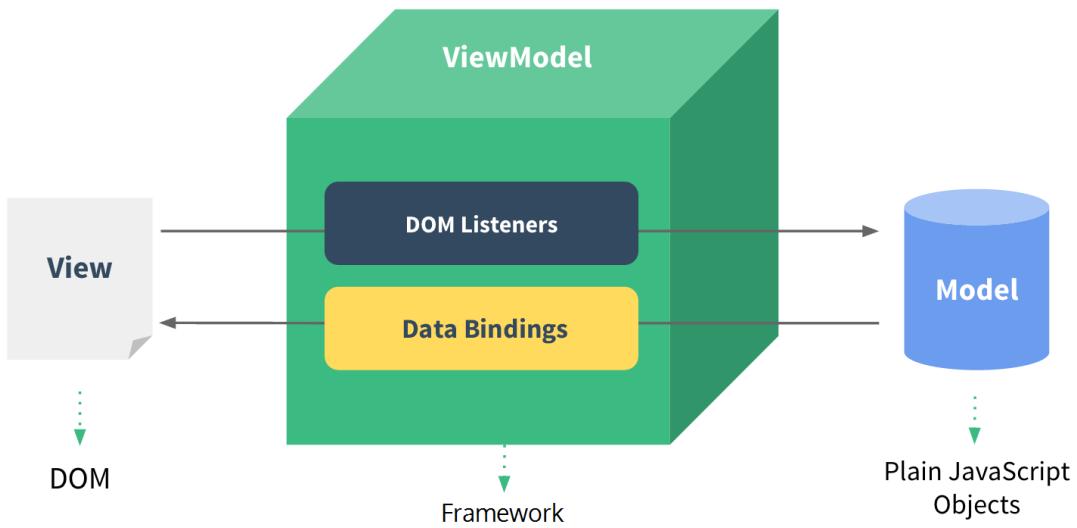
Aufgrund der Verwendung clientseitiger Frameworks zur Entwicklung von Webanwendungen ist es möglich, auf Bibliotheken wie jQuery zu verzichten. Die Manipulation des DOM im Browser wird durch das entsprechende JavaScript-Framework übernommen.

Dabei basieren Frameworks wie Backbone.js, Ember.js oder AngularJS auf dem MVC-Muster oder der Variation Model-View-ViewModel (MVVM). Bei MVVM wird eine Trennung zwischen der Darstellung (View) und der Logik für die Darstellung (ViewModel) vorgenommen. Auf diese Weise können Designer und Entwickler leichter unabhängig voneinander arbeiten.<sup>29</sup>

<sup>28</sup> Abgerufen am 19. Juli 2016 von <https://google.com/trends>

<sup>29</sup> Vgl. Microsoft (2012)

Abbildung 8: Model-View-ViewModel-Entwurfsmuster

Quelle: Vue.js<sup>30</sup>

Einige der neuen Frameworks, darunter Angular, React, Aurelia oder Vue.js, setzen vermehrt auf einen komponentenbasierten Ansatz. Auf diese Weise lässt sich die Benutzerschnittstelle einer Anwendung in möglichst kleine Komponenten herunterbrechen. Diese sind leicht wiederzuverwenden und in sich selbst gekapselt. Als Bausteine lassen sich diese Elemente frei zusammensetzen. Die meisten Applikationen können als Baum-Hierarchie dargestellt werden. In Listing 6 wird der typische Aufbau einer Website mit Navigationsleiste, Seitenleiste und Inhalt gezeigt. Dabei lassen sich diese einzelnen Elemente jeweils als eigenständige Komponente verwenden.

Listing 6: HTML Aufbau einer App mit Komponenten

```

1  <app-container>
2    <app-navigation></app-navigation>
3    <app-view>
4      <app-sidebar></app-sidebar>
5      <app-content></app-content>
6    </app-view>
7  </app-container>
```

Quelle: Eigene Darstellung

<sup>30</sup> Abgerufen am 19. Juli 2016 von <https://vuejs.org/guide/introduction.html>

Die Deklaration der Komponenten innerhalb der Frameworks unterscheidet sich gravierend. Mit Ausnahme von React setzen die meisten Frameworks eine domänenspezifische Sprache (DSL) ein, um HTML durch Logik und Datenbindung zu erweitern.

Im Gegensatz zum Design-Prinzip „Separation of Concerns“<sup>31</sup> vermischt React JavaScript mit HTML. Eigentlich handelt es sich um zwei verschiedene Problembereiche, die getrennt werden sollten. Die Vermischung der Struktur und Logik bringt einen entscheidenden Vorteil für JavaScript-Entwickler: Innerhalb der Komponenten können alle Funktionen und die bereits bekannte Konditionallogik aus JavaScript verwendet werden.

Eine weitere Entwicklung ist die Unterteilung monolithischer Frameworks in mehrere unabhängige Bibliotheken. Darum ist der Begriff *Framework* in einigen Fällen irreführend. Die Projekte React und Vue.js bezeichnen sich explizit als Bibliothek und beschäftigen sich nur mit dem „View“-Teil aus dem MVC-Prinzip.

*„React isn't an MVC framework. React is a library for building composable user interfaces. It encourages the creation of reusable UI components which present data that changes over time.“*

– Pete Hunt, 5. Juni 2013

*„Vue.js [...] is a library for building interactive web interfaces. The goal of Vue.js is to provide the benefits of reactive data binding and composable view components with an API that is as simple as possible. Vue.js itself is not a full-blown framework - it is focused on the view layer only.“* – Evan You, 18. Oktober 2015

Alle verbleibenden Bereiche eines typischen JavaScript Frameworks wie Routing, HTTP-Anfragen oder Architektur können durch beliebige andere Bibliotheken übernommen werden. Für die meisten Anwendungsfälle existieren auch offizielle Lösungen der Projekte.

## 3.2 Angular

Innerhalb dieses Kapitels werden die Neuerungen von Angular genauer untersucht. Zu Beginn wird kurz die Geschichte des Angular-Projektes zusammengefasst. Im Verlauf der Analyse

---

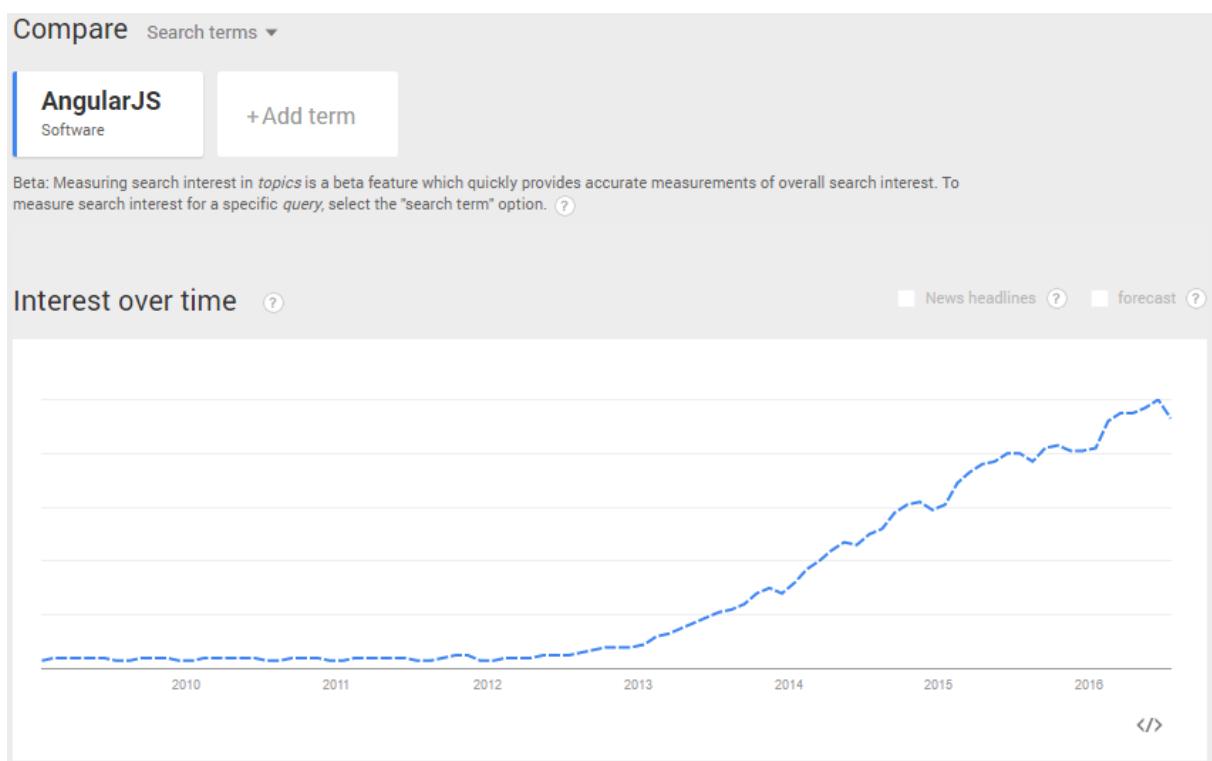
<sup>31</sup> Aufgerufen am 20. Juli 2016 von <http://stackoverflow.com/questions/98734/what-is-separation-of-concerns>

werden die neuen Funktionalitäten an passenden Stellen mit vergleichbaren Funktionen aus der Vorgängerversion verglichen.

#### 3.2.1 Geschichte

In 2009 wurde AngularJS als Nebenprojekt von Miško Hevery entwickelt.<sup>32</sup> Bereits ein Jahr später wurde die erste öffentliche Version auf GitHub veröffentlicht. Mittlerweile wird das Projekt durch Google unterstützt und weiterentwickelt. Als Open-Source-Projekt ist die Community maßgeblich an der Entwicklung beteiligt.

Abbildung 9: Google-Suchanfragen-Trend nach AngularJS



Quelle: Google Trends<sup>33</sup>

Durch die Unterstützung von Google wird das Interesse an AngularJS seit 2012 immer stärker. Seit 2014 verfolgt das Projekt aufgrund des eingestellten Supports für das Betriebssystem Windows XP und damit einhergehend für Internet Explorer 8 zwei separate Versionen. Für diese Browser-Version sind viel Code und aufwendige Tests nötig. Der stabile

<sup>32</sup> Vgl. Hevery (2009)

<sup>33</sup> Abgerufen am 21. Juli 2016 von <https://www.google.com/trends>

Entwicklungsstrang verzichtet auf den expliziten Support für Internet Explorer 8-Support und. Hierdurch kann er effizienter weiterentwickelt werden und bietet bessere Geschwindigkeit.<sup>34</sup>

Ebenfalls im Jahr 2014 wird Angular, allerdings als vollständige Neuentwicklung des Frameworks, angekündigt. Dadurch soll eine Reihe von Kritikpunkten der ersten Version aufgegriffen und behoben werden. Ein zentrales Problem von AngularJS ist die im Vergleich zu konkurrierenden oder neuen Frameworks schlechte Performance. Verantwortlich ist hierfür unter anderem das Alter des Frameworks. Die Unterstützung älterer Browser-Versionen und das System zur Erkennung von Änderungen führt gerade bei großen, komplexen Projekten zu einer langsamen Ausführung. Darüber hinaus ist das Framework nach einem monolithischen Design entworfen worden. Erst neuere Versionen führen optionale Erweiterungen ein. Die inhärenten Probleme von AngularJS lassen sich daher nur durch eine Neuentwicklung beseitigen. Dabei kann das Framework auf moderne Technologien ausgerichtet werden.

Gegen Ende des Jahres 2015 wurde das Erreichen des Beta-Stadiums von Angular angekündigt. Das Team hinter Angular beschreibt die Beta-Version als geeignet, um erfolgreich große Anwendungen zu entwickeln. Hinter den Kulissen wurden frühere Versionen bereits für interne Google-Projekte eingesetzt.<sup>35</sup>

Mittlerweile befindet sich Angular in der „Release Candidate“-Phase und steht damit kurz vor der finalen Veröffentlichung.

#### 3.2.2 Grundlagen

Im Gegensatz zu AngularJS soll durch Angular eine vollständige Plattform für die Entwicklung von Apps geschaffen werden. Das Motto des Projekts unterstreicht dieses Ziel.

„One framework. Mobile and desktop.“ – Angular

Mit dem Wissen über Angular soll es möglich sein, Anwendungen für beliebige Plattformen zu schreiben. Selbst für native Apps ist das Framework sowohl auf mobilen als auch auf Desktop-Geräten geeignet.

---

<sup>34</sup> Vgl. Minar (2013)

<sup>35</sup> Vgl. Green, Angular 2 Beta (2015)

Der Fokus liegt dabei auf der im Vergleich zu AngularJS verbesserten Geschwindigkeit des Frameworks. Darüber hinaus werden neue Funktionen der Web-Plattform, wie Service Worker, zur Erreichung einer parallelen Ausführung verwendet. Auch serverseitig lässt sich Angular nutzen, um den ersten Aufruf einer Seite zu rendern. Dadurch ist eine Seite für Suchmaschinen optimiert und steht dem Besucher schneller zur Verfügung.

Innerhalb der meisten Editoren und Entwicklungsumgebungen (IDE) wird die Syntax von Angular unterstützt. Für erhöhte Produktivität sorgt die automatische Code-Vervollständigung, insbesondere unter Verwendung typisierter JavaScript-Varianten.

Weiterhin wird die Testbarkeit der Anwendungen von Grund auf unterstützt. Die ursprünglich für AngularJS entwickelten Test-Systeme Karma und Protractor können auch für Anwendungen der neuen Generation eingesetzt werden.

Das Erstellen einfacher bis komplexer Animationen wird durch das Framework ebenfalls unterstützt. Für verbesserte Zugänglichkeit und Barrierefreiheit folgen Angular-Komponenten der Accessible Rich Internet Applications (ARIA) Initiative.<sup>36</sup>

#### 3.2.3 Programmiersprachen

Ein grundlegender Unterschied von Angular zu AngularJS wird bereits früh bei der Wahl der Programmiersprache sichtbar. Die erste Version des Frameworks wurde ausschließlich in JavaScript oder genauer ECMAScript 5 geschrieben. Bei ECMAScript handelt es sich um die standardisierte Spezifikation hinter JavaScript. Im Gegensatz dazu unterstützt die Neuentwicklung von Angular die folgenden Programmiersprachen:

- TypeScript
- JavaScript
- Dart

Aufgrund der Auswahlmöglichkeit kann Angular potenziell eine größere Entwicklergemeinschaft erreichen. Innerhalb der Dokumentation ist es möglich, über ein Dropdown-Menü die Programmiersprache der Beispiele zu ändern.

---

<sup>36</sup> Vgl. Google (2016)

Bis zum 20. Juli 2016 wurde das Projekt ausschließlich in TypeScript entwickelt. Die Versionen in JavaScript und Dart wurden durch Transpiler ermöglicht. Diese Art der Compiler übersetzt eine Quellsprache in ihr Äquivalent in der Zielsprache. Dabei arbeitet ein Transpiler üblicherweise auf der gleichen Abstraktionsebene zwischen den Programmiersprachen.<sup>37</sup>

Aufgrund von Programmierschwierigkeiten für Einsteiger und Inkompatibilitäten zwischen den Programmiersprachen wird die Dart-Version von Angular zukünftig durch ein eigenes Team entwickelt. Dadurch kann die API in allen Variationen des Frameworks vereinfacht werden.<sup>38</sup>

#### TypeScript

Mit dieser Programmiersprache führt Microsoft als Entwickler Typisierung und objektorientierte Prinzipien in JavaScript ein. Sie wird als Open-Source-Projekt seit 2012 entwickelt und wurde maßgeblich durch C# und Java beeinflusst. Die Kompatibilität zu JavaScript ist vollständig gewährleistet, und einige Neuerungen aus zukünftigen ECMAScript-Spezifikationen sind in TypeScript bereits implementiert. Jedes JavaScript-Programm ist demnach ein gültiges TypeScript-Programm. Aufgrund der optionalen Typisierung lassen sich bestehende JavaScript-Anwendungen leicht zu TypeScript migrieren. Die Implementierung von Angular in TypeScript bringt einige Vorteile für Entwickler. Eine häufige Fehlerquelle in JavaScript ist die dynamische Natur der Sprache. Durch die Typisierung ist der TypeScript-Compiler in der Lage, den Entwickler über Fehler durch inkompatible oder falsche Typen zu informieren. Das erhöht die Produktivität und verhindert Ausfälle nach der Auslieferung an den Kunden. Außerdem werden die Autovervollständigung und die allgemeine Sprachunterstützung innerhalb von Entwicklungsumgebungen verbessert.<sup>39</sup>

Weitere Schwächen von JavaScript werden durch die Unterstützung zukünftiger Spezifikationen der Sprache beseitigt. Besonders hilfreich für Angular ist der Einsatz von Decorators, um Meta-Informationen an Klassen und Funktionen zu hängen. Veranschaulicht

---

<sup>37</sup> Vgl. Sengstacke (2016)

<sup>38</sup> Vgl. Black (2016)

<sup>39</sup> Vgl. Malik, TypeScript: The Best Way to Write JavaScript (2015)

wird dies in Listing 7. Der Component-Decorator von Angular an die *AppComponent*-Klasse gehängt.

Listing 7: Angular Komponente in TypeScript

```

1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-component',
5   template: '<h1>App Component</h1>'
6 })
7 export class AppComponent {}
```

Quelle: Eigene Darstellung

Darüber hinaus unterstützt TypeScript das Modul-System aus ECMAScript 6. Auf diese Weise lässt sich Logik aus externen JavaScript-Dateien importieren.

## JavaScript

Die Verwendung der reinen JavaScript-Version von Angular ist sowohl für Einsteiger als auch komplexe Projekte sinnvoll, die nicht ohne weiteres auf eine andere Sprache wechseln können. Obwohl das Framework in ECMAScript 5 vorliegt, lässt sich die Entwicklung durch den Einsatz von Transpilern verbessern. Dadurch kann der eigene Quelltext neue Syntax und Funktionen aus kommenden JavaScript-Versionen nutzen.

Aufgrund der Generierung des Quelltextes dieser Ausgabe von Angular durch den TypeScript-Compiler ist die API im Vergleich unklarer und schwieriger zu nutzen. Für viele der genutzten Funktionen in TypeScript sind Behelfslösungen notwendig.

Listing 8: Angular Komponente in JavaScript

```

1 (function(app) {
2   app.AppComponent =
3     ng.core.Component({
4       selector: 'app-component',
5       template: '<h1>App Component</h1>'
6     })
7     .class({
8       constructor: function() {}
9     });
10 })(window.app || (window.app = {}));
```

Quelle: Eigene Darstellung

Um die globale Umgebung des laufenden JavaScript-Programms nicht mit lokalen Variablen zu füllen, wird eine *Immediately Invoked Function Expression (IIFE)* verwendet. Andernfalls könnte es versehentlich zu mehrfach deklarierten Bezeichnern kommen.<sup>40</sup> Mit ECMAScript 6 ist der Einsatz einer IIFE aufgrund der Modularisierung nicht mehr notwendig. Jedes Modul besitzt seinen eigenen Geltungsbereich und beeinflusst die globale Umgebung nicht implizit.

Der Import von Funktionalität und die Verwendung von Decorators entfallen bei der JavaScript-Version. Stattdessen muss die gewünschte Funktion über das globale Angular-Objekt aufgerufen werden. Auch die Definition von Klassen lässt sich in ECMAScript 5 noch nicht verwenden. Daher muss ein normales JavaScript-Objekt mit entsprechenden Attributen übergeben werden.

#### Dart

Seit 2011 wird Dart als Open-Source-Programmiersprache von Google entwickelt. Dabei soll sie allgemein für alle Szenarien eingesetzt werden können, insbesondere für Web-, Server- und mobile Applikationen. Unter anderem werden Klassen, Vererbung, objektorientierte Prinzipien, Interfaces und optionale Typisierung unterstützt.

Zur Ausführung von Programmen in Dart wird die zugehörige Virtual Machine (VM) benötigt. Eine Integration der Dart VM in alle Browser ist unrealistisch. Pläne, diese in Google Chrome einzubauen, wurden aufgrund der Fragmentierung des Webs kritisiert.<sup>41</sup> Daher lässt sich die Sprache mithilfe des dart2js-Transpilers zu JavaScript kompilieren. Der übersetzte Code kann wiederum durch alle bekannten Browser – in Einzelfällen durch Optimierungen auch schneller – ausgeführt werden.

Für Angular ergeben sich aus Dart ähnliche Vorteile wie bei TypeScript. Die optionale Typisierung unterstützt viele Funktionalitäten und kann, insbesondere bei komplexen Projekten, die Entwicklung verbessern.

---

<sup>40</sup> Vgl. Alman (2010)

<sup>41</sup> Vgl. Bak & Lund (2015)

Listing 9: Angular Komponente in Dart

```
1 import 'package:angular2/core.dart';
2
3 @Component(
4   selector: 'app-component',
5   template: '<h1>App Component</h1>'
6 )
7 class AppComponent {}
```

Quelle: Eigene Darstellung

Die Syntax unterscheidet sich leicht von TypeScript, und der Pfad der Import-Anweisung muss angepasst werden. Dart verfügt über einen eingebauten Paket-Manager und organisiert Referenzen auf externe Pakete daher abgeändert.

### 3.2.4 Vergleich der Programmiersprachen

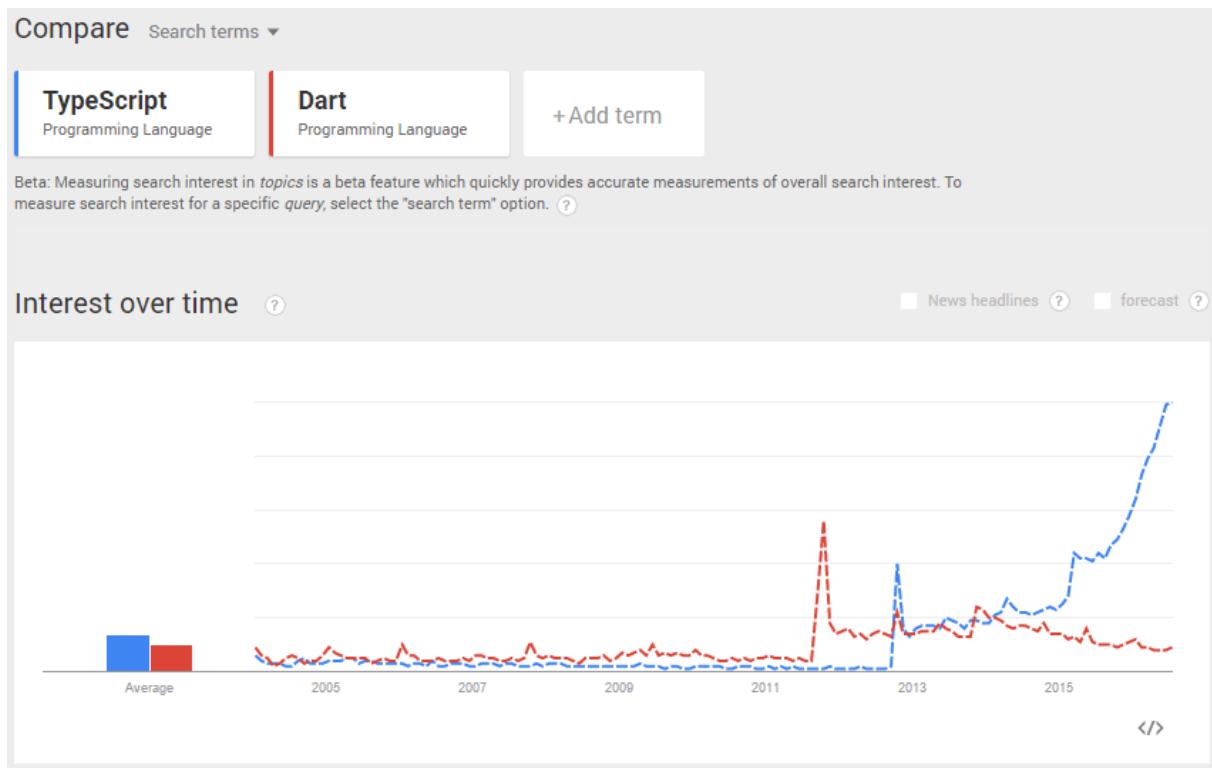
Jede Edition von Angular lässt sich einer spezifischen Zielgruppe zuordnen. Dabei kann die Wahl der Programmiersprache, abhängig vom Einsatzszenario, Vor- und Nachteile bringen. Die Entscheidung sollte gut überlegt sein, da eine nachträgliche Migration viele Herausforderungen an ein Team stellt.

Der Hauptfokus des Angular-Teams liegt auf der TypeScript-Version. Diese Tatsache spiegelt sich in der Entscheidung wider, die beiden anderen Versionen durch Transpiler zu generieren. Seit kurzer Zeit steht der Dart-Version ein dediziertes Team zur Verfügung, daher werden zusätzlich Ressourcen eingesetzt, um diese Edition für den produktiven Einsatz zu verbessern.

Allgemein sollte die einfache JavaScript-Version nur verwendet werden, wenn Einschränkungen den Einsatz von TypeScript verhindern. Die API muss oft über lange Bezeichner angesprochen werden. Daher müssen Entwickler, vor allem zu Beginn, oft in der API-Referenz nachschlagen. Hierdurch wird die Entwicklungsgeschwindigkeit gesenkt und die Lesbarkeit des Quelltextes ist beeinträchtigt. Darüber hinaus verwendet Angular viele Funktionen aus TypeScript, welche in JavaScript nur schwer nutzbar sind.

Grundsätzlich ähneln sich TypeScript und Dart in Bezug auf Angular. Insbesondere die optionale Typisierung und die Implementierung zukünftiger ECMAScript-Funktionen lassen den Code kompakt, lesbar und ausdrucksvooll werden.

Abbildung 10: Google-Trend-Analyse – TypeScript & Dart



Quelle: Google Trends<sup>42</sup>

Allerdings fristet Dart als Programmiersprache im Web eher ein Nischendasein. Die Sprache wird in den meisten Fällen für Google-Produkte eingesetzt. Wie Abbildung 10 zeigt, bleiben die Suchanfragen zu Dart seit der Veröffentlichung im Jahr 2011 eher konstant bzw. sinken leicht. Im Gegensatz dazu ist das Interesse an TypeScript speziell seit 2015 stark gestiegen. Dies kann der Unterstützung der React-Syntax sowie der Implementierung neuer JavaScript-Funktionen zugeschrieben werden. Auf der Plattform *Stack Overflow* können Entwickler Fragen stellen und beantworten. Ein Indiz für die Größe der Community, sowie der Popularität liefert ein Vergleich der aufgegebenen Fragen auf der Plattform. Darin finden sich zu TypeScript 20.402 Beiträge, für Dart hingegen gibt es lediglich 9.006 Ergebnisse.<sup>43</sup>

Teams mit dem Fokus auf Dart-Projekten und Entwickler mit Dart-Erfahrung sollten in jedem Fall die entsprechende Version von Angular verwenden. In allen anderen Fällen ist TypeScript in Kombination mit Angular zu bevorzugen. Das Hauptaugenmerk von Google und der

<sup>42</sup> Abgerufen am 25. Juli 2016 von <https://www.google.com/trends>

<sup>43</sup> Abgerufen am 25. Juli 2016 von <http://stackoverflow.com/search>

Community liegt auf TypeScript, dies spiegelt sich auch in der Dokumentation wider. Ein Argument für die sichere Zukunft von TypeScript ist die Unterstützung durch drei große Firmen: Microsoft als Hauptentwickler, Google mit Angular und Facebook mit React.

Tabelle 2: Vergleich von TypeScript, JavaScript und Dart für Angular

	TypeScript	JavaScript (ES5)	Dart
<b>Entwickler</b>	Microsoft	Ecma International	Google
<b>Erscheinungsjahr</b>	2012	2009	2011
<b>Module</b>	Ja	Nein (ab ES6)	Ja
<b>Klassen</b>	Ja	Nein (ab ES6)	Ja
<b>Typisierung</b>	Optional	Nein	Optional
<b>Decorator</b>	Ja	Nein (ab ES7)	Ja
<b>Transpiler</b>	Ja	Nein (ab ES6)	Ja
<b>Dokumentation<sup>1</sup></b>	99% <sup>2</sup>	26%	70%
<b>Stack Overflow<sup>1</sup></b>	1.811 Fragen	566 Fragen	114 Fragen

ES = ECMAScript

<sup>1</sup>Stand 25. Juli 2016

<sup>2</sup>Die TypeScript-Dokumentation wird noch aktiv überarbeitet.

Quelle: Eigene Darstellung

### 3.2.5 Applikationszyklus

Der Start des Lebenszyklus jeder AngularJS- oder Angular-Anwendung wird durch die Bootstrap-Methode ausgelöst. Die darauffolgenden Phasen unterscheiden sich je nach Framework-Version. In AngularJS kann die Initialisierung über zwei Wege ausgelöst werden:

- Automatische Initialisierung
- Manuelle Initialisierung

Beim automatischen Mechanismus wartet das Framework darauf, dass die Inhalte des DOM vollständig geladen sind. Vorher kann die Kompilierung der HTML-Elemente nicht begonnen werden. Das Framework durchsucht die Seite nach dem Auftreten der *ng-app* Direktive und verwendet den angegebenen Namen, um das Modul zu laden.

Eine höhere Flexibilität bietet die manuelle Initialisierung der Anwendung. Bei dieser Vorgehensweise sollte die *ng-app* Direktive nicht vorkommen. Stattdessen wird die Bootstrap-Methode von AngularJS genutzt, um den Start der Applikation zu signalisieren.

In beiden Versionen von Angular müssen sämtliche Module der Anwendung vor dem Bootstrap-Prozess am Framework registriert sein. Eine nachträgliche Registrierung von Elementen ist nicht vorgesehen.

Nach dem Bootstrap-Aufruf durchläuft die Applikation die Config- und Run-Blöcke der Module. Im Grunde besteht ein Modul in AngularJS ausschließlich aus einer beliebigen Anzahl dieser beiden Blöcke. Sämtliche weitere Methoden dienen als syntaktischer Zucker für eine einfache Registrierung der Elemente einer Anwendung.

Listing 10: Config- und Run-Block eines AngularJS Moduls

```
1 angular.module('app', [])
2     .value('myValue', 42)
3     .directive('myDirective', ...)
4     .config(function($provide, $compileProvide) {
5         $provide.value('myValue', 42); // = .value()
6         $compileProvide.directive('myDirective', ...); // .directive()
7     })
8     .run(function() {
9         // Vergleichbar zur main()-Methode einer Anwendung
10    });

```

Quelle: Eigene Darstellung

Angular vereinfacht die Komplexität der Startphase. Auf explizite Konfigurations- und Run-Blöcke wird verzichtet. Diese können durch den Entwickler frei implementiert oder ausgelassen werden. Um in den Lebenszyklus der Anwendung einzugreifen, können mehrere Events genutzt werden. Diese werden in Angular als *Lifecycle Hooks* bezeichnet und können durch Komponenten abgefangen werden. Jede Methode dieser Events wird zu einem fest definierten Zeitpunkt aufgerufen. Der Lebenszyklus von Komponenten reicht von der Erstellung, über die

Aktualisierung und schlussendlich zur Zerstörung, sobald das Element aus dem DOM entfernt wurde. Nach dem Aufruf des Konstruktors einer Komponente werden die *Lifecycle Hooks* in der folgenden Reihenfolge aufgerufen:<sup>44</sup>

- ngOnChanges
- ngOnInit
- ngDoCheck
- (*ngAfterContentInit*)
- (*ngAfterContentChecked*)
- (*ngAfterViewInit*)
- (*ngAfterViewChecked*)
- ngOnDestroy

Dabei können die Events in Klammern nur innerhalb von Komponenten verwendet werden. Alle anderen *Lifecycle Hooks* können sowohl durch Komponenten als auch durch Direktiven implementiert werden. Unter Verwendung von TypeScript können Interfaces eingesetzt werden, um die Semantik zu erhöhen. Dadurch wird auf einen Blick klar, welche Methoden durch die Komponente implementiert werden. Der Name der entsprechenden Interfaces entspricht dem der Methoden ohne den *ng*-Prefix.

Listing 11: *OnInit* Lifecycle Hook in Angular

```

1 import { OnInit } from '@angular/core';
2
3 export class LifecycleComponent implements OnInit {
4     constructor() {
5         // Wird zuerst aufgerufen
6     }
7
8     ngOnInit() {
9         // Wird nach dem Konstruktor aufgerufen
10    }
11 }
```

Quelle: Eigene Darstellung

---

<sup>44</sup> Vgl. Abgerufen am 15. August 2106 von <https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html#!#hook-sequence>

### 3.2.6 Komponenten

Bereits in AngularJS 1.5 wird der komponentenbasierte Ansatz unterstützt und empfohlen. Im Vergleich zu Direktiven besitzen Komponenten eine simplere Konfiguration und erlauben es, die Anwendung in Richtung Web-Komponenten und Angular zu entwickeln.<sup>45</sup>

Listing 12: Komponente in AngularJS 1.5

```

1  class ComponentController {
2      constructor() {
3          this.message = 'Hello World!';
4      }
5  }
6
7  angular
8      .module('app', [])
9      .component('app-component', {
10          controller: ComponentController,
11          template: '<h1>App Component</h1><p>{{ $ctrl.message }}</p>'
12      });

```

Quelle: Eigene Darstellung

In Angular bilden Komponenten die Grundsteine für die gesamte Anwendung. Der Aufbau lässt sich als Komponenten-Baum darstellen. Dabei wird die Wurzel durch eine Root- oder App-Komponente gebildet. Diese enthält in ihrem Template üblicherweise weitere Komponenten zum Aufbau der Anwendung.

Im Grunde lassen sich Komponenten als Kombination aus Template und Controller bezeichnen. Das Template enthält die HTML-Struktur sowie Angular-spezifische Anweisungen zur Datenbindung und Event-Weitergabe. Der zugehörige Controller agiert als Vermittler zwischen dem Template und der Geschäftslogik. Diese sollte sich niemals im Controller befinden und stattdessen in eigene Service-Klassen ausgelagert werden.

Darüber hinaus können Daten über Attribute an Komponenten übergeben werden. Über Events kann eine Komponente mit ihrem übergeordneten Element kommunizieren.

---

<sup>45</sup> Vgl. Google (2016)

Listing 13: Angular Komponente mit Input- und Output-Attributen

```

1  // example.component.ts
2  import { Component, Input, Output, EventEmitter } from '@angular/core';
3  @Component({
4      selector: 'example-component',
5      template: `
6          <h1>{{ message }}</h1>
7          <button (click)="alert()"></button>`  

8  })
9  export class ExampleComponent {
10     @Input() message: string;
11     @Output() onAlert = new EventEmitter<string>();
12     alert() {
13         this.onAlert.emit(this.message);
14     }
15 }
16
17 // app.component.ts
18 import { Component } from '@angular/core';
19 import { ExampleComponent } from './example.component.ts';
20 @Component({
21     selector: 'app-component',
22     template: '<example-component [message]="message" (onAlert)="onAlert($event)" />'  

23 })
24 export class AppComponent {
25     message: string = 'Hello World!';
26     onAlert(message: string) {
27         window.alert(message);
28     }
29 }

```

Quelle: Eigene Darstellung

Mithilfe des Decorator-Features sind Attribute einer Komponenten-Klasse als Eingabe- oder Ausgabe-Attribute markierbar. Die Namen dieser Attribute können anschließend übernommen werden, um Daten an das Element zu übergeben oder um auf ein Event zu warten.

### 3.2.7 Templates

Die Interaktion mit einer Angular-Anwendung findet primär über die Benutzeroberfläche statt. Diese wird durch HTML definiert und in Templates abgelegt. Jede Komponente besitzt ein zugehöriges Template. Nach dem MVC-Prinzip ist das Template die View und die

Komponente der Controller. Um eine Verbindung zwischen Komponente und Template herzustellen, reicht HTML alleine nicht aus. Die Syntax wird um Angular-Ausdrücke und -Anweisungen erweitert. Auf diese Weise lassen sich Interpolationen durchführen, Attribute binden, Events abfangen und Kontrollfluss-Anweisungen verwenden.

## Interpolation

Mit dieser Technik werden Ausdrücke durch Angular evaluiert und anschließend nahtlos in das resultierende HTML eingebunden.

Listing 14: Einfache Interpolation in Angular

```

1  <p>The quick brown {{'fox'}} jumps over the lazy dog.</p>
2  <!-- Ergebnis: "The quick brown fox jumps over the lazy dog. -->
3  <p>The result of 42 + 8 is: {{ 42 + 8 }}</p>
4  <!-- Ergebnis: "The result of 42 + 8 is: 50 -->
```

Quelle: Eigene Darstellung

Ein viel wichtiger Anwendungsfall ist jedoch die Einbindung von Daten oder der Aufruf von Funktionen an einer Komponente. Hierfür wird der Name des Attributs oder der Funktion als Ausdruck evaluiert und durch das entsprechende Ergebnis ersetzt.

Listing 15: Angular-Interpolation mit Komponente

```

1  @Component({
2      // ...
3      template: `
4          <p>Attribut: {{message}}</p>
5          <p>Funktion: {{calculate()}}</p>
6          <p>Optional: {{data?.result}}</p>
7      `})
8  export class Component {
9      public message: string = 'Hello World!';
10     public data: object = null;
11
12     constructor() {
13         setTimeout(() => this.data = { result: true }, 1000);
14     }
15
16     calculate() {
17         return 42 + 8;
18     }
19 }
```

Quelle: Eigene Darstellung

Im Vergleich zur vorherigen Version AngularJS ist die Syntax, unter Nutzung doppelter geschweifter Klammern, unverändert geblieben. Eine Neuerung stellt der optionale Operator dar. Bei dem Versuch, ein Attribut von einem nicht vorhandenen Objekt (*null*) abzurufen, wird ein Fehler ausgelöst. Für den Benutzer ist die gesamte, durch Angular gerenderte, Ansicht nicht mehr sichtbar. Durch das Hinzufügen eines Fragezeichens wie in Zeile 8 in Listing 14 wird die Anwendung gegen fehlende Werte gesichert. In der gerenderten Ansicht ist der Ausdruck nicht sichtbar, solange das Objekt *null* ist.

## Binding

Dieser Mechanismus dient dazu, einen Datenfluss zwischen Komponente und Template herzustellen. Ursprünglich wurde diese Aufgabe unter Zuhilfenahme von Bibliotheken wie jQuery manuell programmiert. Dieser Ansatz produziert mit der Zeit viel Code, welcher gewartet werden muss und fehleranfällig ist.<sup>46</sup> Daher übernimmt Angular diese Aufgabe als Framework. In Angular existieren drei grundlegende Arten von Bindings, dargestellt in Abbildung 11.

Abbildung 11: Angular-Bindings

Data direction	Syntax	Binding type
One-way from data source to view target	<code>{{expression}}</code> <code>[target] = "expression"</code> <code>bind-target = "expression"</code>	Interpolation Property Attribute Class Style
One-way from view target to data source	<code>(target) = "statement"</code> <code>on-target = "statement"</code>	Event
Two-way	<code>[(target)] = "expression"</code> <code>bindon-target = "expression"</code>	Two-way

Quelle: Angular-Dokumentation<sup>47</sup>

<sup>46</sup> Vgl. Google (2016)

<sup>47</sup> Abgerufen am 26. Juli 2016 von <https://angular.io/docs/ts/latest/guide/template-syntax.html#!#binding-syntax>

Mit der ersten Variante wird ein Property eines HTML-Elements an einen Angular-Ausdruck gebunden. So entsteht ein Datenfluss, in eine Richtung, von der Datenquelle zum Zielelement in der View. Auch die zuvor besprochene Interpolation zählt zu dieser Art des Bindings. Einige beispielhafte Anwendungsfälle sind:

- Manipulation des CSS-Styles oder der Klassen eines Elements (*style, class*)
- Setzen der URL an einem Bild-Element (*src*)
- Deaktivierung eines Buttons (*disabled*)

In AngularJS existieren über 70 eingebaute Direktiven, um Properties zu beeinflussen oder Events abzufangen.<sup>48</sup> Für Entwickler ist es ein großer Aufwand, diese Direktiven zu lernen und ihre Verwendung in der Dokumentation nachzuschlagen.

Mit Angular entfällt ein Großteil der eingebauten Direktiven. Es genügt, die Binding-Syntax zu kennen. Darüber hinaus wird lediglich der Name des Property oder des Events benötigt.

In der folgenden Tabelle werden die Binding-Direktiven von AngularJS und Angular verglichen. Zur besseren Übersicht werden HTML-Tags in den Beispielen ausgelassen. Der Ausdruck `$ctrl` in den Beispielen zu AngularJS ist die Bezeichnung des Controllers. Ab der Version 1.5 ist dies der Standardbezeichner unter Verwendung der Component-Funktion.<sup>49</sup> Außerdem wird dieser Name in Styleguides für diese AngularJS-Version empfohlen.<sup>50</sup> Ein Controller-Name ist in Angular nicht mehr notwendig. Die Bezeichner im Template beziehen sich immer auf die aktuelle Komponente.

---

<sup>48</sup> Abgerufen am 27. Juli 2016 von <https://docs.angularjs.org/api/ng/directive>

<sup>49</sup> Abgerufen am 27. Juli 2016 von <https://docs.angularjs.org/guide/component>

<sup>50</sup> Vgl. Motto (2016)

Tabelle 3: Property Binding Direktiven in AngularJS und Angular

AngularJS	Angular
<code>ng-style="{'color: \$ctrl.elementColor'}"</code>	<code>[ngStyle]={"color: elementColor"}</code> <code>[style.color]="elementColor"</code>
<code>ng-class="{'active: \$ctrl.isActive'}"</code>	<code>[ngClass]={"active: isActive"}</code> <code>[class.active]="isActive"</code>
<code>ng-show="\$ctrl.isVisible"</code> <code>ng-hide="\$ctrl.isHidden"</code>	<code>[hidden]="'isHidden"</code>
<code>ng-src="{{ctrl.imageUrl}}"</code>	<code>[src]="'imageUrl"</code>

Quelle: Eigene Darstellung

Anstelle der Verwendung eckiger Klammern kann auf die kanonische Syntax zurückgegriffen werden. Hierfür wird ein *bind-* vor den Namen des Property gestellt.

Mithilfe der zweiten Variante des Bindings lassen sich Events abfangen und an Komponenten weitergeben. In diesem Fall geht der Einweg-Datenfluss von dem Element in der View zu einer Komponente. Auch an dieser Stelle vereinfacht Angular das Abfangen von Events aufgrund der neuen Syntax-Schreibweise.

Tabelle 4: Event Bindings in AngularJS und Angular

AngularJS	Angular
<code>ng-click="\$ctrl.onClick(\$event)"</code>	<code>(click)="onClick(\$event)"</code> <code>on-click="onClick(\$event)"</code>
<code>ng-change="\$ctrl.onChange(\$event)"</code>	<code>(input)="onInput(\$event)"</code> <code>on-input="onInput(\$event)"</code>

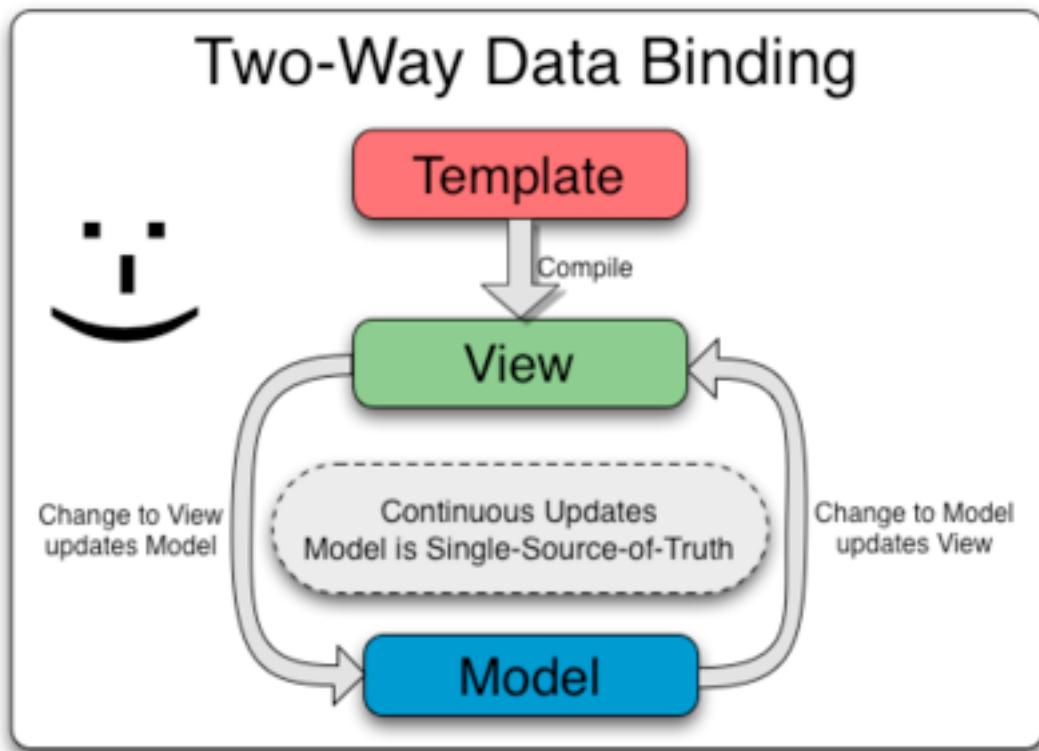
Quelle: Eigene Darstellung

Die kanonische Syntax setzt *on-* vor den Namen des entsprechenden Events.

Eines der Alleinstellungsmerkmale von AngularJS ist die Realisierung bidirektonaler Datenbindung (Two-way data binding). Dabei werden die ersten beiden Varianten kombiniert,

um eine kontinuierliche Synchronisation zwischen Model und View herzustellen. Änderungen in der View, beispielsweise das Tippen in einem Eingabefeld, werden mit dem Model synchronisiert. Umgekehrt lösen Änderungen am Model eine Anpassung der View aus.

Abbildung 12: Bidirektionale Datenbindung



Quelle: Data Binding in AngularJS<sup>51</sup>

Diese Art der Datenbindung wird auch in Angular weiterhin unterstützt. Besonders hilfreich ist diese Technik bei der Erstellung von Formularen und der clientseitigen Validation der Benutzereingaben.

Tabelle 5: Bidirektionale Datenbindung in AngularJS und Angular

AngularJS	Angular
<code>ng-model="\$ctrl.text"</code>	<code>[(ngModel)]="text"</code>

Quelle: Eigene Darstellung

<sup>51</sup> Abgerufen am 27. Juli 2016 von <https://docs.angularjs.org/guide/databinding>

Dabei dient die Schreibweise mit eckigen und runden Klammern zugleich als Abkürzung. Das Angular-eigene Property `ngModel` wird gebunden und gleichzeitig als Event genutzt, um über Änderungen benachrichtigt zu werden. Listing 16 zeigt ausführliche Schreibweisen für die bidirektionale Datenbindung.

Listing 16: Ausführliche Syntax für bidirektionale Datenbindung in Angular

```
1 <input [value] = "text" (input) = "text = $event.target.value">
2 <input [ngModel] = "text" (ngModelChange) = "text = $event">
```

Quelle: Eigene Darstellung

In der ersten Zeile wird das Value-Property gebunden, um den Inhalt des Eingabefelds auf das entsprechende Model zu setzen. Über das Input-Event werden zur Aktualisierung des Models Änderungen abgefangen. Das Detailwissen wird in der zweiten Zeile durch eingebaute Angular-Direktiven versteckt.

## Struktur-Direktiven

Das Framework erlaubt die Manipulation des Templates durch Bedingungen (*if*) und Schleifen (*for*). Diese Direktiven unterscheiden sich aufgrund ihrer Komplexität von simplen Binding-Attributen. Abhängig vom Zustand der Anwendung erlauben sie das Ausblenden ganzer Teile des DOM oder die Wiederholung bestimmter Elemente. Im Gegensatz zu einer serverseitigen Webanwendung muss dafür nicht die vollständige Seite neu geladen werden.

Tabelle 6: Struktur-Direktiven in AngularJS und Angular

AngularJS	Angular
<code>ng-if="\$ctrl.elements.length &gt; 0"</code>	<code>*ngIf="elements.length &gt; 0"</code>
<code>ng-repeat="element in \$ctrl.elements"</code>	<code>*ngFor="let element of elements"</code>
<code>ng-switch="\$ctrl.type"</code>	<code>[ngSwitch] = "type"</code>
<code>    ng-switch-when="true"</code>	<code>    *ngSwitchCase="true"</code>
<code>    ng-switch-when="false"</code>	<code>    *ngSwitchCase="false"</code>
<code>    ng-switch-default</code>	<code>    *ngSwitchDefault</code>

Quelle: Eigene Darstellung

Mit der If-Anweisung wird das entsprechende Element, abhängig von der Auswertung des Ausdrucks, ein- oder ausgeblendet. Allerdings betrifft es im Gegensatz zu `ng-show` und `ng-`

*hide* aus AngularJS nicht die Sichtbarkeit des Elements. Bei diesen Direktiven bleibt das Element im DOM, während das *hidden*-Property manipuliert wird. Dahingegen wird das Element, inklusive seiner Kinder, vollständig aus dem DOM entfernt, sofern die Auswertung des zugehörigen Ausdrucks einen unwahren Wert (*false*) ergibt.

Die Wiederholung von Elementen wird durch die For-Schleife ermöglicht. Dafür iteriert Angular über das angegebene Objekt und fügt bei jedem Durchlauf eine Kopie des Elements ein. Eine Neuerung ist die Verwendung des *let*-Keywords zu Beginn der Schleifendefinition. Für die Lebensdauer der Schleife wird eine lokale Variable erzeugt, welche das aktuelle Element referenziert. Außerhalb des Schleifenblocks kann diese Variable nicht referenziert werden. Damit folgt Angular der Definition des *let*-Keywords aus ECMAScript 6 und macht die Variable nur im lexikalischen Block verfügbar.<sup>52</sup>

Eine weitere Besonderheit der Struktur-Direktiven ist der vorangestellte Stern. Hierbei handelt es sich um syntaktischen Zucker zur Vereinfachung des Schreibens und Lesens dieser Anweisungen.<sup>53</sup> Im Hintergrund setzt Angular das Template-Element ein, um den Inhalt der entsprechenden Elemente zu verwenden.<sup>54</sup>

Tabelle 7: Syntaktischer Zucker bei *ngIf* und *ngFor*

```

1   <div *ngIf="shouldBeIncluded">The quick brown fox jumps over the lazy dog</div>
2   // wird zu
3   <template [ngIf]="shouldBeIncluded">
4     <div>The quick brown fox jumps over the lazy dog</div>
5   </template>
6
7   <div *ngFor="let element of elements">{{ element }}</div>
8   // wird zu
9   <template ngFor let-element [ngForOf]="elements">
10    <div>{{ element }}</div>
11  </template>
```

Quelle: Eigene Darstellung

---

<sup>52</sup> Vgl. Ecma International (2015)

<sup>53</sup> Abgerufen am 27. Juli 2016 von <https://angular.io/docs/ts/latest/guide/structural-directives.html#!#asterisk>

<sup>54</sup> Siehe 2.3.1

#### 3.2.8 Services

Ein Service ist ein allgemeiner Begriff für einen Wert, eine Funktion oder eine Klasse, die eine Applikation benötigt.<sup>55</sup> Üblicherweise handelt es sich dabei um eine Klasse mit einem spezifischen Zweck. Nach dem *Single Responsibility Principle* (SRP) sollte ein Service nur eine Aufgabe besitzen und diese gut erledigen.<sup>56</sup>

„*The Single Responsibility Principle (SRP) states that each software module should have one and only one reason to change.*“ – Robert C. Martin

Ursprünglich gibt es in AngularJS eine Reihe von Funktionen zur Registrierung von Services:

- Constant
- Value
- Factory
- Service
- Provider

Unter anderem ist dies durch die Tatsache bedingt, dass es zu Beginn von AngularJS noch kein Modul-System in der JavaScript-Spezifikation gab. Daher bietet das Framework zur Registrierung von Modulen und anderer Elemente der Anwendung ein eigenes System an. Für Entwickler ergibt sich aufgrund fünf verschiedener Wege zur Registrierung eines Services ein erheblicher Mehraufwand.

---

<sup>55</sup> Vgl. Google (2016)

<sup>56</sup> Vgl. Martin, The Single Responsibility Principle (2014)

Listing 17: Service Definition in AngularJS

```

1  // example.service.js
2  export default class ExampleService {
3      sharedFunctionality(message) {
4          console.log(message);
5      }
6  }
7
8  // app.module.js
9  import ExampleService from './example.service';
10
11 angular
12     .module('app', [])
13     .service('exampleService', ExampleService);

```

Quelle: Eigene Darstellung

In Angular ist das Prinzip der Services nicht mehr an das Framework gekoppelt. Dennoch sind sie integraler Bestandteil einer Angular-Anwendung. Über Services lässt sich Funktionalität an einer zentralen Stelle ablegen, um sie für mehrere andere Klassen zur Verfügung zu stellen. Die Geschäftslogik lässt sich damit isolieren und mit Services innerhalb von Komponenten verwenden. Ähnlich wie Controller sollten Komponenten keine Geschäftslogik enthalten.

Ein Service wird in Angular üblicherweise als eine einfache JavaScript-Klasse erstellt. Der *Injectable* Decorator sorgt dafür, dass der Service von anderen Klassen durch das *Dependency-  
Injection*-System genutzt werden kann.<sup>57</sup>

Listing 18: Service-Definition in Angular

```

1  import { Injectable } from '@angular/core';
2
3  @Injectable()
4  class ExampleService {
5      sharedFunctionality(message: string) {
6          console.log(message);
7      }
8  }

```

Quelle: Eigene Darstellung

<sup>57</sup> Siehe 3.2.9

### 3.2.9 Dependency Injection

Bei Dependency Injection werden die Abhängigkeiten einer Klasse nicht durch diese selbst erzeugt. Nach dem SRP ist das nicht die eigentliche Aufgabe der Klasse. Daher werden die Abhängigkeiten an einer zentralen Stelle verwaltet und an ihre Einsatzstelle weitergereicht. Normalerweise wird diese Verantwortlichkeit durch ein Dependency Injection Framework übernommen. Es existieren verschiedene Ansätze zur Realisierung dieses Entwurfsmusters:<sup>58</sup>

- Constructor Injection
- Interface Injection
- Setter Injection
- Andere (z.B. Reflection oder Property Injection)

Angular verwendet in beiden Versionen des Frameworks den Ansatz *Constructor Injection*. Die Abhängigkeiten werden im Konstruktor des Objekts oder der Klasse definiert. Bei der Instanziierung kümmert sich das Framework darum, die benötigten Abhängigkeiten aufzulösen und an das Ziel weiterzugeben.

Listing 19: Dependency Injection Varianten in AngularJS

```

1  const module = angular.module('app', []);
2
3  // Inline Array
4  module.controller('InlineController', ['$scope', '$http', function($scope, $http) {
5      // ...
6  }]);
7
8  // Implizit
9  module.controller('ImplicitController', function($scope, $http) {});
10
11 // $inject Property
12 const InjectController = function($scope, $http) {};
13 InjectController.$inject = ['$scope', '$http'];
14
15 module.controller('InjectController', InjectController);

```

Quelle: Eigene Darstellung

<sup>58</sup> Abgerufen am 28. Juli 2016 von  
[http://symfony.com/doc/current/components/dependency\\_injection/types.html](http://symfony.com/doc/current/components/dependency_injection/types.html)

Im Fall der impliziten Variante untersucht AngularJS die Parameternamen der Funktion. Der Injector-Service versucht diese Namen aufzulösen und die Abhängigkeiten bereitzustellen. Allerdings verursacht diese Methode Probleme bei der Komprimierung des Quelltextes (Minification). Um die Größe der auszuliefernden Skript-Dateien zu verringern, werden Minification-Tools eingesetzt. Unter anderem werden dabei die Parameternamen von Funktionen durch möglichst kurze Bezeichner ersetzt. Für diese zufällig generierten neuen Bezeichner findet das Dependency Injection-System von AngularJS keine Abhängigkeiten.

Daher sind die Varianten der Array-Deklaration oder des Inject-Property zu bevorzugen, wenn der Code durch Minification komprimiert wird. Als Behelfslösung können Tools wie *ng-annotate*<sup>59</sup> in ein Build-System eingebaut werden. Diese sorgen dafür, dass implizite Deklarationen automatisch durch die Inline-Array-Variante ersetzt werden.

Die neue Version von Angular behebt dieses Problem und stellt einen einheitlichen Weg für Dependency Injection bereit.

Listing 20: Dependency Injection in Angular

```

1 // dependency.service.ts
2 import { Injectable } from '@angular/core';
3
4 @Injectable()
5 export default class DependencyService {}
6
7 // app.component.ts
8 import { Component } from '@angular/core';
9 import { Http } from '@angular/http';
10 import DependencyService from './dependency.service';
11
12 @Component({
13   // ...
14 })
15 class AppComponent {
16   constructor(private http: Http, private dependencyService: DependencyService) {}
17 }
```

Quelle: Eigene Darstellung

<sup>59</sup> Siehe <https://github.com/olov/ng-annotate>

Die Kombination aus TypeScript und dem Einsatz der Decorator-Funktionen erlauben Angular die exakte Auflösung der Abhängigkeiten. Zur Laufzeit kann das Framework die Metadaten der Klasse abrufen. Aufgrund der Typdefinition im Konstruktor verursacht selbst Minification keine Probleme für das Dependency Injection-System. Eigene Klassen werden mit dem Injectable-Decorator markiert, um sie für Dependency Injection verfügbar zu machen.

### 3.2.10 Formulare

Über Formulare können Benutzereingaben an eine Anwendung geleitet werden. Im Normalfall werden die Eingabefelder ihrem Kontext entsprechend eingeschränkt und auf korrekte Eingaben überprüft. Für eine optimale Benutzererfahrung werden fehlerhafte Felder markiert und mit erklärenden Meldungen versehen.

Diese Komplexität kann durch den Einsatz eines Frameworks verringert werden, insbesondere dann, wenn das Formular noch während der Benutzereingaben kontinuierlich validiert werden soll.

Als Framework bietet Angular durch die bidirektionale Datenbindung einen einfachen Weg, Formulare zu implementieren. Darüber hinaus verfolgt es den Zustand der Felder und versieht die Elemente mit CSS-Klassen. Hierdurch wird die Hervorhebung der Felder, abhängig von ihrem Status, maßgeblich vereinfacht.

Tabelle 8: CSS-Klassen in einem Angular Formular

Status	<i>true</i>	<i>false</i>
Validierung	<code>ng-valid</code>	<code>ng-invalid</code>
Eingabe wurde verändert	<code>ng-pristine</code>	<code>ng-dirty</code>
Feld wurde fokussiert	<code>ng-touched</code>	<code>ng-untouched</code>
Asynchrone Validation	<code>ng-pending</code>	

Quelle: Eigene Darstellung

Die Palette der CSS-Klassen blieb in den verschiedenen Versionen von Angular unverändert. Ein Styling wird nicht vorgegeben und muss durch den Entwickler festgelegt werden. Grundsätzlich unterscheidet sich die Syntax insbesondere in Bezug auf die Eingabefelder.

Listing 21: Formular in AngularJS

```

1   <form name="form" ng-submit="$ctrl.onSubmit()" novalidate>
2     <input type="text" name="name" ng-model="user.name" minlength="3">
3     <div ng-hide="form.name.$valid || user.name.$untouched">Name ist zu kurz!</div>
4
5     <input type="email" name="email" ng-model="user.email">
6     <div ng-hide="form.email.$valid || user.email.$untouched">Ungültige Email!</div>
7
8     <button type="submit" ng-disabled="!form.$valid">Account erstellen</button>
9   </form>

```

Quelle: Eigene Darstellung

Das *novalidate*-Attribut wird benötigt, um die browsereigene Validation des Formulars zu unterbinden. Für die meisten Eingabetypen und Validierungs-Attribute existiert eine Implementation im Framework. In AngularJS wird das Absenden des Formulars durch die Direktive *ng-submit* abgefangen. Aufgrund des Event-Bindings fällt diese Direktive in der aktuellen Version weg.

Das neue Angular-Framework unterstützt zwei verschiedene Methoden zur Implementierung von Formularen:

- Template Driven
- Model Driven

Jeder Ansatz hat seine eigenen Vor- und Nachteile hinsichtlich Komplexität, Lesbarkeit und Testbarkeit.<sup>60</sup>

### **Template Driven**

Bei dieser Variante befindet sich die Logik zur Validierung des Formulars innerhalb des Templates. Eine Besonderheit ist die Definition der Template-Referenzvariablen an den Form- und Input-Elementen. Durch die vorgestellte Raute (# alternativ ref-) wird eine lokale Variable im Template erzeugt. Auf der linken Seite befindet sich der Bezeichner, während der rechte Teil ein DOM-Element oder eine Direktive referenziert. In diesem Fall wird eine Referenz auf die

---

<sup>60</sup> Vgl. Angular University (2016)

Direktiven *ngForm* und *ngModel* erzeugt. Die Instanz dieser Direktiven enthält den Status der Validierung, welcher für die Anzeige der Fehlermeldungen und der Deaktivierung des Absenden-Buttons genutzt wird.

Listing 22: Template Driven Form in Angular

```

1   <form #form="ngForm" (ngSubmit)="onSubmit()" novalidate>
2     <input type="text" #name="ngModel" [(ngModel)]="user.name" minlength="3">
3     <div [hidden]="name.valid || name.unouched">Name ist zu kurz!</div>
4
5     <input type="email" #email="ngModel" [(ngModel)]="user.email">
6     <div [hidden]="email.valid || email.unouched">Ungültige Email!</div>
7
8     <button type="submit" [disabled]!="form.valid">Account erstellen</button>
9   </form>
```

Quelle: Eigene Darstellung

Die Lesbarkeit dieser Methode sinkt mit der Größe des Formulars. Bei vielen Validatoren an einem Element wird die Definition zunehmend unübersichtlich. Bei geteilten Teams wird die Anpassung des Formulars für Webdesigner schwierig.<sup>61</sup>

## Model Driven

Diese Methode zieht sämtliche Validationslogik aus dem Template in die zugehörige Komponente. Auf diese Weise entsteht eine zentrale Stelle für die Definition der Regeln zur Validation. Dabei lassen sich die einzelnen Eingabefelder über die *FormControl*-Klasse oder durch die verkürzte Schreibweise mittels eines Array erstellen. Jegliche Validatoren und andere benötigte Klassen werden aus einem separaten Angular-Modul importiert. Die Funktionalität rund um Formulare gehört nicht zur Kernfunktionalität des Frameworks und wurde daher ausgelagert.

---

<sup>61</sup> Vgl. Angular University (2016)

Listing 23: Model Driven Form in Angular

```

1  // model-driven.html
2  <form [FormGroup] = "form" (ngSubmit) = "onSubmit()" novalidate>
3      <input type="text" formControlName="name">
4      <input type="text" formControlName="email">
5      <button type="submit" [disabled] = "!form.valid">Absenden</button>
6  </form>
7
8  // model-driven.component.ts
9  import { FormGroup, FormControl, Validators, FormBuilder } from '@angular/form';
10 import { REACTIVE_FORM_DIRECTIVES } from '@angular/form';
11 // ...
12
13 @Component({
14     directives: [REACTIVE_FORM_DIRECTIVES]
15     // ...
16 })
17 class ModelDrivenComponent {
18     form: FormGroup;
19     name = new FormControl('', Validators.minLength(3));
20
21     constructor(formBuilder: FormBuilder) {
22         this.form = formBuilder.group({
23             name: this.name,
24             email: ['', Validators.required]
25         });
26     }
27
28     onSubmit() {}
29 }
```

Quelle: Eigene Darstellung

Neben einer übersichtlichen Schreibweise im Template, ergibt sich ein elementarer Vorteil: Das Formular kann vom Template getrennt durch Unit-Tests überprüft werden. Dafür reicht es aus, die Klasse der Komponente zu instanziieren und mit den zu testenden Eingaben zu befüllen.<sup>62</sup>

---

<sup>62</sup> Vgl. Angular University (2016)

### 3.2.11 HTTP

Im Web ist HTTP(S) das zentrale Protokoll für die Kommunikation zwischen Browser und Server. Es existieren drei verschiedene Schnittstellen, um HTTP-Anfragen in JavaScript zu starten:

- XMLHttpRequest (XHR)
- JSONP
- Fetch

Letzteres ist eine im Vergleich zu XHR vereinfachte API. Aufgrund der neuen Spezifikation variiert die Browserunterstützung.<sup>63</sup>

Als Framework unterstützt Angular die Verwendung der ersten beiden Methoden. Für dynamische Webanwendungen und insbesondere Single-page-Applikationen ist das nachträgliche Laden von Daten ein elementarer Bestandteil.

In AngularJS wird der `$http`-Service genutzt, um HTTP-Anfragen auszulösen.

Listing 24: \$http Service in AngularJS

```

1  $http({
2      method: 'GET', // alternativ POST, PUT, DELETE, HEAD, PATCH oder JSONP
3      url: '/api/something'
4  }).then(function success(response) {
5      // ...
6  }, function error(response) {
7      // ...
8 });
9
10 // Shortcuts
11 $http.get(url, options);
12 $http.post(url, data, options);

```

Quelle: Eigene Darstellung

Ein Aufruf des Service gibt ein Promise-Objekt für die weitere Verarbeitung zurück. Bei Promises handelt es sich um ein Objekt, welches einen asynchronen Vorgang beschreibt.

---

<sup>63</sup> Abgerufen am 28. Juli 2016 von [https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)

Ein Promise-Objekt kann sich in drei verschiedenen Zuständen befinden:

- *pending* – Startzustand
- *fulfilled* – Operation erfolgreich
- *rejected* – Operation gescheitert

Darüber hinaus lassen sich durch Promises leicht mehrere asynchrone Vorgänge verketten oder parallelisieren.<sup>64</sup>

Um den HTTP-Client in Angular zu nutzen, muss dieser zuvor dem Dependency Injection-System bekannt gemacht werden. Aufgrund des modularen Aufbaus stellt Angular ohne Konfiguration lediglich die Kernfunktionalität des Frameworks bereit.

Listing 25: Registrierung der HTTP Service Provider in Angular

```

1 // main.ts
2 import { bootstrap } from '@angular/platform-browser-dynamic';
3 import { HTTP_PROVIDERS } from '@angular/http';
4 import { AppComponent } from './app.component';
5
6 bootstrap(AppComponent, [HTTP_PROVIDERS]);

```

Quelle: Eigene Darstellung

Eine *Best Practice* zur Einhaltung des Single Responsibility Principle ist die Auslagerung der HTTP-Logik in Service-Klassen. Komponenten sollten niemals direkt mit dem HTTP-Service von Angular kommunizieren müssen. Auf diese Weise lässt sich in Unit-Tests die Abhängigkeit leicht durch ein *Mock*-Objekt austauschen. Diese imitieren das Interface der echten Abhängigkeit, ohne dabei tatsächlich HTTP-Anfragen abzusenden.<sup>65</sup>

---

<sup>64</sup> Abgerufen am 29. Juli 2016 von  
[https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise)

<sup>65</sup> Vgl. Google (2016)

Listing 26: Verwendung des HTTP-Client in Angular

```

1  // app.component.ts
2  import { Component, OnInit } from '@angular/core';
3  import { HttpService } from './http.service';
4
5  @Component({ // ... })
6  export class AppComponent implements OnInit {
7      constructor(private httpService: HttpService) {}
8
9      ngOnInit() {
10         this.httpService.loadData()
11             .subscribe(data => console.log(data),
12                         error => console.log(error));
13     }
14 }
15
16 // http.service.ts
17 import { Injectable } from '@angular/core';
18 import { Http, Response } from '@angular/http';
19 import { Observable } from 'rxjs/Observable';
20
21 @Injectable()
22 export class HttpService {
23     constructor(private http: Http) {}
24
25     loadData() {
26         return this.http.get('/api/some/url')
27             .map((response: Response) => {
28                 const body = response.json();
29                 return body || {};
30             });
31     }
32 }
```

Quelle: Eigene Darstellung

In Listing 26 erhält der Service den HTTP-Client über *Constructor Injection*. Grundsätzlich ähnelt die API dem ursprünglichen \$http-Service aus AngularJS. Die entscheidende Neuerung ist die Einführung einer neuen Abhängigkeit von Angular. Während eine Verwendung von Promises weiterhin möglich ist, empfiehlt das Angular-Team *Observables* einzusetzen.

## Observables

„While promises may be more familiar, observables have many advantages. Don't rush to promises until you give observables a chance.“ – Angular Documentation<sup>66</sup>

*Reactive Extensions* ist eine Bibliothek zur Unterstützung der asynchronen und event-basierten Programmierung. Das Projekt wird von Microsoft und der Open-Source-Community entwickelt. Für jede große Programmiersprache existiert eine Implementierung der Bibliothek.

Dabei wird die zentrale Aufgabe der *Reactive Extensions* durch das Observable-Objekt abgebildet. Es erweitert das Beobachter-Entwurfsmuster (englisch: Observer), welches dazu dient, den registrierten Beobachtern mitzuteilen.<sup>67</sup> Mit Observables wird es möglich, eine Sequenz von Daten oder Events zu abonnieren. Darüber hinaus stellt die Bibliothek eine Reihe von Operatoren bereit, um diese Sequenzen zusammenzustellen und zu manipulieren. Ein Observable ist demnach eine asynchrone, endlose Datenquelle. Diese Quelle kann beliebig oft abonniert und das Abonnement anschließend ebenso beliebig oft wieder aufgehoben werden.

Listing 27: Observable mit Filterung der Ergebnismenge

```

1  const source = getAsyncDataAsObservable();
2
3  const subscription = source
4      .filter(data => data.value > 42)
5      .map(data => data.value)
6      .subscribe(
7          value => console.log('Value greater than 42: ' + value),
8          err => console.error(err)
9      );
10
11  subscription.dispose();

```

Quelle: Eigene Darstellung

Die Bibliothek erlaubt die flexible Verarbeitung von Daten unabhängig von ihrer Quelle. Observables lassen sich daher beliebig für Ajax-Anfragen, DOM-Events oder Web Sockets

---

<sup>66</sup> Abgerufen am 29. Juli 2106 von <https://angular.io/docs/ts/latest/guide/server-communication.html#!#promises>

<sup>67</sup> Gamma, Helm, Johnson, & Vlissides (1994), p. 293

einsetzen.<sup>68</sup> Die zur Verfügung gestellten Operatoren sind anhand bereits bestehender Array-Methoden abgeleitet. Auf diese Weise lässt sich die Ergebnismenge mit wenig Code filtern, aggregieren und manipulieren.

Tabelle 9: Matrix-Klassifikation von Werten

	<b>Single</b>	<b>Multiple</b>
<b>Synchronous / Pull</b>	Function	Enumerable
<b>Asynchronous / Push</b>	Promise	Observable

Quelle: RxJS Evolved<sup>69</sup>

In Tabelle 9 werden die verschiedenen Zugriffsarten auf Werte miteinander verglichen. Im einfachsten Fall handelt es sich dabei um den Aufruf einer Funktion. Dabei wird auf Anfrage ein einzelner Wert zurückgegeben. Für mehrere Werte muss ein aufzählbares Objekt verwendet werden, beispielsweise ein Array. Auf Funktionen und Aufzählungen wird synchron und aktiv zugegriffen.

Das asynchrone Gegenteil zu einer Funktion ist ein Promise-Objekt. Es stellt einen einmaligen asynchronen Vorgang dar. Dieser wird mit einem einzigen Wert zu einem späteren Zeitpunkt beendet.

Ein Observable hingegen kann beliebig viele Werte an seine Abonnenten senden. Ähnlich wie bei Promises wird die Ergebnismenge nach dem Hollywood-Prinzip zurückgegeben.

„*Don't call us, we'll call you.*“ – Martin Fowler<sup>70</sup>

Bei dem initialen Aufruf einer Promise oder eines Observable wird lediglich eine Callback-Funktion übergeben. Sobald der asynchrone Vorgang abgeschlossen ist, ruft das Objekt diese Funktion mit dem Ergebnis auf. Im Fall von Promises geschieht dies nur einmal.

---

<sup>68</sup> Vgl. Styoanov (2014)

<sup>69</sup> Vgl. Taylor (2015)

<sup>70</sup> Vgl. Fowler, Inversion of Control (2005)

### 3.2.12 Routing

Für eine Single-page-Applikation funktioniert das klassische Routing-Modell zwischen Server und Browser nicht mehr. Alle Seitenaufrufe werden clientseitig verarbeitet und erreichen den Server nicht. Lediglich die initiale Anfrage wird durch den Server beantwortet.

Über die History API des Browsers lässt sich der Verlauf mit JavaScript manipulieren. Darüber hinaus werden Events bereitgestellt, um Veränderungen oder Navigationsvorgänge abzufangen. Auf dieser Grundlage basieren clientseitige Router in JavaScript. In älteren Browerversversionen ohne Implementierung dieser Schnittstelle, kann eine Fallback-Methode angewendet werden. Durch das Einfügen einer Raute in die Adresse findet die Navigation ausschließlich lokal statt. Normalerweise wird diese Funktion genutzt, um Elemente in einer Seite zu referenzieren.<sup>71</sup>

Tabelle 10: URL-Arten im Browser

HTML5 Modus	Fallback
<code>http://example.org/page</code>	<code>http://example.org/#/page</code>

Quelle: Eigene Darstellung

Die Routing-Funktionalität ist ein eigenständiges Modul in AngularJS und muss separat eingebunden werden. Während der Startphase der Anwendung können die Routen definiert sowie mit ihrem entsprechenden Template und Controller verknüpft werden.

AngularJS vergleicht die aktuelle URL mit den konfigurierten Routen und sucht das zugehörige Template mit Controller. Über die `ng-view` Direktive wird die gefundene Route eingebunden. Allerdings kann diese Direktive nur einmal in jeder Anwendung vorkommen. Ein hierarchischer Aufbau der Routen ist ebenfalls nicht möglich.

---

<sup>71</sup> Abgerufen am 2. August 2016 von <https://developer.mozilla.org/de/docs/Web/API/History>

Listing 28: Routing-Konfiguration in AngularJS

```
1 // index.html
2 <div ng-view></div>
3
4 // app.js
5 angular.module('app', ['ngRoute'])
6 .config(function($routeProvider) {
7     $routeProvider.
8         .when('/', {
9             templateUrl: '/pages/home.html',
10            controller: 'HomeController'
11        })
12        .when('/login', {
13            templateUrl: '/pages/login.html',
14            controller: 'LoginController'
15        });
16    });

```

Quelle: Eigene Darstellung

Das Angular UI-Router-Projekt wird durch die Community entwickelt und versucht die Schwächen des normalen Routers zu beseitigen. Anstelle eines URL-gebundenen Ansatzes arbeitet der UI-Router wie ein endlicher Automat. Die verschiedenen Routen der Anwendung werden als Zustände definiert. Ein hierarchischer Aufbau dieser Zustände, kombiniert mit der Möglichkeit mehrere Ansichten anzusprechen, sorgt für eine größere Flexibilität.

Mit der neuen Router-Lösung aus Angular greift das Angular-Team die Wünsche und Kritikpunkte der Community auf. Aufgrund der Ausrichtung auf Komponenten genügt es in Zukunft, die Routen der Anwendung einer Komponente zuzuordnen. Daher stammt auch der neue Name des Moduls: *Angular Component Router*.

Listing 29: Konfiguration des Angular Component Router

```
1 // app.component.html
2 <router-outlet></router-outlet>
3
4 // main.ts
5 import { bootstrap } from '@angular/platform-browser-dynamic';
6 import { provideRouter, RouterConfig } from '@angular/router';
7 import { AppComponent } from './app.component';
8
9 const routes: RouterConfig = [
10   { path: '/', component: HomeComponent },
11   { path: '/login', component: LoginComponent }
12 ];
13
14 bootstrap(AppComponent, [
15   provideRouter(routes)
16 ]).catch(err => console.error(err));
```

Quelle: Eigene Darstellung

Die Konfiguration der Routen geschieht über eine simple Assoziation zwischen URL und Komponente in einem Objekt. Dieses Objekt wird der Angular-Anwendung während der Bootstrap-Phase übergeben. Anstelle von *ng-view* wird die *router-outlet*-Komponente eingesetzt, um die aktuelle Route in die Ansicht einzubinden.

Nach jeder erfolgreichen Navigation baut der Angular Component Router einen Baum der aktuell aktiven Routen auf. Dieser repräsentiert den gültigen Zustand des Routers. Jede Route kann ihre eigenen Kind-Routen besitzen. Dadurch lässt sich die Anwendung weiterhin leicht aus Komponenten zusammensetzen.<sup>72</sup>

---

<sup>72</sup> Vgl. Google (2016)

Listing 30: Hierarchischer Routen-Aufbau in Angular

```

1  const routes: RouterConfig = [
2    {
3      path: 'user',
4      component: UserComponent,
5      children: [
6        { path: ':id', component: UserDetailComponent },
7        { path: '', component: UserListComponent }
8      ]
9    }
10 ];

```

Quelle: Eigene Darstellung

In Listing 30 dient die User-Komponente als Grundgerüst für die folgenden Routen. Die beiden Kinder-Routen stellen eine Detail- und Listenansicht bereit. Dabei wird die URL auch hierarchisch anhand der übergeordneten Routen zusammengesetzt. Im Gegensatz zu AngularJS darf die Elternkomponente ein weiteres *router-outlet* besitzen. Darin wird die aktuelle Kind-Route angezeigt. Die resultierende Routen-Definition wird in Tabelle 11 gezeigt.

Tabelle 11: Ergebnis der Routen-Definition in Angular

URL	Komponente
/user	UserDetailComponent
/user/:id	UserListComponent

Quelle: Eigene Darstellung

Ein Doppelpunkt markiert Routen-Parameter als dynamische Platzhalter in einer URL. Auf den Wert dieses Parameters lässt sich über den Router-Service in der entsprechenden Komponente zugreifen.

### 3.2.13 Pipes

Eine häufige Anforderung an Anwendungen ist die Aufbereitung der darzustellenden Daten für den Benutzer. Auf diese Weise kann die Lesbarkeit verbessert und die Benutzerfreundlichkeit erhöht werden. Um diese Transformationslogik nicht wiederholen zu müssen, empfiehlt sich die Auslagerung in eine zentrale, separate Stelle.

In AngularJS übernehmen Filter diese Verantwortung. Mit der neuen Version wurden sie, angelehnt an ihre Syntax, zu Pipes umbenannt. Das Prinzip hinter den Pipes ähnelt dem einer normalen Funktion. Eingabeparameter werden transformiert und wieder ausgegeben. Allerdings sind Pipes für den Einsatz innerhalb von Templates gedacht.

Beide Framework-Versionen verfügen über eine Reihe eingebauter Filter beziehungsweise Pipes. Diese decken übliche Formatierungen ab, beispielsweise die Darstellung eines Datums oder einer Währungsformatierung.

Tabelle 12: Vergleich von Filter und Pipes

	<b>AngularJS - Filter</b>	<b>Angular - Pipe</b>
<b>currency</b>	<code>{{ price   currency }}</code>	<code>{{ price   currency:'EUR':true }}</code>
<b>date</b>	<code>{{ createdAt   date }}</code>	<code>{{ createdAt   date }}</code>
<b>filter</b>	<code>ng-repeat="item in items   filter:search"</code>	<i>nicht vorhanden</i>
<b>json</b>	<code>{{ data   json }}</code>	<code>{{ data   json }}</code>
<b>limitTo / slice</b>	<code>ng-repeat="item in items   limitTo:2:0"</code>	<code>*ngFor="let item in items   slice:0:2"</code>
<b>lowercase</b>	<code>{{ name   lowercase }}</code>	<code>{{ name   lowercase }}</code>
<b>uppercase</b>	<code>{{ name   uppercase }}</code>	<code>{{ name   uppercase }}</code>
<b>number</b>	<code>{{ percent   number:2 }}</code>	<code>{{ percent   number:1.2-2 }}</code>
<b>orderBy</b>	<code>ng-repeat="item in items   orderBy:'name'"</code>	<i>nicht vorhanden</i>

Quelle: Eigene Darstellung

In Angular wurden Pipes, welche die Filterung oder Sortierung von Listen betreffen, aus Performancegründen entfernt. Beim Einsatz dieser Filter in AngularJS hat der Entwickler keine Kontrolle darüber, wie oft diese Funktionen aufgerufen werden. Es handelt sich dabei um kostspielige Operationen, welche die Geschwindigkeit stark beeinträchtigen. Stattdessen sollte diese Funktionalität in die entsprechende Komponente ausgelagert werden. An dieser Stelle lässt sich das Aufrufen der Filter- oder Sortierungslogik besser regulieren.

Filter werden in AngularJS als Factory-Funktion registriert. Diese Factory gibt die Filter-Funktion zurück, sobald sie vom Framework aufgerufen wird. Der erste Parameter des Filters ist der Eingabewert, gefolgt von weiteren Optionen.

Listing 31: Filter-Definition in AngularJS

```
1 angular.module('app')
2     .filter('reverse', function() {
3         return function(input) {
4             return input.split('').reverse().join('');
5         };
6     });

```

Quelle: Eigene Darstellung

Im Gegensatz dazu nutzt Angular zur Umsetzung von Pipes Decorator, Klassen und Interfaces. Als Äquivalent zur Filter-Funktion dient die Transform-Methode einer Pipe. Unter Verwendung von TypeScript lassen sich sämtliche Ein- und Ausgabeparameter typisieren. Für neue Entwickler ist die Semantik mit dieser Schreibweise schneller ersichtlich.

Listing 32: Pipe-Definition in Angular

```
1 import { Pipe, PipeTransform } from '@angular/core';
2
3 @Pipe({name: 'reverse'})
4 export class ReversePipe implements PipeTransform {
5     transform(input: string): string {
6         return input.split('').reverse().join('');
7     }
8 }
```

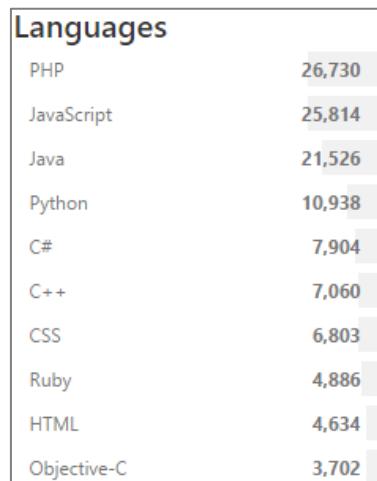
Quelle: Eigene Darstellung

### 3.3 Alternative Frameworks

Die Umgebung der JavaScript-Frameworks wächst mit rasanter Geschwindigkeit. Insbesondere durch Node.js wurde der Einsatz von JavaScript auf dem Server ermöglicht und die Popularität weiter gesteigert. Dabei entwickeln sich sowohl Frontend als auch Backend stetig weiter.

Abbildung 13 zeigt die Anzahl der Frameworks in den verschiedenen Programmiersprachen auf GitHub. JavaScript befindet sich mit einer Differenz von 1.000 Projekten relativ knapp hinter PHP. In Zukunft könnte es PHP sogar überholen.

Abbildung 13: Anzahl der Frameworks nach Programmiersprache



Quelle: GitHub<sup>73</sup>

Es existiert keine allgemeine Framework-Lösung, die alle Probleme der Webentwicklung abdeckt. Daher werden bestehende Ansätze kontinuierlich überdacht und überarbeitet. Das junge Alter der Webentwicklung und die stetige Weiterentwicklung des Webs an sich tragen dazu bei, dass ehemals etablierte Lösungen schnell an Aktualität verlieren.

In den letzten Jahren wurden immer mehr JavaScript-Frameworks veröffentlicht. Dabei bezogen sich viele dieser Projekte auf die gleiche Problemstellung. Ein Großteil dieser Frameworks erfordert eine Menge Konfiguration und eine Auswahl unterstützender Tools und Bibliotheken. Mittlerweile hat sich daher der Begriff „*JavaScript Fatigue*“<sup>74</sup> etabliert.

Nichtsdestotrotz gibt es einige Projekte, die seit längerer Zeit bestehen oder eine große, unterstützende Community bilden konnten. In den folgenden Abschnitten werden die unterschiedlichen Ansätze alternativer Frameworks in Bezug auf Angular untersucht und verglichen.

---

<sup>73</sup> Abgerufen am 2. August 2016 von <https://github.com/search?q=framework+language%3Ajavascript>

<sup>74</sup> Vgl. Clemons (2015)

#### 3.3.1 Ember

„A framework for creating ambitious web applications“ – Ember.js<sup>75</sup>

In 2011 wurde das Ember-Projekt durch Yehuda Katz gestartet. Der Gründer ist gleichzeitig Mitglied der Entwicklerteams von *jQuery* und *Ruby on Rails*. Daher stammt ein Großteil der Inspiration von *Ruby on Rails*.

Das Projekt wird einzig und allein durch seine Open-Source-Community entwickelt. Es wird bis heute von diversen Firmen finanziell und technisch gesponsert. Allerdings üben die Sponsoren keine direkte Entscheidungsgewalt auf das Projekt aus.

Zu den direkten Abhängigkeiten gehören die Bibliotheken *jQuery* und *Handlebars*. Letztere wird für das Templating und die Datenbindung des Frameworks eingesetzt. Das Framework selbst orientiert sich stark an dem MVC-Prinzip und bietet dadurch Entwicklern aus diesem Bereich einen einfachen Einstieg.

Eine zentrale Richtlinie der Ember-Community ist „*Convention over configuration*“. Nach diesem Prinzip wird die Komplexität eines Projektes durch Einhaltung vordefinierter Konventionen reduziert. Für ein Ember-Projekt bedeutet dies eine deutlich erhöhte Entwicklungsgeschwindigkeit, sofern die durch Ember definierten Vorgaben eingehalten werden. Die häufigsten Aufgaben einer Webanwendung werden hiermit dem Entwickler abgenommen und durch das Framework bereitgestellt.<sup>76</sup>

Darüber hinaus bietet das Projekt eine Kommandozeile, um den Entwickler bei der Entwicklung eines Ember-Projekts zu unterstützen. Unter anderem hilft das Tool bei der Generierung neuer Komponenten der Anwendung sowie bei der Ausführung des Build-Prozesses.

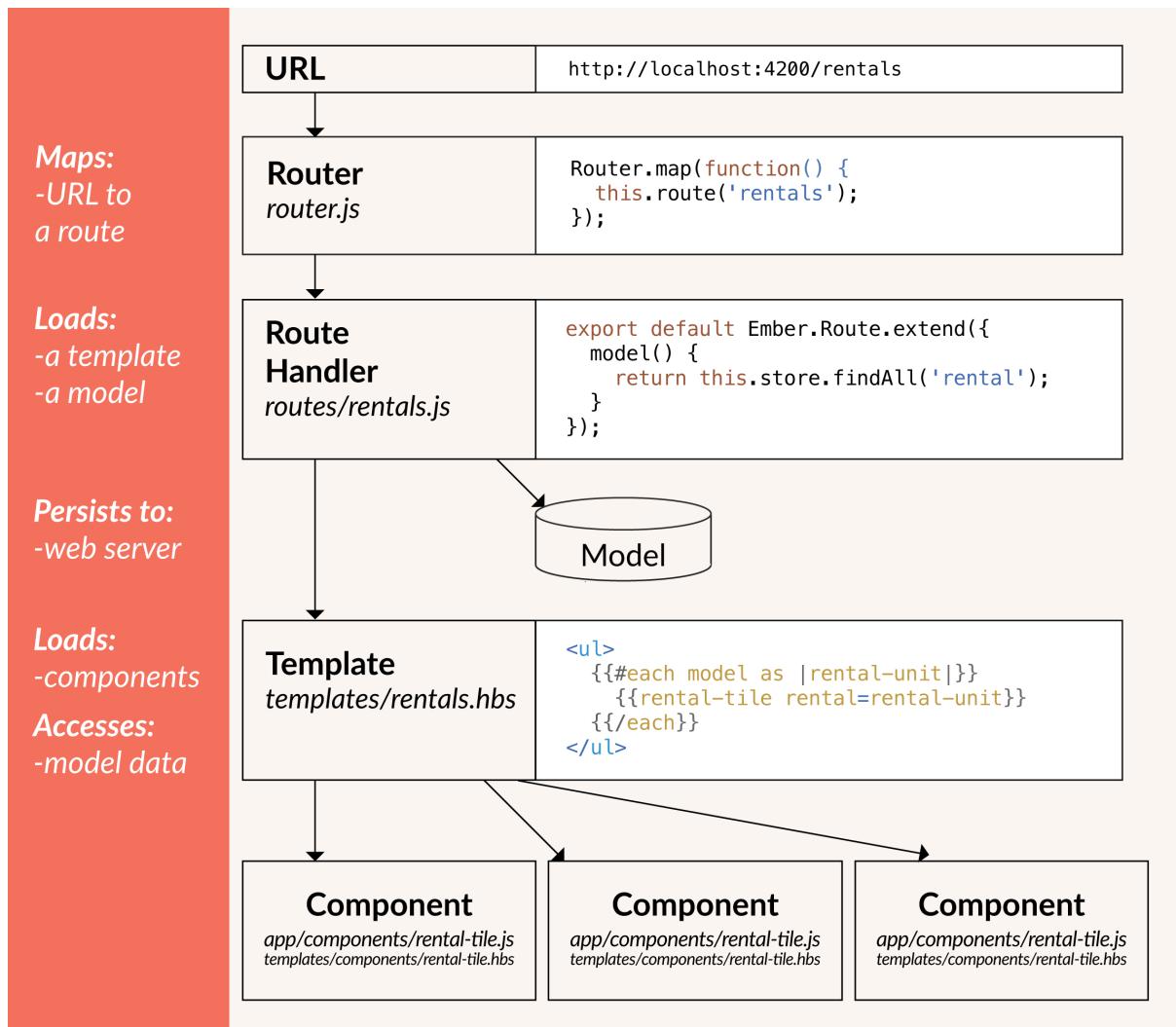
Allgemein besteht das Framework aus den folgenden Komponenten: *Route*, *Model*, *Template* und *Component*. In Ember ist ein *Controller* lediglich eine spezielle Form eines *Component* und ist daher in Abbildung 14 nicht aufgelistet.

---

<sup>75</sup> Abgerufen am 3. August 2016 von <http://emberjs.com/>

<sup>76</sup> Vgl. Hannah (2015)

Abbildung 14: Ember.js Architektur Übersicht



Quelle: Ember.js – Core Concepts<sup>77</sup>

Ein Nachteil von Ember ist die fehlende Flexibilität aufgrund der einzuuhaltenden Konventionen. Es ist per Definition ein allumfassendes Framework und sollte daher im richtigen Kontext eingesetzt werden. Wird keine flexible, aber eine schnelle und robuste Lösung benötigt, zeigen sich die Stärken des Frameworks.

<sup>77</sup> Abgerufen am 3. August 2016 von <https://guides.emberjs.com/v2.7.0/getting-started/core-concepts/>

#### 3.3.2 React

„A JavaScript library for building User Interfaces“ – React<sup>78</sup>

Das React-Projekt wird seit 2013 von Facebook entwickelt und dient dem Bau komponentenbasierter Anwendungen im Web. In erster Linie ist React eher eine einfache Bibliothek und kein vollständiges Framework. Es konzentriert sich lediglich auf den View-Teil des MVC-Prinzips.

Im Vergleich zu anderen Lösungen nutzt React einen radikal neuen Ansatz. Mithilfe der optionalen *JavaScript Syntax Extension* (JSX) vermischt React JavaScript und HTML. Daher kann die volle JavaScript-Funktionalität innerhalb von React-Komponenten genutzt werden. Das Lernen einer domänenspezifischen Sprache, um Logik in HTML zu ermöglichen, ist nicht notwendig.

Darüber hinaus erreicht die Bibliothek eine vergleichsweise hohe Performance durch den Einsatz eines *Virtual DOM* und *Synthetic Events*. Diese Techniken abstrahieren die browsereigene Implementierung des DOM, um eine höhere Geschwindigkeit zu erreichen.

Ein zentraler Vorteil der Bibliothek ist die Unterteilung einer Anwendung in möglichst kleine und wiederverwendbare Komponenten. Von sich selbst aus bietet React lediglich das Komponentensystem mit dem zugehörigen Rendering. Zusätzliche Funktionalität muss durch das Einbinden anderer Bibliotheken ergänzt werden. Dabei schränkt React Entwickler bei der Wahl weiterer Bibliotheken nicht ein. Diese Flexibilität ist vor allem für große und komplexe Projekte wertvoll. Allerdings entsteht dadurch für die Entwickler ein höherer Aufwand bei der Zusammenstellung eines vollwertigen Frameworks.

Trotz des relativ jungen Alters bringt React eine große und aktive Community mit sich. Aus der Vielzahl der veröffentlichten Bibliotheken für React haben sich bereits Grundsteine für die meisten Anwendungen gebildet. Mit Redux wurde das durch Facebook vorgestellte Architekturprinzip Flux weiterentwickelt und verbessert. Die Verwaltung des Zustands und der unidirektionale Datenfluss dieses Architekturstils sind nicht auf React beschränkt, eignen sich

---

<sup>78</sup> Abgerufen am 3. August 2016 von <https://facebook.github.io/react/>

aber hervorragend für die Entwicklung komponentenbasierter Applikationen. Des Weiteren hat Facebook mit React Native eine Plattform geschaffen, um die Entwicklung mobiler Anwendungen zu vereinfachen.

#### 3.3.3 Aurelia

*„A JavaScript client framework for mobile, desktop and web leveraging simple conventions and empowering creativity“ – Aurelia<sup>79</sup>*

Aurelia nutzt neuste Webtechnologien, um die Entwicklung moderner und zukunftsfähiger Anwendungen zu ermöglichen. Es ist der Nachfolger von Durandal, welches als Single-page Application Framework mit AngularJS konkurriert. Hinter beiden Frameworks steht die Firma Durandal Inc mit Hauptentwickler Rob Eisenberg, der bereits Teil des Angular-Teams war. Die erste Entwickler-Version von Aurelia wurde 2015 veröffentlicht, während das erste offizielle Release am 27. Juli 2016 freigegeben wurde.

Im Gegensatz zu Angular legt Aurelia eine hohe Priorität auf die Einhaltung bestehender Standards und die Nutzung der Webplattform. Darüber hinaus versucht das Framework sich soweit wie möglich im Hintergrund zu halten. Komponenten werden durch einfache JavaScript-Klassen, ohne Framework-spezifischen Code, abgebildet. So müssen Entwickler weniger Eigenheiten lernen, und die Wiederverwendbarkeit des Codes bleibt erhalten.

Konventionen spielen, ähnlich wie bei Ember, eine wichtige Rolle bei Aurelia. Allerdings ist es möglich, alle Konventionen explizit zu überschreiben. Dadurch wird weiterhin Flexibilität gewährleistet, und das Framework lässt sich leichter in verschiedenen Szenarien einsetzen.

Des Weiteren müssen Komponenten nicht manuell beim Framework registriert werden. Aufgrund der Konventionen findet Aurelia diese automatisch. Ein zusätzlicher von den Entwicklern geschriebener Code ist nicht erforderlich. Eine Komponente kommt demnach vollständig ohne Framework aus. Das zugehörige Template enthält nur die nötigen Anweisungen zur Datenbindung.

---

<sup>79</sup> Abgerufen am 3. August 2016 von <http://aurelia.io/>

### 3.3.4 Vue

„Reactive Components for Modern Web Interfaces“ – Vue.js<sup>80</sup>

Vue konzentriert sich wie React einzig und allein auf die View-Schicht. Es versucht dabei, mithilfe einer einfachen, intuitiv verwendbaren API die bestmögliche Entwicklererfahrung zu bieten. Die erste Version der Bibliothek wurde 2014 veröffentlicht. Hauptentwickler des Projekts ist Evan You, welcher bereits bei Google gearbeitet hat und Teil des Meteor Framework-Teams war. Mittlerweile steht eine kontinuierlich wachsende Community hinter dem Projekt.

Die Bibliothek soll sowohl für kleine als auch für komplexe Projekte geeignet sein. Von sich aus bietet Vue lediglich ein Komponentensystem mit zugehöriger Datenbindung. Weitere Funktionalitäten, welche in größeren Projekten benötigt werden, können durch beliebige Bibliotheken ergänzt werden. Allerdings bietet Vue auch eigene Lösungen an, beispielsweise zur Umsetzung des Routings einer Single-page Application.

Viele Ideen anderer Frameworks finden sich vereinfacht in Vue wieder. Das Projekt vereinigt die besten Ideen moderner Webentwicklung und fasst sie in einer Bibliothek zusammen.

Im Gegensatz zu React setzt Vue auf eine leichtgewichtige, domänen spezifische Sprache innerhalb der Templates. Allerdings unterstützt die Bibliothek *Single file components*, um das Template, die Logik und das Styling einer Komponente innerhalb einer Datei zu halten. Für Entwickler ergibt sich dadurch während der Arbeit an einer Komponente ein kleinerer Kontextwechsel.

Zurzeit befindet sich die zweite Version von Vue, eine vollständige Neuentwicklung, in der Beta-Phase. Das Team versucht, die Änderungen an der API im Vergleich zum Vorgänger möglichst gering zu halten. Es soll lediglich einfacher und intuitiver sein. Darüber hinaus setzt das Projekt in Zukunft ebenfalls auf einen Virtual-DOM-Ansatz. Allerdings soll es möglich sein, frei zwischen Templates und Render-Logik zu wählen.

---

<sup>80</sup> Abgerufen am 3. August 2016 von <http://vuejs.org/>

### 3.4 Vergleich der Frameworks

Im folgenden Abschnitt sollen die JavaScript-Frameworks hinsichtlich ihres Einsatzzweckes miteinander verglichen werden. Darüber hinaus wird untersucht, inwieweit die Migration einer AngularJS-Anwendung realisierbar ist.

Jedes der hier vorgestellten Frameworks ist für einen spezifischen Anwendungsfall konzipiert. Eine positive oder negative Bewertung ist demnach nicht sinnvoll. Die in Tabelle 13 verwendeten Zahlen sollen daher einen Aufschluss über die Aktivität und Dimension der Projekte geben.

In die Kategorie der Bibliotheken fallen React und Vue. Im Gegensatz zu einem vollwertigen Framework sind diese primär nicht dafür gedacht, die volle Funktionalität einer Webanwendung abzudecken. Es bietet sich die Möglichkeit, weitere Funktionen durch beliebige Bibliotheken zu ergänzen. Der entscheidende Vorteil ist die Freiheit, Teilbereiche der Anwendung auszutauschen und mit neuen oder besser geeigneten Lösungen umzusetzen. Bei monolithischen Frameworks ist dieser Schritt schwierig oder nicht umsetzbar. Allerdings haben die Entwickler des Projekts durch die zu treffende Auswahl der eingesetzten Bibliotheken eine größere Verantwortung.

Es existieren keine vordefinierten Migrationswege, um die Funktionalität von AngularJS mit React oder Vue abzubilden. Allerdings besteht die Möglichkeit, einzelne Komponenten einer Applikation mithilfe der genannten Bibliotheken zu implementieren. Insbesondere performancekritische Teile können mit diesen Projekten im Vergleich zu AngularJS wesentlich effizienter umgesetzt werden. Eine vollständige Migration auf diese Bibliotheken würde dagegen eine vollständige Neuentwicklung der zugrundeliegenden Anwendung erfordern, da sich die Ansätze der Projekte zu sehr voneinander unterscheiden und die Migration im laufenden Betrieb unwirtschaftlich wäre.

Als ältestes Framework deckt Ember alle Bereiche einer Webanwendung ab. Aufgrund der Konventionen kann eine hohe Entwicklungsgeschwindigkeit erreicht werden. Es gilt dabei zu bedenken, dass die Flexibilität zugunsten von Ember aufgegeben wird. Eine progressive Migration von AngularJS zu Ember ist undenkbar, da die beiden Frameworks nicht parallel

betrieben werden können. Des Weiteren erfordern die spezifischen Konventionen eine vollständige Umstrukturierung des Projekts. Nichtsdestotrotz ist eine Portierung der Anwendung aufgrund ähnlicher Konzepte grundsätzlich möglich. Vor allem Direktiven, Templates und Controller lassen sich umschreiben. Allerdings müsste die eigentliche Entwicklung am Projekt für den Zeitraum der Portierung unterbrochen werden.

Ähnliches gilt für Aurelia, wobei das Framework viele Prinzipien aus AngularJS aufgreift und vereinfacht. Der Einsatz ist vor allem für zukunftsorientierte Projekte sinnvoll. Es werden neuste Webtechnologien eingesetzt und Standards eingehalten. Aufgrund der automatischen Registrierung von Klassen wird wenig Framework-spezifischer Code benötigt. Diese Eigenschaft reduziert die Bindung an Aurelia und hält das Projekt flexibel. Eine gleichzeitige Ausführung von Aurelia und AngularJS ist nicht möglich.

Für eine effiziente, fortlaufende Migration einer AngularJS-Anwendung bietet sich der Nachfolger Angular am ehesten an. Es existiert ein Upgrade-Adapter, der die parallele Ausführung beider Frameworks ermöglicht. Darüber hinaus sorgt die Community mit Styleguides und Anleitungen für eine möglichst reibungslose Migration. Durch die Einhaltung von Styleguides kann eine bevorstehende Portierung vorbereitet werden.

Eine Migration von AngularJS auf ein völlig neues Framework kann, wirtschaftlich betrachtet, nur mit Unterbrechung der eigentlichen Entwicklung durchgeführt werden. Im Fall eines bereits veröffentlichten Produkts ist dies zumeist nicht tragbar. Falls ein Team Wert darauf legt, möglichst viel Wissen von AngularJS mitzunehmen, empfiehlt sich der Wechsel zu Aurelia. Das Framework greift viele Prinzipien auf und fordert aktiv den leichten Einstieg. Für die Entwicklung isomorphischer Anwendungen zur Ausführung auf dem Server und im Browser bietet sich React an. Die Bibliothek ist agnostisch gegenüber der Render-Umgebung. Daher lassen sich auch mobile Apps leicht entwickeln.

Tabelle 13: Vergleich der JavaScript-Frameworks

	<b>Angular</b>	<b>Ember</b>	<b>React</b>	<b>Aurelia</b>	<b>Vue</b>
<b>Entwickler<sup>1</sup></b>	Google	Community	Facebook	Rob Eisenberg Durandal	Evan You Community
<b>Kategorie</b>	Framework	Framework	Bibliothek	Framework	Bibliothek
<b>Erscheinungsjahr</b>	2014	2011	2013	2015	2014
<b>Version</b>	2.0.0-rc.4	2.9.0	15.3.0 16.0.0-alpha	1.0.1	1.0.26 2.0.0-beta.6
<b>Mitwirkende</b>	311	604	752	43	81
<b>GitHub Stars</b>	14.532	16.591	46.712	7.073	23.834
<b>GitHub Issues</b>	1.155	188	519	26	25
<b>Open / Closed</b>	5.149	4.527	3.039	389	2.562
<b>YouTube-Videos</b>	~406.000	~27.000	~89.000	~6.750	~15.500
<b>Stackoverflow</b>	13.591	19.442	19.941	1.062	1.527
<b>Development</b>	766 KB	1,69 MB	1,1 MB	781 KB	266 KB
<b>Production<sup>2</sup></b>	484 KB	419 KB	133 KB	389 KB	75 KB

Stand: 4. August 2016

<sup>1</sup> Alle Projekte sind Open Source und werden durch die Community mitentwickelt.<sup>2</sup> Größe der Projekte in Kilobyte (KB) oder Megabyte (MB). Production bezieht sich auf die Größe des Scripts nach *Minification*.

Quelle: Eigene Darstellung

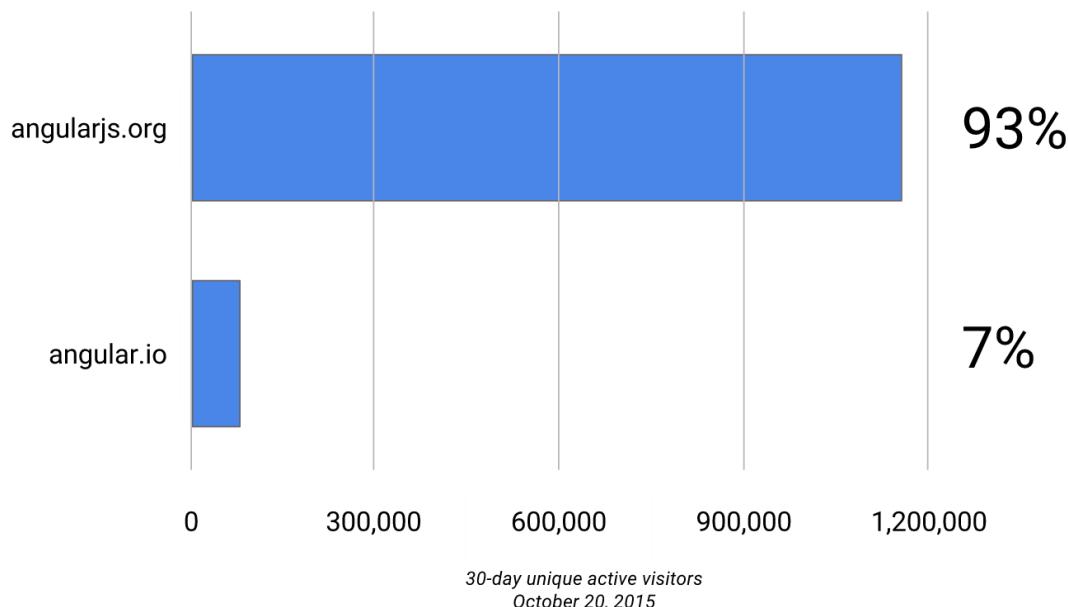
## 4. Migration der Webapplikation

Die Webanwendung von SMARTCRM wurde auf Basis des Ionic-Frameworks mit AngularJS entwickelt. Um die Anwendung zukunftsorientiert zu halten, gilt es, die Entwicklung des Webs zu beobachten und zum richtigen Zeitpunkt neue Technologien aufzugreifen, sofern diese einen Mehrwert für das Produkt bieten.

Durch die Veröffentlichung von Angular rückt das alte Framework AngularJS in die Support-Phase. Es werden weiterhin Sicherheitsupdates und neue Funktionalität zur Vereinfachung der Migration veröffentlicht. Allerdings richtet sich der Fokus der Weiterentwicklung auf das neue Framework. Laut Aussage von Google und dem Team hinter Angular soll die erste Version des Frameworks unterstützt werden, so lange die Community Interesse am Projekt zeigt.<sup>81</sup> Als Metriken dienen dafür die Aufrufe der AngularJS- und Angular-Seiten sowie die Aktivität der GitHub Repositories.

Abbildung 15: Aufrufe der AngularJS- und Angular-Seiten

## Angular 1 vs Angular 2



Quelle: AngularConnect Keynote<sup>82</sup>

<sup>81</sup> Vgl. Green & Minar, ng-conf 2015 Day 1 Keynote (2015)

<sup>82</sup> Abgerufen am 5. August 2016 von <https://goo.gl/UwuKFG>

Demnach wird Google den Support von AngularJS erst einstellen, wenn ein Großteil der Community auf den Nachfolger umgestiegen ist. Nichtsdestotrotz wird eine Migration in einigen Jahren unausweichlich sein.

In den nachfolgenden Kapiteln wird das Projekt hinsichtlich der Migration auf Angular untersucht. Als Grundlage dient im ersten Schritt die Analyse des Projektaufbaus und der Abhängigkeiten der Applikation. Daraufhin folgen die nötigen Schritte zur Durchführung einer Migration. Im letzten Teil werden verschiedene Strategien zur Vorgehensweise erörtert und schließlich die optimale Strategie ausgewählt.

### 4.1 Projektanalyse

Für die erfolgreiche Umstellung eines komplexen Software-Projekts ist die Analyse der Codebasis elementar. Ohne eine sorgfältige und vorsichtige Planung besteht ein hohes Risiko, die Effizienz der eigentlichen Migration zu mindern.

Die Struktur des Projekts befolgt übliche Konventionen der Webentwicklung im Bereich JavaScript und Node.js. Auf der obersten Ebene befinden sich Konfigurationsdateien für Transpiler, Package-Manager, Build-Systeme und Tests. Die Deklaration der Abhängigkeiten ist zwischen Bower und dem Node Package Manager (npm) getrennt. Dabei werden Frontend-Abhängigkeiten wie Ionic und Angular durch Bower bedient. Durch npm werden sämtliche Tools abgedeckt, um Builds durchzuführen und das Projekt zu testen.

Abbildung 16: Projektstruktur von SMARTCRM.Web

* .babelrc	* makefile	► app
♫ .bowerrc	{ } package.json	► assets
* .editorconfig	* Procfile	► hooks
❖ .gitignore	JS protractor.base.conf.js	► mocks
{ } .io-config.json	JS protractor.desktop.conf.js	► output
* .jshintrc	JS protractor.mobile.conf.js	► resources
JS app.js	MD README.md	► scss
♫ bower.json	JS test-main.js	► tasks
config.xml	{ } version.json	► tests
JS Gruntfile.js	* webAppStart.bat	► tmp
JS Gruntfile.test.js	JS webpack.config.js	► webstorm
* ionic.project	JS webpack-prod.config.js	► www
JS karma.conf.js	JS webpack-test.config.js	

Quelle: Eigene Darstellung

Als Build-System kommt eine Kombination aus Grunt und Webpack zum Einsatz. Grunt sorgt als Task-Runner für die Kompilierung der HTML- und CSS-Dateien. Dieser Schritt ist aufgrund der Syntaxerweiterungen – Jade für HTML und Syntactically Awesome Stylesheets (Sass) für CSS – notwendig. Webpack fasst als Bundling-Tool zusammengehörige Ressourcen zu Modulen zusammen. In diesem Fall wird es primär dafür verwendet, ECMAScript 6 unter Einsatz des Transpilers Babel zu browserkompatiblem JavaScript zu kompilieren.

Der eigentliche Quelltext der Anwendung befindet sich innerhalb des App-Ordners. Darin enthalten sind JavaScript-Module, Jade-Templates und Sass-Stylesheets. Zugehörige Unit- und End-to-End-Tests befinden sich in einem separaten Tests-Ordner.

Von einer Migration sind nicht alle Teile des Projekts betroffen. Relevant sind die Ordner *app* und *tests* sowie eine Reihe von Konfigurationsdateien. Innerhalb des App-Ordners müssen ausschließlich JavaScript- und Jade-Dateien beachtet werden. Aufgrund der Syntaxänderungen in Angular werden HTML-Templates direkt durch eine Portierung beeinflusst. Die natürliche Abhängigkeit der Tests zu ihren Modulen erfordert auch an dieser Stelle Änderungen. In Tabelle 14 sind die für die Migration relevanten Teile des Projekts aufgelistet.

Tabelle 14: Analyse des Projekts in Bezug auf die Migration

<b>Ordner</b>	<b>Dateityp</b>	<b>Anzahl Dateien</b>	<b>Gesamtanteil</b>	<b>Codezeilen</b>	<b>Anteil an Migration</b>
<b>Projekt</b>		2.360	100%		
<i>Davon betroffen durch die Migration</i>					
<i>app</i>		316	~ 13,39%		
	.jade	62	~ 2,63%	1.430	~ 19,81%
	.js	205	~ 8,69%	14.684	~ 65,45%
	.json	2	~ 0,08%		
	.scss	47	~ 1,99%		
<i>tests</i>		35	~ 1,48%		
	.js	33	~ 1,40%	4.841	~ 10,54%
	.jshintrc	2	~ 0,08%		
<b>Konfiguration</b>		13	~ 0,55%		~ 4,15%
<b>Summe</b>		313	~ 13,26%	20.955	100%
<i>Grau hinterlegte Zeilen sind lediglich Inhalte eines Ordners, aber nicht durch die Migration betroffen.</i>					

Quelle: Eigene Darstellung

## 4.2 Vorbereitung

Ein großer Teil der Arbeit kann bereits während der Vorbereitungsphase umgesetzt werden. Dadurch wird die Dauer der Migrationsphase verkürzt, welche den operativen Betrieb einschränken kann. Alle Vorbereitungen sind größtenteils unabhängig von Angular und können daher jederzeit begonnen werden.<sup>83</sup> Darüber hinaus kann die Vorbereitung bereits eine

---

<sup>83</sup> Vgl. Precht (2015)

klarere Struktur der AngularJS-Anwendung und damit eine gute Grundlage für ein Entwickler-Team schaffen.

### 4.2.1 Projektstruktur

Der Aufbau der Anwendung sollte modular gehalten werden. Jede Datei enthält dabei lediglich eine Klasse oder Komponente. Hilfreich ist hierbei die Einhaltung des Single Responsibility Principle, nach dem jede Klasse nur eine Zuständigkeit und damit nur einen Grund haben sollte, sich zu ändern.<sup>84</sup>

Modularität ist eine entscheidende Eigenschaft zur Entwicklung der Applikation mit einem komponentenbasierten Aufbau. Dies erleichtert die Umstellung der AngularJS-Anwendung auf Komponenten. Darüber hinaus können sich Entwickler in einem modularen Projekt leichter zurechtfinden. Denn da sich alle Komponenten eines Moduls nah beieinander befinden, werden weniger Kontextwechsel benötigt.

Die aktuelle Ordnerstruktur teilt das Projekt nach Kategorien auf. Daher existieren für Direktiven, Services und Filter jeweils eigene Ordner. Es ergibt sich folgende vereinfachte Struktur:

- app/
  - directives/
  - filters/
  - lib/
    - states/
    - ...
  - mainViews/
  - services/
  - ...

Durch diese Struktur werden zusammengehörige Dateien über mehrere Ordner verteilt, da sie nach ihrer Kategorie unterteilt sind. Eine schrittweise Migration nach Komponenten wird dadurch erschwert.

---

<sup>84</sup> Vgl. Martin, The Single Responsibility Principle (2014)

Es ist nicht sofort ersichtlich, welche Dateien zu einem bestimmten Feature oder einer Komponente gehören.

Ein alternativer Ansatz ist die Aufteilung nach Komponenten:

- app/
  - components/
    - calendar/
    - dashboard/
      - dashboard.controller.js
      - dashboard.service.js
      - dashboard.spec.js
      - dashboard.jade
      - dashboard.scss
      - ...
    - detail/
    - list/
    - ...
  - common/
    - menubar/
    - sidebar/
    - ...

Bei dieser Aufteilung befinden sich alle Dateien eines Features innerhalb eines gemeinsamen Ordners. Sogar die Tests werden an der gleichen Stelle abgelegt. Im Fall einer Migration lässt sich die Komponente gezielt umstellen. Neuen Entwicklern reicht ein Blick in die Projektstruktur, um eine bestimmte Funktionalität zu finden.<sup>85</sup>

### 4.2.2 Services

In Angular sind Services einfache JavaScript-Klassen. Eine Unterscheidung zwischen Factory, Service, Value, Constant und Provider wird nicht mehr vorgenommen. Anstelle von Factory sollte daher die Service-Funktion in AngularJS verwendet werden. Unter Verwendung von

---

<sup>85</sup> Vgl. Motto (2016)

ECMAScript 6 oder TypeScript können Services auch in AngularJS bereits als Klassen geschrieben werden. Hierdurch lässt sich der Aufwand, der Migration von Services auf ein Minimum reduzieren.

### 4.2.3 Komponenten

Mit AngularJS 1.5 wurde als Alternative zur klassischen Registrierung von Direktiven eine neue Component-Funktion eingeführt. Diese dient lediglich als Hilfsfunktion zur Erstellung einer Direktive mit bestimmten Standards und Einschränkungen. Die Standardwerte gelten für diese Version von AngularJS als Best Practice, um einen einfachen und einheitlichen Aufbau der Anwendung zu gewährleisten.

Durch diese Neuerung wird die Unterteilung einer Applikation in einzelne Komponenten ermöglicht. Unter Verwendung von Komponenten kann der Stil von Angular imitiert werden und die Migration der Komponenten lässt sich leichter bewerkstelligen.

Zu den wichtigsten Eigenschaften von Komponenten gehört die exklusive Einschränkung auf Elemente. Darüber hinaus wird in jedem Fall ein isolierter Scope verwendet, eine Vererbung wie bei Direktiven ist nicht möglich. Alle Bindings einer Komponente müssen explizit deklariert sein und werden direkt an den Controller gebunden. Der Controller wird im Template im Standardfall als `$ctrl` verfügbar gemacht. Dieser Wert lässt sich überschreiben, allerdings wird die Verwendung der `controllerAs`-Option implizit vorgeschrieben. Auf diesem Weg wird das `$scope`-Objekt nicht weiterverwendet, da sich alle Werte am Controller befinden. In Angular existiert dieses Objekt nicht mehr, daher sollte während der Vorbereitungsphase alle Codezeilen mit `$scope` umgeschrieben werden. Darüber hinaus sollte die `ng-controller` Direktive aus AngularJS nicht weiter verwendet werden. Diese existiert in Angular nicht mehr, stattdessen sollten Controller und Template immer durch eine Komponente aneinander gebunden werden. Daher lässt sich diese Direktive durch den Einsatz von Komponenten vermeiden.

Listing 33: Component-Hilfsfunktion in AngularJS 1.5

```
1   .component('message', {  
2     bindings: {  
3       text: '<' // One-way Binding ab AngularJS 1.5  
4     },  
5     controller: function MessageController() {  
6       this.getText = function getText() {  
7         return 'Text: ' + this.text;  
8       };  
9     },  
10    template: '<div class="message">{{ $ctrl.getText() }}</div>'  
11  });
```

Quelle: Eigene Darstellung

Die Verwendung von Komponenten sollte in den meisten Fällen vorgezogen werden. Allerdings verlieren Direktiven dadurch ihre Daseinsberechtigung nicht. Für die Manipulation von DOM-Elementen oder als CSS-Klasse sind Direktiven weiterhin sinnvoll. Dabei sollte sich ihre Form auf Attribute und CSS-Klassen beschränken.

Sofern ein Wechsel auf die neuste AngularJS-Version nicht möglich ist, lässt sich ein Polyfill ab Version 1.3 einsetzen.<sup>86</sup>

### 4.2.4 Programmiersprache

Sofern ein Upgrade auf Angular nicht sofort durchgeführt wird, kann die Entwicklung mit AngularJS durch den Einsatz alternativer Programmiersprachen verbessert werden. Hierfür bieten sich im Fall von Angular insbesondere ECMAScript 6 / 7 und TypeScript an.

Das Webprojekt setzt bereits vermehrt Neuerungen aus den kommenden ECMAScript-Standards ein, darunter insbesondere Module und Klassen aus ES6 sowie zum Teil Decorators aus ES7. Mithilfe von Klassen können Controller, Services, Direktiven und Komponenten einfacher deklariert werden. Im Zusammenspiel mit der Modularisierung enthalten die entsprechenden Dateien lediglich eine einzige Klasse. Die AngularJS-spezifische Registrierung dieser Klassen lässt sich an eine zentrale Stelle verschieben. Dadurch sinkt der Anteil des

---

<sup>86</sup> Siehe <https://github.com/toddmotto/angular-component>

Framework-eigenen Codes in diesen Klassen.

Ein weiterer Schritt ist die Einführung von TypeScript in das Projekt. Als Superset zu JavaScript ist jedes bestehende JavaScript-Programm gleichzeitig gültiger TypeScript-Code. Zusätzlich implementiert TypeScript kommende ECMAScript-Versionen und bietet dadurch die gleichen Vorteile.

Die Typisierung des Codes erlaubt im Vorfeld die statische Analyse durch den Compiler. Dadurch können Fehler früher entdeckt werden und treten nicht erst zur Laufzeit der Anwendung auf. Jegliche Typisierung bleibt für den Entwickler optional. Daher kann bestehender Code schrittweise angepasst werden. Für nicht typisierte Abschnitte versucht TypeScript durch Inferenz den Variablen-Typ zu erkennen und bietet ohne Aufwand Hilfestellung für typische Fehler.

In Hinsicht auf Angular entsteht durch den Umstieg auf TypeScript ein leichterer Migrationsweg. Allerdings sollte die Anpassung von bestehendem Code nur durchgeführt werden, solange der Wechsel auf Angular abgewartet wird. Andernfalls entsteht ein unnötiger Mehraufwand, welcher stattdessen in die eigentliche Migration investiert werden sollte.

### 4.2.5 Decorators

Ein Decorator erlaubt die Beeinflussung des Verhaltens einer Funktion. Auf der einfachsten Ebene ist ein Decorator lediglich eine Funktion, welche als Parameter eine Zielfunktion erhält. Diese kann daraufhin frei manipuliert werden. Eine besondere Deklaration eines Decorators existiert nicht. Auf Syntaxebene wird die Funktion über ein At-Zeichen (@) aufgerufen. Das darunterliegende Element wird als Ziel an den Decorator übergeben.

Im Projekt werden Decorators durch Babel ermöglicht und bereits eingesetzt, beispielsweise um Direktiven zu registrieren oder eine Annotation für Dependency Injection hinzuzufügen.

Listing 34: Decorators im SMARTCRM.Web-Projekt

```
1 import {Inject, Directive} from 'smartcrm';
2 ...
3 @Directive({
4     restrict: 'A',
5 })
6 @Inject(['$parse', '$timeout', 'debounce'])
7 class InfiniteScrollDirective {
8     ...
```

Quelle: Eigene Darstellung

Durch die Verwendung von Decorators lässt sich Logik an eine zentrale Stelle auslagern und innerhalb des Projekts flexibel wiederverwenden.

Es existiert eine Reihe von Projekten<sup>87</sup>, welche die Syntax von Angular zurückportieren. Für Entwickler ergibt sich dadurch ein Lerneffekt, aufgrund der Umstellung auf die neue Schreibweise. Der Einsatz von Decorators in AngularJS-Projekten kann daher im richtigen Kontext durchaus sinnvoll sein. Im Hintergrund wird die Anwendung weiterhin durch AngularJS kontrolliert, lediglich die Syntax kann auf diese Weise an Angular angeglichen werden.

### 4.2.6 Styleguides

Einige Mitglieder der Angular-Community haben eine Sammlung von Richtlinien für AngularJS-Anwendungen veröffentlicht. Diese Styleguides dienen Entwicklern als freiwillige Vorgabe zur Entwicklung. Dabei liegt der Fokus auf der Einhaltung von Best Practices und sauberer Codestrukturierung. Neue Entwickler können einen einheitlichen Styleguide als Referenz verwenden, um Unklarheiten aufzulösen und Konventionen durchzusetzen.

Zu den populärsten Styleguide-Autoren gehören John Papa<sup>88</sup> und Todd Motto<sup>89</sup>. In vielen Punkten gleichen sich die Anleitungen, da beide Autoren teilweise zusammengearbeitet haben. Selbstverständlich muss ein Styleguide nicht in seiner bestehenden Form genutzt werden.

---

<sup>87</sup> Siehe <https://github.com/ngUpgraders/ng-forward>, <https://github.com/ngParty/ng-metadata>

<sup>88</sup> Siehe <https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md>

<sup>89</sup> Siehe <https://github.com/toddmotto/angular-styleguide>

Stattdessen kann er als Grundlage verwendet und den Ansprüchen des Teams entsprechend adaptiert werden.

Für die Umstellung empfiehlt es sicher daher einen eigenen Styleguide, basierend auf bereits bestehenden Vorgaben, zu erarbeiten. Dadurch lassen sich klare Richtlinien und eine saubere Struktur für die Migration schaffen.

### 4.3 Strategien

Für die Durchführung der Migration existieren zwei Strategien, welche durch das Angular-Team empfohlen werden. Zusätzlich besteht die Möglichkeit, eine Anwendung von Grund auf neu zu entwickeln, um zukunftsfähig zu bleiben. Allerdings sollte jede Strategie unter Beachtung der Rahmenbedingungen ausschließlich im richtigen Kontext eingesetzt werden.<sup>90</sup>

- „*Big Bang*“ – Stopp der Entwicklung, bis die Migration ohne Unterbrechung abgeschlossen wurde
- *Inkrementell* – Fortlaufende Migration der Anwendung, ohne Unterbrechung der Entwicklung
- *Neuentwicklung* – Vollständiger Neustart des Projekts, wenig bis keine Code-Übernahme

In den folgenden Abschnitten werden die Strategien hinsichtlich ihrer Eigenschaften untersucht. Anschließend erfolgt die Auswahl eines Vorgehens, welches für das Webprojekt geeignet ist.

#### 4.3.1 Big Bang

Bei dieser Strategie wird die produktive, für den Kunden relevante Entwicklung des Projekts unterbrochen. Ziel ist es, in möglichst kurzer Zeit die Migration der Anwendung zu beenden.

Positive Effekte sind die direkte Verfügbarkeit der Neuerungen und damit einhergehend die Vorteile des neuen Frameworks. Darüber hinaus belastet die Portierung der Anwendung den produktiven Betrieb lediglich über einen möglichst kurzen Zeitraum.

---

<sup>90</sup> Vgl. Precht (2015)

Trotz der vielversprechenden Vorteile kann dieser Ansatz Probleme bereiten und sollte daher nicht in jedem Fall eingesetzt werden.

Eine Unterbrechung der Entwicklung zugunsten der Migration lässt sich nur schwer rechtfertigen. Hierfür müssen aussagekräftige Argumente gesammelt werden, um den Mehrwert des Umstiegs auf das neue Framework zu begründen. In jedem Fall sollte die Entwicklung, bei Einsatz dieser Vorgehensweise, pausieren.

Während der Evaluierung dieser Strategie sollte die Projektgröße als wichtigster Entscheidungsfaktor dienen. Mit steigender Größe und Komplexität nimmt die benötigte Zeit für die Migration zwangsläufig zu. Zeitgleich wird es immer schwieriger, einen Entwicklungsstopp zu vertreten.

Des Weiteren müssen auch die Abhängigkeiten der Anwendung in Betracht gezogen werden. Aufgrund des jungen Alters von Angular finden rapide Änderungen statt. Darüber hinaus ist die finale Veröffentlichung noch nicht vollständig erreicht. Daher sind viele erweiternde Bibliotheken zu Angular noch nicht kompatibel mit der neusten Version. Im Fall einer vollständigen Migration können daher noch nicht alle Abhängigkeiten eingesetzt werden. Eine Eigenentwicklung dieser Bibliotheken ist, in Bezug auf die Komplexität, nur selten sinnvoll.<sup>91</sup>

### 4.3.2 Inkrementell

Bei einem inkrementellem Vorgehen wird das Projekt schrittweise in kleinen Teilen migriert. Während der Migration besteht das Projekt aus einer Mischung von AngularJS- und Angular-Code. Ermöglicht wird die parallele Ausführung der Frameworks durch den Angular Upgrade Adapter.<sup>92</sup> Dieser ist bisher ein fester Bestandteil von Angular.

Diese Vorgehensweise erlaubt die größtmögliche Flexibilität bei der Migration der Teilbereiche. Es besteht eine vollständige Interoperabilität zwischen den Frameworks. Alle Komponenten, Direktiven, Controller und Services können für die jeweils andere Version des Frameworks zur Verfügung gestellt werden.

---

<sup>91</sup> Vgl. Hus (2015)

<sup>92</sup> Siehe <https://angular.io/docs/ts/latest/guide/upgrade.html#!#upgrading-with-the-upgrade-adapter>

Allerdings entsteht durch die Auslieferung und Ausführung zweier vollständiger Frameworks innerhalb einer Anwendung ein bedeutender Nachteil. Denn eine optimale Performance lässt sich dadurch nur schwer erreichen.

Die unterschiedlichen Prinzipien der vorliegenden Frameworks führen zu einem hybriden Zustand der Anwendung. Daher sollte auch bei dieser Strategie die Zeitspanne der eigentlichen Migration möglichst kurz gehalten werden.<sup>93</sup>

### 4.3.3 Neuentwicklung

Als Notlösung kann die vollständige Neuentwicklung der Anwendung in Betracht gezogen werden. Hierbei wird parallel zum bestehenden Projekt eine Anwendung auf Basis des neuen Frameworks entwickelt.

Diese Strategie wird auch als „Cold Turkey“-Vorgehensweise bezeichnet. Allerdings zeigten Analysen, dass eine Neuentwicklung in vielen Fällen unwirtschaftlich ist. In manchen Fällen wird sogar das Ziel nicht erreicht. Der Zeitaufwand, die Kosten und das Risiko sind bei diesem Vorgehen meist höher als bei einer Migration.<sup>94</sup>

Dennoch können kritische Schwächen oder Architekturfehler durch eine Neuentwicklung beseitigt werden. Darüber hinaus lassen sich die Best Practices des neuen Frameworks besser in die Codebasis integrieren. An dieser Stelle ist eine erneute Evaluation alternativer Framework-Lösungen möglich. Es bestehen die optimalen Voraussetzungen für den Einsatz einer anderen Architektur oder verschiedener Frameworks.

Zuletzt sollte auch bedacht werden, dass bei diesem Vorgehen ein Rennen zwischen den Projekten entsteht. Die Neuentwicklung ist stets bemüht, eine Feature-Parität zu erreichen, um die Altanwendung abzulösen. Werden zu wenige Ressourcen in das neue Projekt investiert, kann dieses die bestehende Anwendung nicht einholen.

---

<sup>93</sup> Vgl. Hus (2015)

<sup>94</sup> Krypczyk (2016), S. 47

### 4.4 Auswahl der Strategie

Die Auswahl einer geeigneten Strategie erfolgt anhand des Projekt-Kontexts und der Wirtschaftlichkeit der Vorgehensweise. Eine vollständige Neuentwicklung lässt sich schnell ausschließen. Die Grundlage für das Projekt wurde durch eine externe Webagentur geschaffen. In erster Linie würden die Kosten für die Entwicklung bis zu diesem Punkt verloren gehen. Darüber hinaus ist die Neuentwicklung äußerst ressourcenintensiv und erfordert eine Teilung des Entwicklerteams. Die hohen Kosten machen diese Vorgehensweise unwirtschaftlich.

Für die „Big Bang“-Migration an einem Stück muss die Entwicklung des Projekts unterbrochen werden. Dies ist nur schwer möglich, solange neue Features gefordert oder Fehler behoben werden müssen. Einige Kunden haben die Anwendung zu Testzwecken bereits im Einsatz oder warten auf die offizielle Auslieferung. Der Druck von außen ist an dieser Stelle zu hoch. Zusätzlich besteht die Gefahr, dass es während der Migrationsphase zu unerwarteten Problemen kommt. Dadurch würde sich der Zeitplan verschieben und der produktive Einsatz der Anwendung weiter verzögert.

Mit der inkrementellen Migration lässt sich das Projekt schrittweise umstellen. Diese Herangehensweise eignet sich insbesondere für große Anwendungen, welche sich bereits im operativen Einsatz befinden. Die Migration findet zeitgleich zur Entwicklung statt und führt dazu, dass beide Frameworks parallel betrieben werden. Für die Performance der Anwendung kann diese Tatsache zum Nachteil werden. Allerdings spielt die Auslieferungsgröße im innerbetrieblichen Umfeld eine kleinere Rolle. Im Außeneinsatz bei schwachen Datennetzen können aggressive Caching-Strategien eine Linderung verschaffen.

In Tabelle 15 werden die Strategien anhand verschiedener Voraussetzungen und Eigenschaften bewertet. Daraus ergibt sich, dass eine inkrementelle Migration für dieses Projekt am ehesten geeignet ist. Ausschlaggebend dafür ist vor allem die Projektgröße. Bei einer solchen Komplexität empfiehlt es sich, die Anwendung schrittweise umzustellen. Darüber hinaus erfordert eine inkrementelle Umstellung weniger Zeit pro Woche. Das Entwickler-Team kann frei entscheiden, wie viele Ressourcen investiert werden und kann die Migration auch vollständig unterbrechen.

Tabelle 15: Bewertung der Migrationsstrategien

	„Big Bang“	Inkrementell	Neuentwicklung
<b>Voraussetzung<sup>1</sup></b>			
<b>Projektgröße</b>	klein (-)	groß (+)	klein (-)
<b>Abhängigkeiten</b>	wenige (-)	viele (+)	wenige (-)
<b>Verfügbare Zeit</b>	viel (-)	wenig (-)	sehr viel (--)
<b>Eigenschaft</b>			
<b>Zeitaufwand<sup>2</sup></b>	mittel (+)	hoch (-)	sehr hoch (--)
<b>Kosten</b>	hoch (-)	mittel (+)	sehr hoch (--)
<b>Risiko</b>	sehr hoch (--)	mittel (+)	hoch (-)
<b>Framework-Integration</b>	hoch (+)	mittel (-)	sehr hoch (++)
<b>Ergebnis</b>	<b>8 Punkte</b>	<b>11 Punkte</b>	<b>6 Punkte</b>
<i>Legende:</i> (--) = 0 Punkte (-) = 1 Punkt (+) = 2 Punkte (++) = 3 Punkte		<sup>1</sup> In Bezug auf das vorliegende Projekt <sup>2</sup> Gesamter Zeitraum der Migration	

Quelle: Hus (2015) & eigene Bewertung

## 4.5 Durchführung

Mit der Durchführung beginnt die praktische Umstellung des Projekts auf Angular. Innerhalb dieser Phase wird die Anwendung durch beide Versionen des Frameworks gleichzeitig kontrolliert. Aufbauend auf der Vorbereitung werden die Komponenten schrittweise migriert. Zusätzlich müssen die Abhängigkeiten der Applikation beachtet werden. Um die Migration abzuschließen, müssen diese das neue Framework unterstützen oder ersetzt werden. Das Ziel ist es, AngularJS letztendlich vollständig aus der Anwendung zu entfernen. Sobald dieses Ziel erreicht ist, gilt die Durchführung als abgeschlossen.

Zu Beginn bezieht sich dieses Kapitel auf die Umstellung der eigentlichen Anwendung. In 4.5.6 werden die Abhängigkeiten des Projekts genauer untersucht.

### 4.5.1 Bootstrap

Der erste Schritt betrifft die Initialisierung der Anwendung. Eine AngularJS-Applikation lässt sich implizit durch die *ng-app* Direktive starten. Hierfür durchsucht das Framework das Dokument nach der entsprechenden Direktive und verwendet den angegebenen Modulnamen als Einstiegspunkt. Diese Art des Bootstrapping wird in Angular nicht mehr unterstützt. Stattdessen wird die Bootstrap-Methode von Angular genutzt, um die Anwendung zu starten. Diese Form wird auch von AngularJS unterstützt und lässt sich daher leicht umstellen.

Listing 35: Manueller Bootstrap in AngularJS

```

1 // app.jade
2 // Vorher: body(ng-app='smartcrm')
3 // Nachher:
4 body
5
6 // app.js
7 angular.element(document).ready(() =>
8     angular.bootstrap(document.body, ['smartcrm']);
9 });

```

Quelle: Eigene Darstellung

Der neue Code ist an dieser Stelle noch nicht Angular spezifisch. Allerdings ist es eine Voraussetzung für die Verwendung des Upgrade Adapters.

### 4.5.2 Setup

Bevor der Angular Upgrade Adapter verwendet werden kann, muss das neue Framework, inklusive der Abhängigkeiten, installiert werden. Die benötigten Pakete werden als *Dependencies* in der *package.json* des Projekts eingetragen. In der Dokumentation befindet sich eine aktuelle Liste der Abhängigkeiten und ihrer empfohlenen Version.<sup>95</sup> Es müssen nicht alle aufgelisteten Pakete installiert werden. In Tabelle 16 sind die neuen Abhängigkeiten der Migration gezeigt.

Tabelle 16: Abhängigkeiten von Angular

Angular	Polyfill
@angular/core	core-js
@angular/common	reflect-metadata
@angular/compiler	rxjs
@angular/forms	zone.js
@angular/http	
@angular/platform-browser	
@angular/platform-browser-dynamic	
@angular/upgrade	

Quelle: Eigene Darstellung

Die Abhängigkeiten lassen sich in die Kategorien *Angular* und *Polyfill* unterteilen. Dem Namen entsprechend bezieht sich die erste Kategorie auf eigene Pakete von Angular. Darin befinden sich unter anderem die Kernfunktionalität von Angular, häufig genutzte Objekte und der Angular Compiler, um HTML in dynamische Ansichten zu verwandeln. Gegenüber der ausführenden Plattform (Browser, Server, mobile OS) versucht das Framework agnostisch zu sein. Daher wird Browser-spezifische Funktionalität in separate Pakete ausgelagert. Gleiches gilt für den Angular Upgrade Adapter. Dieser wird ebenfalls als eigenständige Abhängigkeit geführt.

---

<sup>95</sup> Siehe <https://angular.io/docs/ts/latest/guide/npm-packages.html>

Abgesehen von den Framework-Abhängigkeiten erfordert Angular einige Polyfills. Dadurch wird fehlende Funktionalität in Browersn ergänzt. Mit *core-js* wird die Unterstützung von ES6 sichergestellt. Eine Implementierung der Decorator-Funktionalität wird durch *reflect-metadata* bereitgestellt. Durch *rxjs* und *zone.js* werden Implementierungen für die Spezifikation von Observables und Zones geliefert. Eine Zone erlaubt deutlich schnellere Change Detection in Angular, ohne zusätzlichen Code auf Seite des Entwicklers.

Der Build-Prozess des Projekts bezieht Module innerhalb des *node\_modules* Ordners bereits in die Kompilierung durch Webpack mit ein. Daher müssen an dieser Stelle keine Änderungen vorgenommen werden. Ursprünglich verwendet das Projekt Bower für die Verwaltung der Abhängigkeiten im Frontend. Angular unterstützt diese Variante weiterhin. Allerdings wächst die Popularität von npm im Frontend Bereich. In Kombination mit Webpack bedeutet es den geringsten Aufwand und ist daher zu bevorzugen.

Bevor der Upgrade Adapter verwendet kann, müssen lediglich die neuen Abhängigkeiten in JavaScript importiert werden. Ohne eine Referenz durch eine Import-Anweisung werden diese nicht berücksichtigt.

Listing 36: Import der Angular Abhängigkeiten

```
1  import 'core-js/client/shim';
2  import 'zone.js';
3  import 'reflect-metadata';
4  import 'rxjs';
5  import '@angular/core';
6  import '@angular/common';
7  import '@angular/platform-browser';
8  import '@angular/platform-browser-dynamic';
9  import '@angular/http';
10 import '@angular/forms';
```

Quelle: Eigene Darstellung

Die Import-Anweisungen können in einer eigenen JavaScript-Datei abgelegt werden. Ausgehend vom Einstiegspunkt der Anwendung lässt sich diese aufrufen. Ab diesem Punkt wird das Angular-Framework geladen und steht zur Verfügung.

### 4.5.3 Hybride Ausführung

Im nächsten Schritt wird der hybride Betrieb beider Frameworks erreicht. Die Anwendung verhält sich weiterhin wie gewohnt und wird vollständig durch AngularJS kontrolliert. Lediglich der Bootstrap-Aufruf wird an den Angular Upgrade Adapter übergeben. Die Syntax gleicht der bisherigen Methode, um den Umstieg zu vereinfachen. Allerdings verhält sich die Funktion des Upgrade Adapters asynchron. Daher ist nicht garantiert, dass die Anwendung bereits initialisiert ist, wenn die Ausführung der Bootstrap-Methode abgeschlossen ist.

Listing 37: Bootstrap durch den Angular Uprade Adapter

```
1 // app.js
2 import { UpgradeAdapter } from '@angular/upgrade';
3
4 angular.module('smartcrm', ...);
5
6 export const upgradeAdapter = new UpgradeAdapter();
7
8 angular.element(document).ready(() => {
9   upgradeAdapter.bootstrap(document.body, ['smartcrm']);
10});
```

Quelle: Eigene Darstellung

Das App-Modul sollte die Instanz des Upgrade Adapters exportieren. Für die Migration muss dieses Objekt an anderer Stelle wiederverwendet werden. Eine weitere Verbesserung ist die Auslagerung des Upgrade Adapters in eine separate Datei. Diese kümmert sich ausschließlich um die Instanziierung des Adapters und exportiert diesen anschließend. Das *Single Responsibility Principle* wird auf diesem Weg eingehalten.

Für eine hybride Ausführung der Anwendung reichen daher im Grunde zwei Änderungen. Zuerst müssen neue Abhängigkeiten von Angular in das Projekt übernommen werden. Im zweiten Schritt übernimmt der Upgrade Adapter den Bootstrap-Prozess.

Die Config- und Run-Blöcke der Applikation können zunächst unverändert bleiben. Provider aus AngularJS oder von externen Modulen werden innerhalb dieser Phasen konfiguriert. Daher werden diese Konfigurationsabschnitte benötigt, solange sich diese Provider in Verwendung befinden. In Angular entfällt dieses Konzept der Config- und Run-Phase und wird nicht mehr

durch das Framework geregelt (Siehe 3.2.5). Daher kann dieser Teil der Anwendung frei gestaltet werden. Denkbar wäre eine separate Config-Datei, welche entsprechende Aufrufe zur Konfiguration externer Module enthält. Vor dem Start der Anwendung lässt sich diese importieren, um die notwendigen Vorbereitungen zu treffen.

Durch den Upgrade Adapter wird der parallele Betrieb der beiden Frameworks gewährleistet. Dabei wird auf Emulation und Einschränkung der Frameworks verzichtet.<sup>96</sup> Das Modul kümmert sich lediglich um die Kommunikation und Kompatibilität zwischen den Versionen. Betroffen sind dadurch drei große Bereiche:

- Dependency Injection
- DOM
- Change Detection

### **Dependency Injection**

Mit Hilfe des Adapters können Services innerhalb der Dependency Injection für die jeweils andere Version zur Verfügung gestellt werden. Von AngularJS ausgehend geschieht dies durch ein *Upgrade* des Service. In Angular wird dieser unter dem gleichen String registriert wie bisher. Die umgekehrte Richtung wird durch ein *Downgrade* ermöglicht. An dieser Stelle wird ein String übergeben für die Registrierung im Dependency Injection System. AngularJS arbeitet weiterhin mit Strings und kann Typisierung nicht als Vorteil für Dependency Injection verwenden. Es gilt zu beachten, dass dieselbe Singleton-Instanz eines Service zwischen den Frameworks geteilt wird.

### **DOM**

Innerhalb des DOMs können Komponenten und Direktiven der Frameworks frei gemischt werden. Die Kommunikation findet über die entsprechenden Input- und Output-Bindings des jeweiligen Frameworks statt. Alternativ können geteilte Service-Klassen über Dependency Injection genutzt werden.

---

<sup>96</sup> Vgl. Google (2016)

Trotz der hybriden Ausführung wird jedes Element des DOM nur durch ein Framework kontrolliert. Das jeweils andere Framework ignoriert das entsprechende Element. Darüber hinaus wird das oberste Element des Komponenten-Baums einer hybriden Anwendung in jedem Fall durch AngularJS kontrolliert.

Sobald eine Komponente von Angular verwendet wird, erfolgt ein Kontextwechsel. Während die Komponente sich weiterhin innerhalb eines AngularJS Templates befindet, kümmert sich Angular um das Template dieser Komponenten. Ein Kontextwechsel wird demnach durch die Verwendung einer Komponente des jeweils anderen Frameworks erreicht. Die zugehörige View dieser Komponente wird durch das entsprechende Framework kontrolliert.

### **Change Detection**

Eine AngularJS-Anwendung durchläuft regelmäßig einen Zyklus, um Änderungen am Zustand der Daten zu erkennen. Sofern eine Änderung erfasst wurde, wird die Ansicht entsprechend aktualisiert. Ausgelöst wird dieser Vorgang durch den Aufruf von `$scope.$apply()`. Dies geschieht in den meisten Fällen automatisch oder wird durch Code des Entwicklers manuell ausgelöst.

Angular verwendet *Zones* als Kontext für die Ausführung asynchroner Funktionen. Deshalb wird der entsprechende Polyfill `zone.js` benötigt. Ursprünglich stammt die Idee aus der Programmiersprache Dart und wurde für JavaScript portiert.<sup>97</sup> Für Angular ergibt sich ein entscheidender Vorteil aus der Verwendung von *Zones*. Ein Aufruf der Apply-Funktion ist nicht mehr nötig. Der gesamte Code von Angular wird innerhalb einer Zone ausgeführt. Das Framework kann automatisch feststellen, wenn die Ausführung eines Code-Abschnitts beendet ist und führt darauf folgend den Zyklus der Change Detection aus. Allerdings profitiert nicht nur Angular durch diese Neuerung. Der Upgrade Adapter leitet die Benachrichtigungen der Angular Zone an AngularJS weiter. Auf die Change Detection von Angular folgt daher automatisch `$rootScope.$apply()` in AngularJS. Dennoch das Entfernen zusätzlicher `$apply()`-Aufrufe keine hohe Priorität besitzen. Sie haben keinen Einfluss auf die hybride Anwendung.

---

<sup>97</sup> Abgerufen am 16. August 2016 von <https://www.dartlang.org/articles/libraries/zones>

#### 4.5.4 Komponenten

Durch den Upgrade Adapter werden zwei Möglichkeiten geboten, um Komponenten zu migrieren: *Downgrade* und *Upgrade*. Bei einem Downgrade wird eine Angular-Komponente zu einer AngularJS-Direktive gewandelt. Umgekehrt können AngularJS-Direktiven mittels eines Upgrades als Angular-Komponenten verwendet werden. Voraussetzung hierfür ist, dass die Direktiven über die Component-Funktion von AngularJS registriert wurden. Alternativ reicht es aus, die Direktiven mit den entsprechenden Eigenschaften einer Komponente zu erstellen.<sup>98</sup>

##### Downgrade

In den meisten Fällen wird eine einzelne Direktive zu Angular migriert und muss weiterhin in einem AngularJS-Kontext funktionieren. Um dieses Ziel zu erreichen, wird die Downgrade-Methode des Upgrade Adapters eingesetzt. Als Parameter wird eine Angular-Komponente übergeben. Anschließend kann die resultierende Direktive in AngularJS registriert werden. Damit befindet sich der Code bereits vollständig auf dem Stand der neusten Version.

In Listing 38 wird die Angular-Komponente aus Übersichtsgründen ausgelassen.

Listing 38: Downgrade einer Angular-Komponente

```

1  // migrated.component.ts
2  import { Component, Input, Output, EventEmitter } from '@angular/core';
3  import upgradeAdapter from './upgrade-adapter';
4
5  @Component({
6    selector: 'migrated-component'
7  })
8  export class MigratedComponent {
9    @Input() text: string;
10   @Output() click = new EventEmitter<Object>();
11 }
12
13 const downgradedComponent = upgradeAdapter.downgradeNg2Component(MigratedComponent);
14
15 angular.module('smartcrm')
16   .directive('migratedComponent', downgradedComponent);
17

```

<sup>98</sup> Siehe 4.2.3

```

18 // Beispiel: Verwendung in AngularJS
19 <div ng-controller="AppController as $ctrl">
20   <migrated-component [text]="$ctrl.text"
21     (click)="$ctrl.onClick($event)"
22     ng-repeat="item in $ctrl.items">
23   </migrated-component>
24 </div>

```

Quelle: Eigene Darstellung

Angesprochen wird die Direktive weiterhin über den bei der Registrierung angegebenen Bezeichner. Allerdings muss für das Binding der Attribute die Syntax von Angular verwendet werden (Zeile 20-21 in Listing 38). Die entsprechenden Ausdrücke der Attribute entsprechen wiederum der Syntax von AngularJS. Zusätzlich können beliebige AngularJS-Direktiven an dem Element angebracht werden (Zeile 22 in Listing 38).

### Upgrade

Während der Migration ist es gut möglich, dass eine Angular-Komponente in ihrem Template Direktiven aus AngularJS benötigt. Dieses Szenario wird ebenfalls durch den Upgrade Adapter abgedeckt. Hierfür erhält eine AngularJS-Direktive ein Upgrade und kann anschließend innerhalb des neuen Frameworks eingesetzt werden. Für die Direktive gelten die gleichen Voraussetzungen hinsichtlich der Eigenschaften einer Komponente.

Listing 39: Upgrade einer AngularJS-Direktive

```

1 import { Component } from '@angular/core';
2 import upgradeAdapter from './upgrade-adapter';
3
4 const UpgradedDirective = upgradeAdapter.upgradeNg1Component('myDirective');
5
6 @Component({
7   selector: 'app-component',
8   template: '<my-directive></my-directive>',
9   directives: [UpgradedDirective]
10 })
11 export class AppComponent {}

```

Quelle: Eigene Darstellung

Um das Upgrade einer Direktive durchzuführen, wird ihr Name an den Upgrade Adapter übergeben (Zeile 4 in Listing 39). Daraufhin kann das resultierende Objekt zum Directives-

Array einer Komponente hinzugefügt werden (Zeile 9 in Listing 39). Ab diesem Punkt kann die Angular-Komponente die Direktive innerhalb ihres Template verwenden. Der Selektor für die Direktive entspricht der Version in AngularJS.

Für das Binding der Attribute wird auch an dieser Stelle die Syntax von Angular verwendet. Der Upgrade Adapter sorgt für die Zusammenarbeit zwischen den Frameworks.

Tabelle 17: Template Syntax für AngularJS-Bindings

Art des Binding	Definition	Template Syntax
<b>Attribut</b>	<code>attribute: ,@'</code>	<code>attribute="value"</code>
<b>Ausdruck</b>	<code>output: ,&amp;'</code>	<code>(output)="action()"</code>
<b>One-way<sup>99</sup></b>	<code>value: ,&lt;'</code>	<code>[value]="expression"</code>
<b>Two-way</b>	<code>value: ,='</code>	<code>[(value)]="expression"</code>

Quelle: Google (2016)<sup>100</sup>

### 4.5.5 Dependency Injection

Unter Verwendung des Upgrade Adapters können Services für das jeweils andere Framework zur Verfügung gestellt werden. Der Begriff Services umfasst an dieser Stelle auch andere Objekte aus AngularJS, wie Factory oder Provider.

Das Prinzip des Up- und Downgrades der Komponenten wird auch für Services angewendet. Mit Übergabe des Bezeichners einer AngularJS-Klasse kann diese durch den Upgrade Adapter für Angular bereitgestellt werden.

---

<sup>99</sup> Verfügbar ab AngularJS 1.5

<sup>100</sup> Abgerufen am 17. August 2016 von <https://angular.io/docs/ts/latest/guide/upgrade.html#!#using-angular-1-component-directives-from-angular-2-code>

Listing 40: Upgrades eines AngularJS-Service

```
1 // example.service.ts
2 import { Injectable } from '@angular/core';
3 import upgradeAdapter from './upgrade-adapter';
4
5 @Injectable()
6 export class ExampleService {}
7
8 angular.module('smartcrm')
9   .service('exampleService', ExampleService);
10
11 upgradeAdapter.upgradeNg1Provider('exampleService');
12
13 // app.component.ts
14 import { Component, Inject } from '@angular/core';
15 import ExampleService from './example.service';
16
17 @Component(...)
18 export class AppComponent {
19   constructor(@Inject('exampleService') exampleService: ExampleService) {}
20 }
```

Quelle: Eigene Darstellung

Normalerweise nutzt Angular die Typisierung der Parameter des Konstruktors, um die entsprechenden Abhängigkeiten zu finden. Im Fall eines Upgrades muss das Framework allerdings auf einen String zurückgreifen. Auf diese Weise entspricht das System AngularJS.

Daher wird der Inject-Decorator eingesetzt, um die benötigten Informationen für die Dependency Injection zu ergänzen (Zeile 19 in Listing 40). Die Angabe des Typs ist optional und verbessert lediglich die statische Analyse des Codes durch unterstützende Tools.

Umgekehrt können neue Services aus Angular ein Downgrade erhalten. Dadurch können die Klassen von AngularJS genutzt werden. Dabei gleicht das Vorgehen der Upgrade-Methode. Es muss lediglich die Service-Klasse zusätzlich als Provider am Upgrade Adapter registriert werden.

Listing 41: Downgrade eines Angular-Service

```

1  // example.service.ts
2  import { Injectable } from '@angular/core';
3  import upgradeAdapter from './upgrade-adapter';
4
5  @Injectable()
6  export default class ExampleService {}
7
8  upgradeAdapter.addProvider(ExampleService);
9
10 const downgradedProvider = upgradeAdapter.downgradeNg2Provider(ExampleService);
11
12 angular.module('smartcrm')
13   .factory('exampleService', downgradedProvider);

```

Quelle: Eigene Darstellung

Der Upgrade Adapter liefert nach dem Aufruf der Downgrade-Methode eine Factory-Funktion für die Verwendung in AngularJS zurück. Daher muss das Objekt als Factory registriert werden.

### 4.5.6 Abhängigkeiten

Die Abhängigkeiten des Projekts spielen eine wichtige Rolle für die erfolgreiche Migration. Es müssen jedoch lediglich Module beachtet werden, welche in irgendeiner Form mit Angular in Verbindung stehen.

Tabelle 18: Angular spezifische Abhängigkeiten des Projekts

	Anzahl	Anteil
<b>Gesamt</b>	107	100%
<b>npm</b>	76	~ 71%
<b>Bower</b>	31	~ 29%
<b>Angular spezifisch</b>	<b>19</b>	<b>~ 18%</b>

Quelle: Eigene Darstellung

Angular befindet sich in der Phase der „Release Candidates“ und steht kurz vor der finalen Veröffentlichung.<sup>101</sup> Dennoch unterläuft das Framework zurzeit zahlreiche Änderungen, welche teilweise die Kompatibilität zum vorherigen Release Candidate betreffen. Einige Stellen der API sind als überholt (englisch: deprecated) markiert und werden vor der ersten Version noch entfernt. Daher ist es für Drittanbieter schwierig, ihre Module bereits zu diesem Zeitpunkt auf Angular umzustellen. Dennoch arbeiten viele Projekte, parallel zur Entwicklung von Angular, an ihrer eigenen Migration. Darüber hinaus portieren unabhängige Personen alte AngularJS-Module auf die neue Version. Ermöglicht wird das durch die Open Source Community hinter den Projekten.

Solange sich die Anwendung im hybriden Zustand zwischen den Frameworks befindet, können die Abhängigkeiten unberührt bleiben. Um die Migration jedoch abzuschließen, ist es notwendig, jede Referenz auf AngularJS zu entfernen. Daher müssen die genutzten Drittanbieter Pakete für AngularJS durch eine neue Version, welche mit Angular kompatibel ist, ersetzt werden. Falls solch eine Aktualisierung nicht angeboten wird, muss die Abhängigkeit durch eine andere Lösung ausgetauscht werden. Dabei kann es sich um ein alternatives Modul oder eine Eigenentwicklung handeln. Allerdings bedeutet das in den meisten Fällen einen bedeutend höheren Aufwand.

Innerhalb des SMARTCRM Web-Projekts sind drei Abhängigkeiten zentrale Bausteine für die Anwendung:

- Ionic
- Angular Schema Forms
- Angular Translate
- UI-Router

### **Ionic**

Das Ionic Framework ist ein Open Source-Projekt zur Entwicklung mobiler Apps unter Verwendung von Webtechnologien. Es kümmert sich primär um das Aussehen und die

---

<sup>101</sup> Stand 17. August 2016

Interaktion einer Anwendung. In Kombination mit PhoneGap oder Cordova können mobile Applikationen für verschiedene Plattformen erstellt werden.

Ionic selbst basiert auf AngularJS, um dynamische Interaktion umzusetzen. Allerdings soll es in Zukunft möglichst agnostisch gegenüber dem eingesetzten JavaScript-Framework sein.<sup>102</sup> Aufgrund der tiefen Integration mit AngularJS befindet sich zurzeit Ionic 2 in der Entwicklung, um auf Angular umzusteigen. Diese Neuentwicklung befindet sich in der Version Beta 11 und basiert auf dem Angular Release Candidate 4.<sup>103</sup>

Die Dokumentation für die neue Version ist bereits nahezu vollständig. Insbesondere Komponenten und native Features sind ausführlich beschrieben. Allerdings fehlt bisher eine detaillierte Anleitung zur Migration einer Anwendung auf Basis von Ionic und AngularJS. Mit der Zeit werden dafür mehr Ressourcen und Tools entstehen. Dafür müssen die Frameworks, Ionic und Angular, zuerst ihre finale Veröffentlichung erreichen. Ab dann wird auch die Community hinter den Projekten an zusätzlichen Beiträgen arbeiten.

Für die Migration kann die Umstellung von Ionic vorerst aufgeschoben werden. Solange die Anwendung in ihrem hybriden Zustand vorliegt, wird AngularJS weiterhin unterstützt. Daher sollten zuerst andere Schritte der Migration umgesetzt werden. Sobald das Projekt entsprechend vorbereitet wurde und die Grundlagen der Umstellung umgesetzt wurden, lässt sich der Status von Ionic 2 erneut evaluieren. Die Abhängigkeiten für die neue Version können zusätzlich zum Projekt hinzugefügt werden. Anschließend lassen sich gezielt einzelne Komponenten von Ionic migrieren.

### **Angular Schema Forms**

Die Generierung der Formulare des Projekts wird durch das Modul *Angular Schema Forms* übernommen. Anstelle einer statischen Definition der Formulare wird ein Schema in Form eines JSON-Objekts benutzt. Üblicherweise enthalten dynamische Formulare komplexe Interaktionen und Validierungen, um dem Benutzer eine optimale Erfahrung und frühzeitiges Feedback zu bieten. Ein aufwendiges Ausprogrammieren dieser Funktionalität für jede

---

<sup>102</sup> Vgl. Drifty Co (2016)

<sup>103</sup> Vgl. Bucholtz (2016)

Eingabemöglichkeit führt zu Zeitverlust. Daher nimmt das Modul ein Schema, um das zugehörige Formular automatisch zu generieren. Innerhalb des Schemas werden die entsprechenden Validierungsregeln festgelegt, sowie die Arten der Eingabefelder.

Angular Schema Forms durchläuft eine Reihe organisatorischer Veränderungen. Es entsteht zurzeit ein allgemeiner *json-schema-form* Standard. Dieser ist losgelöst von einer spezifischen Programmiersprache. Auf diesem Weg lassen sich weitere Adaptionen für beliebige Plattformen und Frameworks erstellen. Geplant sind React, Delphi und Laravel-Portierungen. Eine Ankündigung zum Status der Unterstützung von Angular existiert noch nicht. Dies könnte durch die weitreichenden, strukturellen Veränderungen des gesamten Projekts bedingt sein.<sup>104</sup>

Allerdings befinden sich auch an dieser Stelle Open Source-Projekte unter aktiver Entwicklung, um die gleiche Funktionalität für Angular zu bieten.<sup>105</sup> Daher kann die Entscheidung getroffen werden, auf ein alternatives Modul umzusteigen oder schlichtweg eine neue Version von Angular Schema Forms abzuwarten, welche die benötigte Kompatibilität hat.

### **Angular Translate**

Mit diesem Modul wird die Mehrsprachigkeit der Anwendung ermöglicht. Es stellt unter anderem Filter und Direktiven bereit, sowie das asynchrone Laden der Sprachdateien. Aufgrund der Natur von Übersetzungen zieht sich die Verwendung von Angular Translate an mehreren Stellen durch das Projekt. Daher muss für Angular eine kompatible Version gefunden werden. Während der Migration kann das alte Modul weiterhin problemlos eingesetzt werden. Der Upgrade Adapter unterstützt den Einsatz von AngularJS-Direktiven.

Eine Weiterentwicklung des Moduls mit Kompatibilität zu Angular existiert bisher noch nicht.<sup>106</sup> Es kann jedoch auf alternative Projekte der Community zurückgegriffen werden. Darüber hinaus wird Angular eine eigene Implementierung für Übersetzungen bieten. Teile

---

<sup>104</sup> Vgl. Börjesson (2016)

<sup>105</sup> Siehe <https://github.com/makinacorpus/angular2-schema-form>

<sup>106</sup> Stand 18. August 2016

dieser Funktionalität wurden mit Veröffentlichung des *Release Candidate 5* hinzugefügt.<sup>107</sup> Allerdings sind die Neuerungen noch nicht in der Dokumentation festgehalten. Spätestens zum Zeitpunkt der ersten offiziellen Version von Angular werden diese fertiggestellt.

### UI-Router

Das Routing der Anwendung wird mit Hilfe des Angular UI-Routers umgesetzt. Als Community Projekt hat es sich als De-facto-Standard durchgesetzt. Gegenüber der eingebauten Routing-Funktionalität von AngularJS erlaubt der UI-Router eine höhere Flexibilität durch den hierarchischen Aufbau von Routen. Eine Weiterentwicklung des Moduls mit Kompatibilität zu Angular befindet sich im Alpha-Stadium.<sup>108</sup> Es steht demnach als frühe Entwicklerversion bereit und sollte vorerst noch nicht in einer produktiven Umgebung eingesetzt werden.

Allerdings besteht die Möglichkeit die bisherige Funktionalität im Zuge der Migration mit dem Component Router aus Angular zu implementieren (Siehe 3.2.12). Als Vorteil ergibt sich die tiefe Integration mit Angular und die offizielle Unterstützung durch das Angular-Team. Im Vergleich zu AngularJS weist die neue Router-Lösung deutlich mehr Funktionalität auf und ist vergleichbar mit dem UI-Router. Die bisherige Definition der Routen lässt sich durch den Componen Router wie in Listing 42 abbilden.<sup>109</sup>

Listing 42: Migration der Routing Definition

```
1 import { RouterConfig } from '@angular/router';
2 import { HomeComponent } from './home.component';
3 import { PageComponent } from './page.component';
4
5 export default const routes: RouterConfig = [
6   path: '/',
7   component: HomeComponent,
8   children: [ { path: 'page/:pageName/:index', component: PageComponent } ]
9 ];
```

Quelle: Eigene Darstellung

---

<sup>107</sup> Siehe <https://github.com/angular/angular/blob/master/CHANGELOG.md>, „i18n“

<sup>108</sup> Abgerufen am 22. August 2016 von <http://angular-ui.github.io/ui-router/1.0.0-alpha.3/modules/ng2.html>

<sup>109</sup> Siehe Listing 43

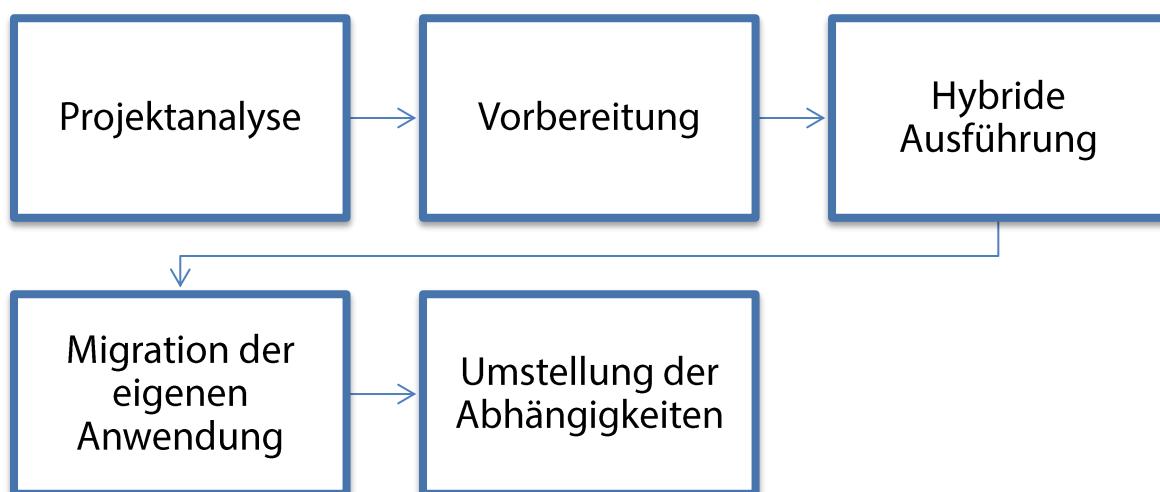
## 4.6 Ergebnis

Anhand der vorangegangenen Abschnitte lässt sich ein grober Ablauf für die Migration der Anwendung ableiten. Zu Beginn wird eine Analyse des Projekts vorgenommen, um ein grundlegendes Verständnis aufzubauen. Außerdem lässt sich die Einflussreichweite der Umstellung besser einschätzen.

Die Vorbereitung stellt eine der wichtigsten Phasen im gesamten Prozess der Migration dar. Durch die beschriebenen Maßnahmen wird die Struktur des Projekts und seiner Bestandteile an die Prinzipien von Angular angepasst. Eine strukturierte Grundlage ist essentiell für eine möglichst reibungslose Umstellung.

Sobald die Vorbereitung abgeschlossen ist erfolgt die eigentliche Migration. Grundlage dafür bietet die hybride Ausführung oder genauer der parallele Betrieb von AngularJS und Angular. Erreicht wird dieser Meilenstein durch den Einsatz des Upgrade Adapters. Dieser kann selbst bei einem komplexen Projekt leicht integriert werden. Die letzten beiden Teile betreffen die Migration der Anwendung und ihrer Abhängigkeiten. Primär sollte der Fokus zuerst auf der Umstellung des eigenen Projekts liegen. Während der hybriden Ausführung können die Abhängigkeiten weiterverwendet werden. Drittanbieter müssen vorerst ihre eigenen Module zu Angular migrieren, daher ist die Umstellung der Abhängigkeiten als letztes aufgelistet. Sie kann sich jedoch mit der vorangegangenen Phase überschneiden.

Abbildung 17: Ablauf der Migration



Quelle: Eigene Darstellung

### 5. Schlussbetrachtung

Zum Abschluss dieser Arbeit folgt eine Zusammenfassung der erreichten Ziele. Anschließend wird ein Ausblick gegeben auf neue Erkenntnisse für die Zukunft des Projekts, sowie eine Betrachtung des weiteren Verlaufs. Letztendlich werden die Durchführung und die Ergebnisse bewertet.

#### 5.1 Zusammenfassung

Mit Abschluss dieser Arbeit wurde eine ausführliche Analyse des Angular Frameworks vorgenommen. Diese umfasst die Gründe für die Neuentwicklung des Frameworks, eine Entscheidungsgrundlage für die Auswahl der Programmiersprache und die einzelnen Bestandteile des Frameworks. Dabei wird ein Bezug zum kommenden Standard der Web-Komponenten hergestellt. Es wird erarbeitet, welche Unterschiede die neue Version mit sich bringt und wie sich diese in Angular verankern. Auf diesem Weg werden die abweichenden Konzepte der beiden Frameworks hervorgehoben. Anschließend wurden alternative JavaScript-Frameworks untersucht und in einen Vergleich mit Angular gebracht. Dadurch zeigen sich die Anwendungsbereiche der Frameworks, sowie das Potential hinter den Projekten. Anhand dieser Ergebnisse wurde festgestellt, dass Angular für eine Migration eine optimale Erfahrung bietet und gleichzeitig wird die Wirtschaftlichkeit der Umstellung gewährleistet.

Im nächsten Teil der Arbeit wurde die Migration der Webapplikation näher betrachtet. Hierfür war zunächst eine umfassende Projektanalyse notwendig, um ein grundlegendes Verständnis für die Applikation aufzubauen. Zusätzlich konnte gezeigt werden, welche Bereiche der Anwendung durch eine Migration betroffen sind. Auf diese Weise entsteht eine grobe Vorstellung über den bevorstehenden Aufwand.

Zunächst wurden vorbereitende Maßnahmen erarbeitet. Diese sind erforderlich für eine effiziente und reibungslose Migration des Projekts. Des Weiteren kann die Vorbereitung sämtliche Zeit vor der eigentlichen Umstellung abdecken. Damit ist sie größtenteils unabhängig von Angular und kann flexibel durchgeführt werden. Allgemein verbessert sich dadurch die Struktur der gesamten Anwendung.

Darüber hinaus wurden verschiedene Vorgehensweisen für die Migration betrachtet und gegenüber gestellt. Die Strategien wurden dafür anhand diverser Voraussetzungen und Eigenschaften miteinander verglichen und in Bezug auf das Projekt gesetzt. Mit dem Vergleich zeigt sich, dass eine inkrementelle Umstellung der Anwendung für die Webapplikation geeignet ist. Bedingt wird diese Tatsache durch die Komplexität und den produktiven Einsatz der Anwendung. Eine vollständige Unterbrechung der Entwicklung steht in keinem Verhältnis zum Mehrwert der Migration.

Das letzte Kapitel umfasst die Durchführung der Umstellung auf Angular. Darin sind praktische Schritte und Empfehlungen enthalten, um den Code an das neue Framework anzupassen. Ein wichtiger Aspekt sind die Eigenschaften der hybriden Ausführung. Daraus ergeben sich bestimmte Konsequenzen und Möglichkeiten für das Projekt. Mitunter erlaubt es die Ausführung von Abhängigkeiten auf Basis von AngularJS. Daher ist es möglich, zuerst eigene Komponenten zu migrieren, bevor ein Austausch der Abhängigkeiten notwendig wird.

### 5.2 Ausblick

Aufgrund des jungen Alters von Angular gilt es, zunächst die erste offizielle Veröffentlichung abzuwarten. Damit einhergehend wird das Framework weiter verändert und Neuerungen werden hinzugefügt. Darüber hinaus wird zu einem ähnlichen Zeitpunkt die Dokumentation vollständig sein und in veralteten Bereichen aktualisiert.

Der Vorgänger AngularJS wird für mehrere Jahre unterstützt bleiben. Dies wird durch Google und die Community sichergestellt sein. Dadurch ergibt sich zunächst eine große Zeitspanne bis eine Migration wirklich dringend erforderlich wird. Des Weiteren müssen Drittanbieter sich ebenfalls an Angular anpassen, ihre eigene Dokumentation aktualisieren und möglicherweise unterstützende Ressourcen bereitstellen.

Aufgrund dieser Faktoren sollte primär die Vorbereitung des Projekts begonnen werden. Mit den empfohlenen Maßnahmen lässt sich die Struktur verbessern und eine optimale Basis für die Migration herstellen. Die Vorbereitung ist in den meisten Bereichen unabhängig von Angular und kann daher nach eigenem Belieben durchgeführt werden.

Es lässt sich festhalten, dass ein gewisser Zeitraum abgewartet werden sollte, bis Angular und

das zugehörige Ökosystem Stabilität erreicht haben. Daraus entstehen durch die Community mit der Zeit viele weitere Ressourcen, auch in Bezug auf die Migration zu Angular und speziell die Ablösung der alten Ionic Version. Außerdem müssen Angular spezifische Abhängigkeiten durch kompatible Versionen ausgetauscht werden. Solange diese nicht verfügbar sind lässt sich die Migration nicht abschließen. Alternativ kann ein vollständiger Ersatz für veraltete Abhängigkeiten gesucht werden.

Am 9. August 2016 wurde Release Candidate 5 von Angular veröffentlicht.<sup>110</sup> Aus zeitlichen Gründen können die Änderungen dieser Version nicht mehr in dieser Arbeit berücksichtigt werden (Siehe 1.6). Im nächsten Schritt sollten daher die Neuerungen hinsichtlich des Modulsystems und des Angular Compilers näher betrachtet werden.

### 5.3 Reflexion

Diese Arbeit wird für die Webabteilung von SMARTCRM in zwei Aspekten hilfreich sein. Zum einen arbeitet sie fundiert die Neuerungen des Angular-Frameworks auf und hebt Änderungen hervor. Das Team kann dadurch lernen in welche Richtung sich Angular bewegt und welche Vorteile daraus für das Projekt entstehen können. Außerdem wurde eine Grundlage für die Migration der Anwendung geschaffen. Die Ergebnisse dieser Arbeit dienen als konkrete Hilfestellung, wie SMARTCRM.Web zukünftig ausgerichtet werden kann. Darüber hinaus ist die Analyse alternativer Frameworks interessant für potenzielle weitere Webprojekte der Firma.

Unterschätzt wurde von mir der Aufwand der praktischen Umstellung innerhalb des Webprojekts. Insbesondere die hohe Komplexität in Kombination mit dem jungen Alter von Angular verhindern eine triviale Migration. Die Dokumentation von Angular ist noch nicht vollständig und das Framework selbst hat während dem Erstellen der Arbeit grundlegende Änderungen erhalten. Dennoch konnte ein Ablaufplan für die Migration des Projekts erstellt werden. Aufbauend auf dieser Strategie lässt sich in Zukunft die Anwendung mit der notwendigen Sorgfalt und Investition der Zeit auf die neuste Version von Angular umstellen.

---

<sup>110</sup> Abgerufen am 25. August 2016 von <http://angularjs.blogspot.de/2016/08/angular-2-rc5-nmodules-lazy-loading.html>

## Literaturverzeichnis

- Abed, R. (18. August 2015). *Hybrid vs Native Mobile Apps – The Answer is Clear*. Abgerufen am 12. Juli 2016 von Y Media Labs: <http://www.ymedialabs.com/hybrid-vs-native-mobile-apps-the-answer-is-clear/>
- Ackermann, P. (2015). *Professionell entwickeln mit JavaScript*. Rheinwerk.
- Alman, B. (15. November 2010). *Immediately-Invoked Function Expression (IIFE)*. Abgerufen am 25. Juli 2016 von Ben Alman Blog: <http://benalman.com/news/2010/11/immediately-invoked-function-expression/>
- Angular University. (21. Juni 2016). *Introduction to Angular 2 Forms - Template Driven and Model Driven*. Abgerufen am 28. Juli 2016 von Angular University Blog: <http://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/>
- Atwood, J. (17. Juli 2007). *The Principle of Least Power*. Abgerufen am 19. Juli 2016 von Coding Horror: <https://blog.codinghorror.com/the-principle-of-least-power/>
- Bak, L., & Lund, K. (25. März 2015). *Dart for the Entire Web*. Abgerufen am 25. Juli 2016 von Dart News & Updates: <http://news.dartlang.org/2015/03/dart-for-entire-web.html>
- Bidelman, E. (28. August 2013). *Custom Elements - Defining new elements in HTML*. Abgerufen am 21. Juli 2016 von HTML5 Rocks: <http://www.html5rocks.com/en/tutorials/webcomponents/customelements/>
- Bidelman, E. (11. November 2013). *HTML Imports - Include for the web*. Abgerufen am 21. Juli 2016 von HTML5 Rocks: <http://www.html5rocks.com/en/tutorials/webcomponents/imports/>
- Black, N. (20. Juli 2016). *A dedicated team for AngularDart*. Abgerufen am 25. Juli 2016 von Angular Blog: <http://angularjs.blogspot.de/2016/07/a-dedicated-team-for-angulardart.html>
- Börjesson, N. (18. April 2016). *Recent developments*. Abgerufen am 18. August 2016 von Angular Schema Form Wiki: <https://github.com/json-schema-form/angular-schema-form/wiki/Recent-developments>
- Bucholtz, D. (5. August 2016). *Announcing Ionic 2, Beta 11*. Abgerufen am 18. August 2016 von Ionic Blog: <http://blog.ionic.io/announcing-ionic-2-beta-11/>
- Clemons, E. (27. Dezember 2015). *JavaScript Fatigue*. Abgerufen am 3. August 2016 von Medium: <https://medium.com/@ericclemons/javascript-fatigue-48d4011b6fc4>
- Crockford, D. (2008). *JavaScript: The Good Parts*. O'Reilly.
- Drifty Co. (21. Juli 2016). *Core Concepts*. Abgerufen am 18. August 2016 von Ionic Docs: <http://ionicframework.com/docs/v2/getting-started/concepts/>
- Duffy, J., & Brown, J. (Juli/August 2016). Aurelia: An Introduction. *Code*, S. 64-69.
- Ecma International. (Juni 2015). *ECMAScript 2015 Language Specification*. Abgerufen am 28. Juli 2016 von Ecma International: <http://www.ecma-international.org/ecma-262/6.0/>
- Eschweiler, S. (2016). *Angular 2 - A Practical Introduction to the new Web Development Platform*. Leanpub.
- Exbrayat, C. (2016). *Become a ninja with Angular 2*. Ninja Squad.

- Fowler, M. (26. Juni 2005). *Inversion of Control*. Abgerufen am 1. August 2016 von Martin Fowler Blog: <http://martinfowler.com/bliki/InversionOfControl.html>
- Fowler, M. (2012). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.
- Gechev, M. (2016). *Switching to Angular 2*. Packt Publishing Ltd.
- Google. (2016). *Angular Website*. Abgerufen am 22. Juli 2016 von Angular: <https://angular.io/>
- Google. (2016). *AngularJS Documentation*. Abgerufen am 25. Juli 2016 von AngularJS - Superheroic JavaScript MVW Framework!: <https://docs.angularjs.org>
- Green, B. (15. Dezember 2015). *Angular 2 Beta*. Abgerufen am 12. Juli 2016 von Angular Blog: <http://angularjs.blogspot.de/2015/12/angular-2-beta.html>
- Green, B., & Minar, I. (5. März 2015). *ng-conf 2015 Day 1 Keynote*. Abgerufen am 5. August 2016 von YouTube: <https://www.youtube.com/watch?v=QHulaj5ZxbI&list=PLOETEcp3DkCoNnlhE-7fovYvqwVPrRiY7&t=721>
- Green, B., & Seshadri, S. (2013). *AngularJS*. O'Reilly.
- Hannah, J. (9. April 2015). *Choosing the Right JavaScript Framework for the Job*. Abgerufen am 3. August 2016 von Lullabot: <https://www.lullabot.com/articles/choosing-the-right-javascript-framework-for-the-job>
- Haverbeke, M. (2014). *Eloquent JavaScript*. No Starch Press.
- Hevery, M. (28. September 2009). *Hello World, <angular/> is here*. Abgerufen am 21. Juli 2016 von Miško Hevery Blog: <http://misko.hevery.com/2009/09/28/hello-world-angular-is-here/>
- Hus, M. (13. August 2015). *The Road to Angular 2.0 part 6: Migration*. Abgerufen am 11. August 2016 von Don't Panic: <http://dontpanic.42.nl/2015/08/the-road-to-angular-20-part-6-migration.html>
- Krassmann, K. (3. Februar 2015). *App-Entwicklung leicht gemacht: Mit Ionic, AngularJS und ngCordova zur mobilen Applikation*. Abgerufen am 11. Juli 2016 von t3n Magazin: <http://t3n.de/magazin/entwicklung-ionic-237246/>
- Kröner, P. (28. Mai 2014). *Web Components erklärt*. Abgerufen am 18. Juli 2016 von Peter Kröner - Blog: <http://www.peterkroener.de/web-components-erklaert-teil-1-was-sind-web-components/>
- Krypczyk, V. (Juli 2016). Mehr als ein Facelift - Software migrieren. *Dotnetpro*, S. 47-51.
- LePage, P. (4. Februar 2016). *Your First Progressive Web App*. Abgerufen am 18. Juli 2016 von Google Developers: <https://developers.google.com/web/fundamentals/getting-started/your-first-progressive-web-app/?hl=en>
- Lerner, A., Coury, F., Murray, N., & Taborda, C. (2016). *ng-book 2*. Fullstack.io.
- Malik, S. (November 2015). TypeScript: The Best Way to Write JavaScript. *Code*, S. 22-28.
- Malik, S. (Mai/Juni 2016). AngularJS 2. *Code*, S. 18-21.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.
- Martin, R. C. (8. Mai 2014). *The Single Responsibility Principle*. Abgerufen am 28. Juli 2016 von 8th Light Blog: <https://8thlight.com/blog/uncle-boleyn/2014/05/08/the-single-responsibility-principle.html>

- bob/2014/05/08/SingleReponsibilityPrinciple.html  
Microsoft. (10. Februar 2012). *The MVVM Pattern*. Abgerufen am 19. Juli 2016 von Microsoft Developer Network: <https://msdn.microsoft.com/en-us/library/hh848246.aspx>
- Minar, I. (14. Dezember 2013). *AngularJS 1.3: a new release approaches*. Abgerufen am 22. Juli 2016 von Angular Blog: <http://angularjs.blogspot.de/2013/12/angularjs-13-new-release-approaches.html>
- Mobile Zeitgeist. (9. Juli 2015). *Hybrid vs Nativ: Die zweite Hybrid App Welle*. Abgerufen am 11. Juli 2016 von Mobile Zeitgeist: <http://www.mobile-zeitgeist.com/2015/07/09/hybrid-vs-nativ-die-zweite-hybrid-app-welle/>
- Motto, T. (2016). *Angular 1.x styleguide (ES2015)*. Abgerufen am 28. Juli 2016 von GitHub: <https://github.com/toddmotto/angular-styleguide>
- Mozilla Developer Network. (14. Oktober 2015). *HTML Template*. Abgerufen am 18. Juli 2016 von Mozilla Developer Network: <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template>
- Precht, P. (24. Oktober 2015). *Upgrading Apps to Angular 2 using NgUpgrade*. Abgerufen am 9. August 2016 von Thoughtram: <http://blog.thoughtram.io/angular/2015/10/24/upgrading-apps-to-angular-2-using-ngupgrade.html>
- Resig, J., & Bibeault, B. (2013). *Secrets of the JavaScript Ninja*. Manning.
- Riepe, P. (2. April 2015). *Warum native wenn es auch hybrid geht?* Abgerufen am 12. Juli 2016 von Computerwoche: <http://www.computerwoche.de/a/warum-native-wenn-es-auch-hybrid-geht,3096411>
- Rimmer, J. (November 2015). *Tracking the progress of Web Components through standardisation, polyfillification and implementation*. Abgerufen am 18. Juli 2016 von Are We Componentized Yet?: <http://jonrimmer.github.io/are-we-componentized-yet/>
- Russell, A., Song, J., & Archibald, J. (25. Juni 2015). *Service Worker Specification*. Abgerufen am 18. Juli 2016 von W3C: <https://www.w3.org/TR/service-workers/>
- Sayar, R. (23. November 2015). *Top JavaScript Frameworks, Libraries and Tools and When to Use Them*. Abgerufen am 19. Juli 2016 von SitePoint: <https://www.sitepoint.com/top-javascript-frameworks-libraries-tools-use/>
- Schmitz, D., & Georgii, D. P. (2016). *Practical Angular 2*. Leanpub.
- Sengstacke, P. (25. April 2016). *JavaScript Transpilers: What They Are & Why We Need Them*. Abgerufen am 25. Juli 2016 von Scotch.io: <https://scotch.io/tutorials/javascript-transpilers-what-they-are-why-we-need-them>
- SMARTCRM GmbH. (2016). *CRM Software für Industrie und Handel*. Abgerufen am 11. Juli 2016 von SMARTCRM: <http://smartcrm.de/>
- Stefanov, S. (2010). *JavaScript Patterns*. O'Reilly.
- Steyer, M. (Juli 2016). Progressive Web-Apps mit Angular 2 und Service Worker. *Windows Developer*, S. 26-31.
- Steyer, M. (Mai 2016). Tutorial: Moderne Apps für alle Plattformen. *Windows Developer*, S. 54-59.
- Steyer, M. (April 2016). Tutorial: Wartbare Frontends entwickeln - JavaScript mit Angular 2 beherrschbar machen. *Windows Developer*, S. 40-45.

- Stoychev, V. (20. April 2016). *What are the key difference between ReactNative and NativeScript?* Abgerufen am 14. Juli 2016 von Quora: <https://www.quora.com/What-are-the-key-difference-between-ReactNative-and-NativeScript/answer/Valentin-Stoychev>
- Styoanov, D. (13. März 2014). *RxJS*. Abgerufen am 1. August 2016 von Reactive Extensions: <https://xgrommx.github.io/rx-book>
- Svanidze, D. (1. Juli 2014). *Native Apps vs. Web Apps vs. Hybride Apps*. Abgerufen am 13. Juli 2016 von APP3null: <https://app3null.com/native-hybride-web-apps/>
- Taylor, P. (11. November 2015). *RxJS Evolved*. Abgerufen am 29. Juli 2016 von SlideShare: <http://www.slideshare.net/trxclnt/rxjs-evolved>
- Vogel, O., Arnold, I., Chughtai, A., Ihler, E., Kehrer, T., Mehlig, U., & Zdun, U. (2009). *Software-Architektur*. Spektrum.
- Williams, O. (14. Januar 2016). *The most popular JavaScript library, jQuery, is now 10 years old*. Abgerufen am 19. Juli 2016 von The Next Web: <http://thenextweb.com/dd/2016/01/14/the-most-popular-javascript-library-jquery-is-now-10-years-old>

## Bibliografie

**Angular Upgrading from 1.x** – <https://angular.io/docs/ts/latest/guide/upgrade.html>

**Angular Migration Guide by Todd Motto** – <http://ngmigrate.telerik.com/>

**Angular Upgrade Strategies from Angular 1.x** –

<http://developer.telerik.com/featured/angular-2-upgrade-strategies-angular-1-x>

**Learn Angular** – <http://learnangular2.com/>

**Thoughtram Blog** – <http://blog.thoughtram.io/>

**Angular Production Build** – <http://blog.mgechev.com/2016/06/26/tree-shaking-angular2-production-build-rollup-javascript>

**Angular by Example** – <https://medium.com/baqend-blog/angular-2-by-example-e85a09fa6480>

**Ionic 2** – <http://ionicframework.com/docs/v2>

## Styleguides

**John Papa** – <https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md>

**Todd Motto** – <https://github.com/toddmotto/angular-styleguide>

## Anhang

Für die Recherche und Vorbereitung zur Migration wurde ein Repository auf der Entwickler Plattform *GitHub* angelegt. Darin befinden sich Projekte zu Angular, React und eine beispielhafte Migration einer AngularJS-Anwendung.

Auf der beigelegten CD befinden sich der Inhalt des Repository und die Bachelorarbeit selbst als PDF-Dokument.

Die URL für das Repository lautet:

<https://github.com/marc1404/Bachelorarbeit>

## Listings

Listing 43: Routing Definition der Webapplikation

```
1 // appConfig.js
2 angular.module('smartcrm')
3     .config(function($stateProvider, ...) {
4         // ...
5         $stateProvider
6             .state('home', {
7                 url: '/',
8                 templateUrl: 'templates/layout/home/home.html'
9             })
10            .state('page', {
11                url: 'page/:pageName/{index:int}',
12                parent: 'home'
13            });
14        // ...
15    });

```

Quelle: Eigene Darstellung

Listing 44: Bootstrap-Prozess einer Angular-Anwendung

```
1 // app.module.ts
2 import { NgModule } from '@angular/core';
3 import { BrowserModule } from '@angular/platform-browser';
4 import { AppComponent } from './app.component';
5
6 @NgModule({
7   imports: [ BrowserModule ],
8   declarations: [ AppComponent ],
9   bootstrap: [ AppComponent ]
10 })
11 export class AppModule {}
12
13 // main.ts
14 import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
15 import { AppModule } from './app.module';
16
17 platformBrowserDynamic().bootstrapModule(AppModule);
```

Quelle: Google (2016)<sup>111</sup>

---

<sup>111</sup> Abgerufen am 22. August 2016 von <https://angular.io/docs/ts/latest/quickstart.html>