FH JOANNEUM Gesellschaft mbH

Konzepte einer Web-App anhand des Beispiels einer Quiz-Marketing-App

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science in Engineering (BSc)

eingereicht am

Fachhochschul-Studiengang Software Design

Betreuer: DI Johannes Feiner

eingereicht von: Michael Mayer

Personenkennzahl: 1210418040

April 2015

Abstract

Mobile devices are not an upcoming trend, they are in any form clearly prevalent in our society. As a result, mobile app development has become popular for companies to serve their products state of the art. Although many companies aim at providing different apps for their customers, they usually face one major obstacle: due to different platforms for these devices such as iOS, Android and Windows Phone, an app has to be developed for each of them. To meet this discrepancy, in contrast of this discrepancy, HTML5 and CSS3 with JavaScript provide a cross platform frontend technology which enables the development of one web app available on all devices. This bachelor thesis deals with concepts of such mobile web apps by focusing on modern JavaScript frameworks making it possible to build structured web applications. The research provides insights into the building of modularized applications which ensure extensibility and testability. For demonstrating the concept of mobile web apps, a prototype with the AngularJS framework has been developed. In this context basic information about AngularJS is provided and a comparison to similar frameworks has been drawn. Generally, AngularJS as well as backbone.js provides a well fitted skeleton for structured and modularized application architecture. New features and technologies, like data binding, make it easier to separate the presentation logic from their actual representation. In conclusion, mobile web apps are worth considering when building cross platform apps. Nevertheless, many JavaScript-based web applications still have problems in terms of maintainability, testability, or extensibility. AngularJS or backbone.js can help to build these apps much more structured. Beyond question, architecture pays off, especially within a long application lifecycle.

Zusammenfassung

Smartphones und mobile Endgeräte sind kein aufstrebender Trend, sondern bereits Alltagsgegenstände unserer Gesellschaft. Die Mobile-App-Entwicklung wurde, als Ergebnis davon, für viele Unternehmen immer beliebter, um Ihre Produkte auf diesen Plattformen anzubieten. Viele Unternehmen stehen jedoch vor einem grundlegenden Hindernis: Um die App möglichst breit einsetzen zu können, muss diese für jede der 3 großen Plattformen iOS, Android und Windows Phone separat entwickelt werden. Dabei sind 3 verschiedene Programmiersprachen und Architekturen zu berücksichtigen. Im Gegensatz dazu bieten HTML5, CSS3 und JavaScript plattformunabhängige Frontend-Technologien, die es ermöglichen, eine App für nahezu alle mobilen Endgeräte zu entwickeln. Diese Bachelorarbeit fokussiert auf Konzepte einer mobilen Web-App und im Speziellen auf den Einsatz moderner Frameworks. Die Arbeit gibt Einblick in den Aufbau einer Web-App, um eine möglichst strukturierte und modulare Applikation im Sinne der Erweiterbarkeit und Testbarkeit zu schaffen. Um das Konzept zu erläutern, wurde in Form einer Quiz-Marketing-App ein Prototyp mit dem AngularJS-Framework entwickelt. In diesem Kontext wird das AngularJS-Framework im Detail vorgestellt und die verwandten Bibliotheken backbone.js und JavaScriptMVC damit verglichen. Sowohl AngularJS als auch beispielsweise backbone.js bieten ein Grundgerüst für eine möglichst modulare und strukturierte Architektur einer Web-App. Dabei machen es vor allem neue Konzepte wie Data-Binding einfacher, die Präsentationsschicht von der Domänenlogik zu trennen. Zusammengefasst sind Web-Apps definitiv eine Option für die Entwicklung einer plattformunabhängigen App. Dennoch haben viele JavaScript-basierte Applikationen Probleme in Bezug auf die Erweiterbarkeit und Testbarkeit. Bibliotheken wie AngularJS oder backbone.js können dabei helfen, die Applikationen strukturierter zu entwerfen. Darüber hinaus zahlt sich eine Investition in Architektur meist aus, vor allem bei Berücksichtigung eines langen Applikations-Lebenszyklus.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Diese Bachelorarbeit befasst sich mit den Konzepten moderner, modularer und strukturierter Web-Apps. Mobile Plattformen und Endgeräte sind aus der heutigen Zeit nicht mehr wegzudenken. Immer mehr Unternehmen bieten für ihre Applikationen und Dienste mittlerweile auch eine mobile Version an. Aus dem Paper von Alex Nicolaou (2013) geht hervor, dass der Datenverkehr auf mobilen Geräten drastisch steigt. Die Prognose für 2017 soll erstmal die 10-Exabyte-Marke übersteigen. Daran ist deutlich zu erkennen, dass mobile Software und Apps längst keine Marktlücke mehr sind.

1.1 Motivation

Die Marketingabteilung des steirischen Unternehmens INTECO GmbH[1] hat sich strategisch dazu entschieden, bei Messeauftritten die modernen Aspekte einer App für sich zu nutzen. Besonders im Bereich von Klein- und Mittelunternehmen stellt sich oft die Frage nach der Wahl der richtigen mobilen Plattform. Für die Entwicklung von Apps für die drei wichtigsten Plattformen iOS, Android und Windows Phone übersteigen die Kosten oft den Nutzen. Das Unternehmen beherbergt unter anderem selbst eine Softwareabteilung für ihre Produkte; das Budget für eine neue App ist darin jedoch nicht enthalten. Die Alternativen zur Entwicklung einzelner nativer Apps zählen Hybrid-Apps, interpretierte Apps, ge-

[1] INTECO special melting technologies GmbH

nerierte Apps sowie Web-Apps (vgl. Xanthopoulos & Xinogalos 2013, S. 215ff.). Alle Alternativen bieten Plattformunabhängigkeit, wobei in den letzten Jahren durch HTML5 die Web-App-Sparte stark forciert wird. Die Europäische Union hat mit *MobiWebApp* ein eigenes Förderprogramm für Web-Applikationen geschaffen. Dies soll sowohl die Standardisierung steigern, als auch europäische Unternehmen in diesem Segment stärken (vgl. Forgue & Hazaël-Massieux 2012, S. 255ff.). Aus den genannten Gründen für Web-Apps hat sich INTECO im Zuge der Quiz-Marketing-App für die Webtechnologie entschieden. Daraus entstand die Motivation für diese Bachelorarbeit, ein nachhaltiges Konzept für Web-Apps zu entwerfen.

1.2 Problem

Die Problemstellung ergibt sich aus der raschen Entwicklung des Webs. Frühere Konzepte führen in der heutigen Zeit zu wenig Akzeptanz. Statische Webseiten mit einem hohen Grad an Wartezeit verärgern zusehends die AnwenderInnen. Dadurch ergeben sich für moderne Web-Applikationen neue Anforderungen an die Softwarearchitektur. Viele Web-Applikationen, welche mit JavaScript-Bestandteilen dynamischer wurden, haben durch historisches Wachstum viele Schwierigkeiten in den genannten Bereichen. Vor allem die Erweiterbarkeit und die fehlende Testbarkeit führten oft langfristig zu hohen Kosten. Durch geringe Struktur im Projekt und im Quellcode wurden einige Web-Apps zu wenig wartbaren Softwareprodukten. Diesem Problem soll die Arbeit mit einem Konzept und Lösungsansätzen entgegenwirken.

1.3 Zielsetzung

Die Arbeit verfolgt das Ziel einer modernen Architektur, sowie eines durchdachten Konzepts einer Web-App. In welchem Maß können durch neue Konzepte und Architekturen, sowie der Einsatz von Frameworks wie AngularJS die Software flexibel, erweiterbar und testbar gestaltet werden? - Diese Frage ist der Kernaspekt

dieses Konzepts. Die Zielsetzung wird mit Hilfe der Prototypenentwicklung ausgearbeitet. Dafür wird eine Web-App vom Beginn der Konzepterstellung bis hin zu den letzten Tests in jeder Phase dokumentiert. Die Auswertung des Konzepts bzw. der erstellten Architektur geschieht anhand der Testbarkeit und Erweiterbarkeit des Prototypen. Dies sind wichtige Bestandteile der Codequalität. Gemessen werden die Ziele unter anderem durch Verwendung sogenannter Softwaremetriken. Dies sind Kennzahlen, welche Auskunft über die Beschaffenheit des Quellcodes bringen.

Für die Erreichung der genannten Struktur und Organisation des JavaScript-Projekts gibt es mittlerweile eine große Auswahl an Bibliotheken und Frameworks. Viele davon werben mit diesen Funktionalitäten, eines davon ist das AngularJS-Web-Framework. Dieses wird für die Entwicklung des Prototypen verwendet und erhält besondere Aufmerksamkeit. Im Literaturteil wird auf die verwendeten Technologien eingegangen, sowie AngularJS näher erläutert. Des Weiteren werden backbone.js und JavaScriptMVC im Vergleich zu AngularJS kurz präsentiert. Aufbauend darauf wird in Kapitel 3 die Umsetzung des Prototyps mit den Ergebnissen der Arbeit dargestellt.

Kapitel 2

Stand der Technik

Die Arbeit beruht in ihrem Grundaufbau auf der Methodik der Prototypenentwicklung. Dabei entsteht in Form einer Web-Applikation eine ausführbare Software, welche hinsichtlich der Problemstellung und Zielsetzung ausgewertet wird. Dieser Web-Applikation liegen grundsätzlich die Technologien HTML und JavaScript zugrunde. Der Prototyp soll letztlich einer modernen Web Applikation entsprechen, welche sich für die AnwenderInnen sehr ähnlich zu einer nativen App verhalten soll. Um dies zu erreichen, kommen während der Entwicklung Konzepte wie Single-Page-Applications zum Einsatz. Als grundlegende Architektur für den Prototypen wird auf AngularJS gesetzt. Das folgende Kapitel bietet eine grundlegende Einführung sowie Verweise auf weiterführende Literatur der genannten Technologien. Zusätzlich wird das AngularJS Framework im letzten Unterkapitel mit verwandten Bibliotheken wie backbone.js oder JavaScriptMVC verglichen.

2.1 HTML5 und JavaScript

Beim Entwurf einer modernen Rich-Internet-Application (RIA) kommt man um zwei Technologien nicht herum: HTML5 und JavaScript. HTML5 bezeichnet dabei den aktuellen Standard des World-Wide-Web-Consortium (W3C). Durch JavaScript-Code, der im Browser ausgeführt wird, bekommt eine sonst statische Webseite ihre Dynamik.

HTML ist die Beschreibungssprache der Benutzeroberfläche im Web. Das W3C

formuliert dazu Standards für HTML als auch für CSS. Durch diese können sich Entwickler darauf verlassen, dass der HTML-Quelltext immer auf dieselbe Weise interpretiert wird. Die Standards schaffen somit eine einheitliche Sicht der Dinge für Entwickler und für Browser-Hersteller (vgl. Clark, Studholme, Murphy & Manian 2012, S. 2f.). HTML5 ist die aktuelle Spezifikation der Beschreibungssprache für Web-Benutzerschnittstellen. Der Spezifikation rund um HTML4 fehlen schlicht einige Bestandteile, um die neuen Anforderungen des Web abzudecken. Webseiten wurden immer dynamischer mit einer stärkeren Verteilung Richtung Client. Erst durch diese fehlende Dynamik boomten Technologien wie Flash und JavaScript. HTML5 soll nun diese Lücke schließen, und somit den Vorteilen von Flash oder Silverlight Widerstand leisten. Mit eigenen Tags für <audio> oder <video> lassen sich nun Multimedia-Elemente in HTML integrieren. Die Spezifikation verbessert aber auch maßgeblich die Beschreibung des *User Interface* (UI) selbst. Weitere neue Tags sorgen dafür, dass der Code intuitiver zu lesen ist. Aus einer Verkettung vieler Elementen wird eine Hierarchie aus <header> und <nav>, bei der bereits die Tags selbst sprechende Namen besitzen. Ebenso wurden neue Features etabliert, wie etwa neue Möglichkeiten der Verbindung zum Server oder im Bereich Grafik durch das <canvas> Element. Bei all den Neuerungen wurde dennoch darauf geachtet, dass die Abwärtskompatibilität zur HTML4-Spezifikation erhalten bleibt (vgl. Clark et al. 2012, S. 8ff.). Dadurch können vor allem auch bestehende Projekte um neue Eigenschaften von HTML5 erweitert werden, ohne dabei alles umstellen zu müssen.

Im Bereich der Eingabeformulare gibt es sowohl verbesserte Möglichkeiten der Validierung als auch neue Eingabeelemente. Zu beachten ist dabei, dass diese neuen Typen oft auf den ersten Blick keine erkennbare Auswirkung haben. Beispielsweise gibt es keinen Unterschied zwischen einem Eingabefeld vom existenten Typ text und dem neuen HTML5-Typ number. Beide Eingabefelder werden im Browser ident dargestellt. Würde man jedoch im Zahlenfeld einen Text eingeben, würde das Zahlenfeld darauf hinweisen, dass eine Zahl einzugeben ist. Der weitere Vorteil versteckt sich in der Verwendung von mobilen Endgeräten. Dabei würde die Tastatur eines Smartphones, im Fall des Zahlenfeldes, nur mit Nummern aus-

gestattet sein (vgl. Lubbers, Albers & Salim 2011, S. 195ff.). Eine ausführliche Referenz zu HTML bietet das Buch *The ultimate HTML reference* (Lloyd 2008). Cascading Style Sheets (CSS) wurden entworfen, um die Gestaltung von Webseiten so weit als möglich von ihren Inhalten zu trennen. Dabei werden in CSS Anweisungen formuliert, welche das Aussehen der Elemente im Browser anpassen. Der aktuelle Standard des W3C ist dabei CSS3. Dieser bringt einige Neuerungen im Bereich der Selektoren. Elemente im HTML-DOM zu finden und zu gestalten soll damit vereinfacht werden. Wo früher viele neue class Attribute erstellt wurden, können Elemente nun über neue Selektoren einfacher angesprochen werden (vgl. Hogan 2013, S. 67f.). Besonders im weiteren Verlauf der Arbeit kommt CSS3 beim Design des Prototyps zum Einsatz. Durch die Verwendung von Twitter Bootstrap[1] wird der Ansatz „Mobile First“ unterstützt. Responsive Webdesign ist Grundvoraussetzung für die Entwicklung von modernen Web-Applikationen, sowohl auf dem Smartphone als auch auf einem Tablet. Bootstrap stellt dabei die CSS-Datei, basierend auf CSS3, bereit.

JavaScript spielt für diese Arbeit eine große Rolle. Immer mehr RIAs gehen dazu über, die Rechenleistungen der Client-Computer zu nutzen. Eine Möglichkeit diese Web-Applikationen zu erstellen, ist einen Teil der Logik in den Client-Bereich zu verschieben. Dies wird in Form von einer JavaScript Anwendung in Verbindung mit HTML und CSS bereitgestellt.

Um einen Einblick in die Sprache und deren Idee zu bekommen, sollte eine Reise ins Jahr 1995 nicht fehlen. Der Entwickler Brendan Eich sollte für Netscape pünktlich zum Release 2.0 eine Sprache entwickeln, welche im neuen Browser ausführbar sei. Netscape sah zu dem Zeitpunkt im Zusammenhang mit Microsofts Windows 95 das Web als Zukunft, selbst für Betriebssysteme. Mit den Kenntnissen aus Java und seinen Vorteilen wollte Netscape eine leichtgewichtige Sprache für nicht-professionelle Entwickler schaffen. Charles Severance (2012), Universität Michigan, äußerte in seinem Artikel auch die Möglichkeit, dass JavaScript künftig die Programmiersprache für Mobile- und Desktop-Apps werden könnte (vgl. Severance 2012).

[1] http://getbootstrap.com/getting-started/

Die zugehörige Sprachedefinition zu JavaScript ist ECMAScript in der aktuellen Version 5.1. Darin wird eine objektorientierte Programmiersprache beschrieben, die ihren Zweck in Berechnungen und Manipulationen innerhalb bestehender Laufzeitumgebungen hat. Um die Möglichkeiten von Web-Applikationen clientseitig zu verbessern, wurde ECMAScript als die Scripting-Sprache des Webs entwickelt. Browser stellen dabei die Host-Umgebung zur Verfügung, in welcher die Script-Instanzen ausgeführt werden. Dabei bedient sich die Sprache auch aus Teilen der Java oder Scheme Sprachspezifikation. Der wichtigste Unterschied zu Java ist aber, dass keine Klassen im herkömmlichen Sinne entwickelt werden. Die Objektorientierung in ECMAScript und Vererbung ist mittels *prototype-inheritance* abgebildet. Das heißt, dass über die Eigenschaft prototype in jedem *Constructor* eines Objekts die Vererbung abgebildet ist. Vergleichend bedeutet dies in den Sprachen Java oder C#: Vererbung ist Teil der Struktur, Methoden sind Teil von Klassen und aktuelle Zustände sind Teil von Instanzen. Bei ECMAScript sind Zustände und Methoden Teil von Objekten und Zustände, Struktur als auch Verhalten werden durch *prototype* vererbt (vgl. ECMA International 2011, S. 1ff.).
JavaScript, CoffeeScript oder TypedScript werden in diesem Kontext „Dialekte“ genannt. Alle haben jedoch die ECMAScript Sprachspezifikation im Kern enthalten. Durch die stetig steigende Bedeutung clientseitiger Web-Applikationen wird auch die Anzahl an JavaScript-Frameworks größer. Dies führt auch automatisch zum Bestreben, den Code strukturierter zu organisieren. Etwa 40 JavaScript Frameworks verstehen sich mittlerweile als Model-View-Controller (MVC) Bibliotheken. Oft werden sie auch als MV* Bibliotheken bezeichnet, da man sie keinem der bekannten Architekturmuster MVC, Model-View-ViewModel (MVVM) oder Model-View-Presenter (MVP) ganz zuordnen kann. (vgl. Hales 2012, S. 3f.). Eines dieser Frameworks ist *AngularJS*, worauf im Verlauf des Kapitels noch detailliert eingegangen wird.

2.2 Single Page Application

Durch die Entwicklung des Webs, sowohl in seiner Geschwindigkeit als auch in der User-Experience ist es auch an der Zeit, die Grundkonzepte von Web-Anwendungen zu überdenken. Was unterscheidet eine Webseite eigentlich von einer Web-Applikation? - Diese Frage blieb bislang noch unbeantwortet. Grundsätzlich zeichnet sich eine Webseite dadurch aus, statische Inhalte vom Webserver zu laden und anzuzeigen. Der Nachteil wird dabei schnell klar: Durch Links oder andere Aktionen auf der Webseite, wird eine Anfrage an den Server gestellt, welcher in Form einer neuen Webseite antwortet. Dadurch wird die ganze Seite neu geladen, obwohl dies sehr oft nicht notwendig wäre. Klassische MVC-Applikationen am Server funktionieren auf diese Art und Weise. In Zeiten der Apps und Mobile-Web-Apps findet jedoch ein solches Vorgehen nur mehr wenig Akzeptanz. BenutzerInnen wenden sich von Systemen ab, die eher mit Wartezeit als mit Benutzerfreundlichkeit bestechen. *Single-Page-Applications* (SPA) bieten dabei ein neues Konzept von Web-Applikationen. Ziel von SPAs ist es, eine Desktop-Applikation im Browser anzubieten. Der Server bleibt im neuen Konzept wichtiger Bestandteil, verändert sich aber in Richtung Datenlieferant (vgl. Mikowski & Powell 2013, S. 1ff.). Grundsätzlich besteht das Konzept aus einer HTML-Shell, die der Applikation ihren Rahmen gibt, und aus einer JavaScript-Applikation im Hintergrund. Navigation, oder in diesem Kontext auch Routing genannt, findet dabei im Client statt. Dabei werden sogenannte HTML-Templates innerhalb der Shell getauscht, ohne einen Request am Server ausgelöst zu haben. Viele JavaScript-Frameworks bauen auf diesem Konzept auf und unterstützen z.B. das clientseitige Routing. Fink (2010) zeigt beispielsweise in einer weiterführenden Literatur die Entwicklung einer SPA mit backbone.js und ASP.NET-Backend.

2.3 AngularJS

Das Angular Framework wurde von Google entworfen, um die Komplexität der Entwicklung moderner Web-Applikationen zu senken. Erfahrungen von Google-Mail oder Google-Maps sind in Angular eingeflossen. Dabei entstand letztlich ein

umfangreiches JavaScript-Framework. Während der Entwicklung einer Angular-App unterstützen verschiedenste Konzepte die Struktur und den Aufbau. Dieses Unterkapitel beruht besonders auf der Literatur von Green & Seshadri (2013) und Branas (2014) aus dem AngularJS Bereich, weitere Einzelquellen sind separat ausgewiesen.

Architektur in Angular ist genauso wichtig wie umstritten. Grundsätzlich lässt sich diese mit dem MVC Entwurf erklären. Letztendlich ist die genaue Zuordnung zu MVC oder einem seiner verwandten Entwürfe nicht ausschlaggebend. MVC und anderen Ausprägungsformen davon können, mit einem Verweis auf Erich Gamma et. al., auf die drei Design Patterns Composite, Strategy und Observer zurückgeführt werden (Gamma, Helm, Johnson & Vlissides 1995, vgl.). Dadurch gibt es viele Möglichkeiten, die Architektur zu implementieren. In der Community gibt es daher auch Diskussionen, welchem Architekturmuster nun AngularJS genau folgen würde. Im Angular-Team spricht man daher nur noch vom Model-View-Whatever (MVW) Design. Worauf es im Framework und beim Design ankommt, ist die saubere Trennung der Schichten. Dadurch wird ein modularer Aufbau, Skalierbarkeit und Testbarkeit erreicht. Die View bezeichnet dabei die HTML-Templates. Controller sind JavaScript-Objekte, welche jeweils hinter einer View liegen. Verbunden sind die beiden mit einem *Scope*-Objekt. Das Modell definiert Plain-Old-JavaScript-Object (POJO)-Objekte, welche die Daten der Domäne repräsentieren. Neben diesen Bestandteilen gibt es noch Services, Directives oder Filter.

Zur Architektur sollte auch die Verteilung des Quellcodes definiert werden. Dabei gibt es zumindest 4 Möglichkeiten, den Code zu organisieren.

1. Inline-Struktur

   Bei der Inline Variante wird eine app.js Datei direkt neben der index.html verwaltet. Diese Variante ist nur für kleinere Projekte oder in der Verwendung von Präsentationen zu empfehlen.

2. Stereotypen-Struktur

   Die Struktur eignet sich für kleinere Projekte mit überschaubarer Anzahl

an Komponenten (Controller, Services, Directives). Dabei werden alle Komponenten eines Types in einer Quelldatei entwickelt. Beispielsweise sind alle Controller in der controllers.js-Datei enthalten.

3. Spezifische Sterotypenstruktur

   Dabei handelt es sich um eine Erweiterung des Stereotypen-Ansatz, wobei jede Komponente in einer eigenen Quelldatei ist. Die Dateien werden nach Komponententyp in den entsprechenden Ordnern zusammengefasst. Diese Struktur kommt zum Einsatz, wenn die Größe des Projekts im Vergleich zur Stereotypen-Struktur zunimmt.

4. Domain-Struktur

   Für große Projekte wird oft auch der Domain-Ansatz empfohlen. Dabei werden die Dateien nicht mehr zugehörig ihren Schichten bzw. Komponententypen, sondern nach domänenspezifischen Features zusammengefasst. Dabei befinden sich in einem Ordner eines Features die Quelldateien aller Komponententypen von Controller, View bis Service.

```
Specific Stereotype Style

app/
    index.html
    css/
    js/
        app.js
        controllers/
            carController.js
            loginController.js
            personController.js
        services/
            carService.js
        directives/
            carDirective.js
    partials/
        login.html
        car.html
        person.html
        car-list.html
    index.html
```

```
Domain Style

app/
    index.html
    assets/
        css/
    application/
        app.js
        app.routes.js
    login/
        login.html
        login.controller.js
    car/
        car.html
        car-list.html
        car.controller.js
        car.service.js
        car.directive.js
    person/
        person.html
        person.controller.js
```

Abbildung 2.1: Vergleich zwischen Stereotypen- und Domänenstruktur

In Abbildung 2.1 werden die beiden größten Strukturen einander gegenübergestellt. Dabei wird verdeutlicht, wie in der Domänen-Struktur alle Bestandteile eines Features bzw. Moduls zusammengefasst werden.

Konzepte in Angular beziehen sich auf verschiedene Mechanismen der Applikation und werden im folgenden dargestellt:

1. Templates

   Ausgehend von einer HTML-Datei, welche als Rahmen (auch Shell genannt) verwendet wird, werden andere HTML-Bestandteile der Applikation am Client zusammengebaut. Dies unterscheidet sich maßgeblich von früheren Webseiten, bei denen der Server jeweils den ganzen HTML-Code zusammengebaut hat. Im Fall einer Angular-App hat der Server lediglich die HTML-Templates statisch zur Verfügung zu stellen. Diese werden zur Laufzeit innerhalb der Shell eingesetzt.

2. Data Binding

   Data-Binding ist ein wesentliches Konzept von AngularJS. Wiederum verglichen mit älteren Konzepten wurden am Server HTML und Daten in Form von *Strings* zusammengeführt und versandt. Im Verlauf konnte man nach einem AJAX-Request auch Updates über jQuery durchführen. Data-Binding zeichnet aus, dass View und Controller getrennt werden können. In einem HTML-Template wird mit spezieller Syntax angegeben auf welche Eigenschaften gebunden werden soll. Diese Eigenschaften werden vom Controller aus auf das `$scope` Objekt geschrieben.

```
1  ...
2  <div> {{ personName }} </div>
```

Listing 2.1: AngularJS Data-Binding Verwenduung

Dieses Codebeispiel zeigt die Verwendung von Data-Binding in einem Template. Der Name der Person wird angezeigt, sobald die Eigenschaft `personName` im Controller beschrieben wurde. Das bequeme an diesem Programmierkonzept ist, dass sowohl Updates auf dem UI automatisiert angezeigt

werden, als auch Änderungen in einem Eingabefeld automatisch in die gebundene Eigenschaft geschrieben werden (vgl. angularjs.org 2015). Dadurch entfallen Eventlistener auf diesen beiden Ebenen der Programmierung. Als Beispiel würde der Controller ein Update vom Server in $scope.personName schreiben und Angular übernimmt das Update auf dem UI.

3. Directives

   Direktiven in AngularJS erlauben es Entwicklern die HTML-Syntax um eigene Elemente zu erweitern. Einige dieser Direktiven werden oft angewandt ohne bemerkt zu werden. Jede Angular-Applikation wird durch das Attribut ng-app deklariert. Dies ist aber nicht Teil der HTML-Spezifikation, sondern eine Direktive von Angular selbst. Durch dieses Konzept ist es möglich, immer wieder benötigte Funktionalitäten in eine eigene Direktive auszulagern. Dabei könnte die Anzeige des Modells Car in einer car-view Direktive vereinheitlicht werden. Dies bedeutet, dass Direktiven selbst ein Template besitzen können, welches anstelle des Elementes im DOM angezeigt wird. Innerhalb der Direktive ist das Instanziieren eines eigenständigen Controllers ebenso möglich, wie das Definieren eines gekapselten Scopes. Der sogenannte *isolated-scope* vererbt keine Eigenschaften des übergeordneten *Scopes* (*Scope* des übergeordneten HTML-Element im Baum) und ist besonders für wiederverwendbare Komponenten sinnvoll.

Dependency Injection in Angular ist ein Konzept, das in diesem Abschnitt detaillierter beschrieben ist. In der objektorientierten Welt nennt man Abhängigkeiten zwischen Komponenten Kopplung. Die Kopplung kann dabei auch Gradmesser sein, wie gut die Applikation strukturiert und testbar ist. Je stärker Komponenten aneinander gebunden sind, desto schwieriger werden auch Änderungen. Anhand eines Beispiels wird dies verdeutlicht: Controller „A“ instanziiert selbsttätig ein Service „S“ mit dem new Operator. Controller „B“ bekommt Service „S“ im *Constructor* übergeben. Während eines Tests soll Service „S“ durch ein Spy-Objekt ersetzt werden. Controller „A“ wäre nicht testfähig, da Service „S“ instanziiert wird, während dem Controller „B“ das Spy-Objekt anstelle von Service

„S“ übergeben werden kann. Aus diesem Grund strebt man nach einem möglichst entkoppelten System. Das AngularJS Framework stellt dabei die Infrastruktur bereit. Die Bibliothek stellt dabei sicher, dass alle verwendeten Komponenten erzeugt, verwaltet und verteilt werden. Die Verteilung übernimmt der Dienst $injector. Der folgende Codeausschnitt zeigt 3 Arten der *Injection*.

```
1 ...
2 .controller('CarController', function ($scope, ...) { });
```

Listing 2.2: AngularJS Controller Definition - Variante 1

```
1 ...
2 .controller('CarController', ['$scope', ...,
3    function($scope, ...) {}]);
```

Listing 2.3: AngularJS Controller Definition - Variante 2

```
1 ...
2 .controller('CarController', );
3 CarController.$inject = ['$scope', ...];
4 function CarController($scope, ...) { };
```

Listing 2.4: AngularJS Controller Definition - Variante 3

Variante 1 ist dabei am einfachsten zu verwenden, wird jedoch auch nur bei kleinen Demoprojekten empfohlen. Gründe dafür sind Namenskonflikte oder die Verwischung der Namen durch *Minification* oder *Obfuscation*. Die Ursache liegt in der Funktionsweise des $injector-Dienstes. Dieser scannt die Funktionen und sucht nach dem Namen der Parameter die geeignete Instanz. Aus diesem Grund werden in Variante 2 die Namen der Abhängigkeiten in einem Array angegeben. Der letzte Eintrag im Array ist die Funktion selbst. Bei Variante 3 wird dem Injector durch Beschreiben der $inject Eigenschaft mitgeteilt, welche Abhängigkeiten benötigt werden (vgl. Knol 2013, S. 29ff.). Die beiden letzten empfohlenen Varianten schützen auch vor *Obfuscation* und *Minification* durch das Bekanntgeben der Abhängigkeiten als *String*.

Services in Angular sind Objekte welche in verschiedenen Komponenten wie

Controller, Direktive oder Filter verwendet werden können. Implementiert werden sie nach dem Singleton Pattern. Diese Dienste verfolgen primär die Aufgabe, Funktionalitäten losgelöst von Komponenten zu kapseln. Eine Möglichkeit dies zu nutzen sind Datenzugriffe auf das Backend. Diese werden in einem Service gekapselt und dem Rest der Applikation zur Verfügung gestellt. Dieses Vorgehen stellt wiederum selbst eine Reduktion von Abhängigkeiten dar. Anhand des Beispiels Datendienst versteckt das Service die Abhängigkeit auf den $http-Dienst von Angular vor seinen Aufrufern. Dadurch könnte $http-Dienst an einer zentralen Stelle ersetzt werden.

Asynchronous JavaScript and XML (**AJAX**) ist ebenso wichtiger Bestandteil einer Angular-App. Der Mechanismus wird verwendet, um in Web-Applikationen asynchron Daten vom Server nachzuladen. Bei Angular kapselt der erwähnte Dienst $http bereits die dafür notwendigen Low-Level-Aktivitäten für den Entwickler. Der Dienst bietet für die Kommunikation verschiedene Methoden, wie z.B. get(...), post(...) uvm., welche alle ein sogenanntes *Promise*-Objekt zurückliefern. Dieses Objekt ermöglicht die asynchrone Weiterverarbeitung nach dem zeitverzögertem Erhalt der Daten oder eines aufgetretenen Fehlers. Über die Methode then(...) kann dem *Promise*-Objekt eine *Callback*-Funktion für diesen Fall übergeben werden.

Für weiterführende Informationen zu AngularJS, ist auf die entsprechende einschlägige Literatur im Literaturverzeichnis zu verweisen.

2.4 Frameworks im Vergleich zu AngularJS

Neben dem Angular Projekt gibt es auch eine Vielzahl vergleichbarer Frameworks. Zwar ist meist der Umfang deutlich eingeschränkter als bei AngularJS, dennoch haben auch diese Bibliotheken ihre Einsatzberechtigung. Im folgenden Unterkapitel werden backbone.js und JavaScriptMVC vorgestellt.

backbone.js

Die Bibliothek backbone.js ist ein leichtgewichtiges MVC-Framework für besser strukturierte SPAs. Backbone.js fokussiert auf das Manipulieren und Abfragen

von Daten. Grundsätzlich ist die Bibliothek sehr modular aufgebaut, was sie sehr skalierbar macht. Durch ihre kompakte Größe gibt es viele Plug-Ins, welche die Applikation um Funktionalität erweitern können (vgl. Osmani 2012*a*, S. 3f.). Ein Bestandteil des Konzeptes sind die Modelle (Models). Diese repräsentieren die Domäne und werden, wie im folgenden Codebeispiel zu sehen ist, erzeugt.

```
1 ...
2 var Car = Backbone.Model.extend({});
```

Listing 2.5: backbone.js Model Definition

Dabei kapselt ein Model in backbone.js nicht nur die Eigenschaften einer Entität, sondern auch alle zugehörigen Funktionalitäten. Ein Model verfügt über getter- und setter-Methoden, sowie eine bereits integrierte Validierung. Auf jedem Model kann dafür eine Funktion zur Validierung gesetzt werden. Diese wird vor jedem .save() oder .set(...) aufgerufen. Allgemein formuliert könnte man sagen, das Modell bildet die Domänenlogik ab. Deshalb ist auch ein wichtiger Bestandteil des Modells der Datenzugriff. Mit einer einfachen REST-URL-Konfiguration können die Modelle mit .save() gespeichert werden. Den HTTP-Post-Aufruf an diese REST-URL übernimmt bereits backbone.js (vgl. Osmani 2012*a*, S. 28ff.). Views sind der zweite Bestandteil von backbone.js. Eine View repräsentiert darin jedoch keinen HTML-Code, sondern bildet die Ablauflogik ab (ähnlich einem Controller in AngularJS). Template-Mechanismen selbst sind in backbone.js nicht enthalten, hier wird von Osmani (2012) auf Underscore, Mustache oder jQuery Templating verwiesen. In der View wird dauf Ereignisse reagiert, welche vom UI oder vom Modell ausgelöst wurden. Mittels der render() Methode können Veränderungen an das UI propagiert werden.

Unit Tests für backbone.js sind ebenfalls möglich. Roemer (2013) verweist dabei auf Mocha, Chai und Sinon.js, wobei letzteres kein Testframework im eigentlichen Sinne, sondern eine Bibliothek mit nützlichen Spy- und Mock-Objekten ist. Grundlegend ist es in backbone.js durch die MVC-Architektur auch möglich, Modell und View getrennt voneinander zu testen.

Zusammengefasst bietet backbone.js eine Möglichkeit, die Daten in Form von

Modelle zu repräsentieren. Nach Manipulation in einer backbone-App werden die Modelle durch ein Restful-JSON-API mit dem Server synchronisiert (vgl. backbonejs.org 2014).

JavaScriptMVC

JavaScriptMVC (JMVC) ist ebenso ein MVC-Client-Framework, das vor allem auf jQuery aufsetzt. Im Vergleich zu backbone.js oder AngularJS will JMVC mehr als Gesamtlösung für den Client auftreten. Dies bedeutet das Framework umfasst mehrere Bibliotheken, welche gemeinschaftlich JavaScriptMVC bilden. In der Version 3.2 sind dies die folgenden Bibliotheken (vgl. Bednarski 2013, S. 7f.):

1. StealJS:
   zuständig für Builds und Abhängigkeiten für das Projekt

2. FuncUnit:
   Modul für Unit Tests sowie funktionale Tests im Projekt

3. jQueryMX:
   jQueryMX bildet, basierend auf jQuery, das eigentliche MVC-Framework innerhalb von JavaScriptMVC ab. Es bietet dabei umfänglich alles, was für den strukturierten Aufbau von größeren Projekten notwendig ist.

4. DocumentJS:
   zuständig für die Projektdokumentation

Ähnlich wie backbone.js bietet jQueryMX innerhalb des Frameworks eine Reihe von Möglichkeiten dies abzubilden. Mit der `$.Class()` Funktion können Klassen ähnlich zu Angular's Services entwickelt werden. Die Modelle der Domäne werden mit `$.Model()` erstellt und können mit einem Restful API verbunden und synchronisiert werden. Die Views in JavaScriptMVC bilden dabei den clientseitigen HTML Template Mechanismus ab. Innerhalb von jQueryMX kann zwischen 4 mitgelieferten Template Engines gewählt werden, wobei *EJS* die Standardvariante von JMVC ist. Organisiert werden Models und Views mit Controller, welche mit `$.Controller()` zu erstellen sind (vgl. Bednarski 2013, S. 49ff.).

2.5 Codequalität

Qualität an sich ist ein dehnbarer Begriff, vor allem wenn es dabei um Software geht. Viele Entwickler und Architekten von Softwaresystemen haben abweichende Vorstellungen davon. Einig ist man sich allerdings, dass die Codequalität absolut notwendig ist. Um die Qualität des Codes einigermaßen genormt messen zu können, wurden verschiedenste Kennzahlen definiert: die Softwaremetriken. Warum ein gut strukturiertes Konzept einer Applikationen im Verlauf ihrer Zeit wichtig ist, veranschaulicht Abbildung 2.2. Diese zeigt einen Lebenszyklus eines Softwaresystems. Dabei besonders zu beachten ist die Angabe der relativen Kosten der einzelnen Bereiche in Prozent. Bemerkenswert ist vor allem der verhältnismäßig große Prozentsatz nach Fertigstellung der ersten Version, dargestellt in der Maintenance Phase. Demzufolge machen die Wartung und Erhaltung eines Systems nach Abschluss der Entwicklung 67% der Gesamtkosten aus (vgl. Tripathy & Naik 2014, S. 83f.). Diese Angaben verdeutlichen die Bedeutung von Qualitätskriterien, welche besonders diese kostenintensive Phase beeinträchtigen. Da Client-Applikationen im Browser sowohl an Größe als auch an Bedeutung zunehmen, steigt gleichermaßen die Bedeutung ihrer Erweiterbarkeit und Testbarkeit. Der folgende Teilabschnitt dient der Begriffdefinition und Erklärung von Erweiterbarkeit und Testbarkeit. Des Weiteren wird erläutert, wie diese Eigenschaften des Codes gemessen werden können und welche Messinstrumente in Form von Metriken dafür verwendet weden können.

> *„Ein Programm heißt (leicht) erweiterbar, wenn es an Änderungen in der Spezifikation ohne übergroßen Aufwand angepaßt werden kann.“* (Rogat 1998)

Das Problem an der **Erweiterbarkeit**, wie Rogat in seiner Online-Publikation darstellte, ist, dass grundsätzlich jedes Softwaresystem an sich erweiterbar ist. Im schlimmsten Fall bedeutet dies nämlich eine Änderung des nahezu ganzen Systems, um den neuen Anforderungen zu entsprechen (vgl. Rogat 1998). Das Zitat wurde bewusst eingesetzt, um zu verdeutlichen, dass in Bezug auf Erweiterbarkeit ein großer Spielraum besteht. Erweiterbarkeit bedeutet, eine Software an neue Gegebenheiten, nach der Erstellung der ersten lauffähigen Version, anzupassen.

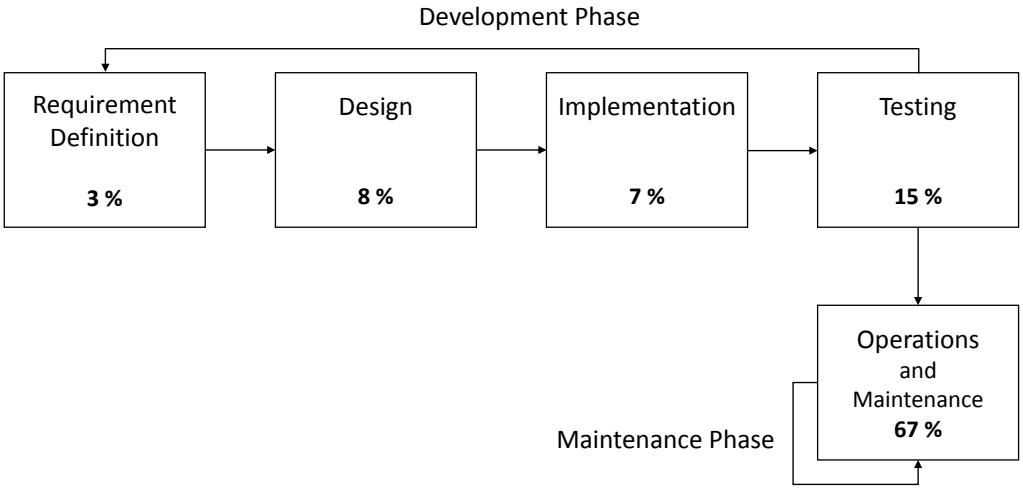

Abbildung 2.2: Typischer Software Application Lifecycle. (vgl. Tripathy & Naik 2014, S. 84)

Diese Gegebenheiten können zum einen neue Kundenanforderungen an das System sein, zum anderen aber auch Fehler des Systems. Fehlverhalten können sich oft erst durch den längeren Einsatz des Systems zeigen.
Wodurch entstehen eigentlich hohe Kosten in der Maintainability Phase? - Dieser Frage gehen viele Softwarehersteller nach. Grundsätzlich weiß man, dass in größeren Softwaresystemen Komponenten oft sehr eng miteinander gekoppelt sind. Dies führt automatisch dazu, dass durch die Änderung einer Komponente sehr viele andere Teile des Gesamtsystems angepasst werden müssen. Um dies beispielhaft zu erläutern, stelle man sich ein Softwareprogramm vor, welches von einem Internetdienst Wetterdaten lädt. Nun verändert sich eine Rahmenbedingung (Veränderung durch außen) und der Wetterdienst ist auf einer neuen Internetadresse erreichbar. Durch wenig Kapselung muss nun im gesamten System nach Zugriffen auf diesen Dienst gesucht werden und die Internetadresse getauscht werden. Aus diesen Gründen ist davon auszugehen, dass die Zerlegung des Monolithen in modulare Bestandteile die Änder- sowie Erweiterbarkeit erhöhen. Um diese Eigenschaften der Software zu messen, kann man sich dem Instrument der Softwaremetrik bedienen. Softwaremetriken generieren Kennzahlen aufgrund der Beschaffenheit des Quellcodes. Die JavaScript-Web-Applikation wird mittels *JSComplexity*[3]

getestet und ausgewertet. JSComplexity generiert dabei eine Vielzahl an Kennzahlen. Die für die Arbeit relevanten Metriken sind logical Lines-of-Code (LOC), die zyklomatische Komplexität sowie der Maintainability-Index. Die LOC (Quellcode ohne Leerzeilen) gibt dabei einen ersten Überblick über die Projektgröße an sich. Die zyklomatische Komplexität ist kurz gefasst die Anzahl der möglichen Pfade durch eine Methode. Die Kennzahl kann daher einen Einblick über die Komplexität des Projektes geben. Der Maintainability Index, im Jahr 1991 von Paul Oman und Jack Hagemeister entwickelt, ist eine Erweiterung der Halstead Komplexitätskennzahlen. Für Details zu den einzelnen Softwaremetriken ist auf entsprechende Literatur zu verweisen (Fenton & Bieman 2014) (Oman 1991). Für die späteren Ergebnisse der Arbeit ist es wichtig, die Zahlen zu interpretieren. Bei der zyklomatischen Komplexität verhält es sich umgekehrt proportional (je geringer der Wert desto besser für die Software). Der Maintainability-Index hat eine Skala von negativ unendlich bis 171, wobei höhere Werte für höhere Wartbarkeit stehen. (vgl. Oman 1991).

„Testbarkeit ist der Grad, zu dem ein Softwareprodukt das Testen (in einem bestimmten Testkontext) ermöglicht.“ (Jungmayr 2005)

Diese Definition von Dr. Jungmayr beruht auf der Definition nach ISO-9126, wonach **Testbarkeit** den Aufwand beschreibt, der zur Prüfung der veränderten Software notwendig ist. Unit-Tests sind eine hervorragende Möglichkeit diesen Aufwand zu minimieren. Ziel soll es sein, viele kleine unabhängige Einheiten des Quellcodes zu testen. Dennoch ist es in vielen Systemen nicht möglich, Unit-Tests zu etablieren da, wie bereits beschrieben, Komponenten und Programmabläufe zu eng miteinander gekoppelt sind. Demnach strebt man auch im Sinne der Testbarkeit nach einer möglichst gelösten und modularen Struktur. Dadurch soll es möglich sein, die Einheiten getrennt voneinander zu validieren. Gerade im Bereich von Web-Applikationen ist es oft schwierig, eine hohe Abdeckung zu erreichen, da meist der DOM-Zugriff oder AJAX Probleme darstellen.
Zu Beginn einer JavaScript-Webentwicklung, vor allem bei kleineren Projekten

[3] http://jscomplexity.org/complexity

und Scripte, stand oftmals das *ad-hoc*-Testen und Debugging im Mittelpunkt. Darunter versteht man Alerts oder andere Hilfsmittel, um simple Ausgaben der Scripte zu erzeugen. Dass diese Methode des Testens hinsichtlich Skalierbarkeit, Wiederverwendbarkeit und vieler weiterer Faktoren kein tragfähiges Konzept ist, steht dabei außer Frage. Es geht dabei vielmehr darum, die erstellten Scripte in einer ersten Version lauffähig zu machen (vgl. Liang 2010, S. 44). Für größere Applikationen können eigene Testpläne weiterhelfen. Durch den Einsatz von Unit-Tests können dabei Teile davon abgedeckt werden, andere wiederum durch beispielsweise UI-Tests. Grundsätzlich kann aber auch bereits eine Syntax-Validierung, also eine statische Codeanalyse, Fehler im System erkennen und minimieren (vgl. Liang 2010).

Unit-Tests für JavaScript-Applikationen können mittels Jasmine[2] erstellt werden. Die Bibliothek ist dabei ein sogenanntes *Behavior-Driven*-Test-Framework. Dies bedeutet, dass ein Code, der eine bestimmte Spezifikation erfüllt, auch gegen diese validiert werden kann. Die Summe aller einzelnen Spezifikationen wird *Test-Suite* genannt. Diese werden in der Function `describe()` zusammengefasst. Innerhalb einer Test-Suite werden demzufolge mehrere Testspezifikationen ausgeführt. Diese werden jeweils in einer `it()` Function deklariert. Validiert wird jeweils am Ende einer `it()` Function durch Überprüfung eines Zustands mit einem sogennanten *Matcher*. Dies sind Functions, wie z.B. `toEqual()`, `toBeNull()`, welche erwartetes und tatsächliches Ergebnis miteinander vergleichen. Für speziellere Fälle gibt es die Möglichkeit unter Jasmine Spy-Objekte zu erstellen. Diese observieren ein bestimmtes Verhalten oder das Auftreten eines Ereignisses und können am Ende dahingehend ausgewertet werden. Jasmine lässt sich auch mit anderen JavaScript-Frameworks kombinieren (vgl. Hahn 2013). Dadurch wird Jasmine zum relevanten Test-Framework für den Prototyp dieser Arbeit.

[2] http://jasmine.github.io/

Kapitel 3

Umsetzung

Im folgenden Kapitel wird vom Stand der Technik ausgehend die Implementierung des Prototyps beschrieben. Der Prototyp in Form einer Web-Applikation dient dabei der Evaluation bezüglich der definierten Zielsetzung. Die Applikation selbst bildet dabei ein Architekturkonzept ab. Gestützt wird das Konzept durch das JavaScript-Client-Framework AngularJS, welches eines der Möglichkeiten darstellt, um modulare Frontend-Applikationen im Web-Bereich zu entwickeln. Wie bereits eingangs beschrieben entsteht durch die Konzeption einer Web-App ein Prototyp, welcher danach die Ergebnisse hinsichtlich der Ziele Testbarkeit und Erweiterbarkeit liefert. Die genannten Qualitätsmerkmale werden einerseits durch Softwaremetriken im Bereich der Erweiterbarkeit gemessen und andererseits durch Unit-Tests bezüglich der Testbarkeit evaluiert.
Weiterer wichtiger Aspekt einer Web-App ist das UI Design. Um mit einer Web-App konkurrenzfähig zu sein, sollte diese sowohl auf einem Smartphone als auch auf einem Tablet optimal eingesetzt werden. Dabei ist Responsive-Webdesign unabdingbar, wobei der „Mobile-First"-Ansatz bei einer modernen Web-App besondere Aufmerksamkeit erhalten soll. Diese Kriterien, bzw. die sich daraus ergebenen Anforderungen, sind ebenfalls in den Prototyp eingeflossen.
In den nachfolgenden Unterkapiteln wird die Umsetzung der Arbeit Schritt für Schritt erläutert. Ausgehend von der Idee, einer Aufbau- und Ablaufbeschreibung sowie der Durchführung, wird die Erstellung des Konzepts und die Entwicklung des Prototyp im Detail beschrieben. In der zweiten Hälfte des Kapitels wird be-

sonders auf die Zielsetzung eingegangen. Dabei ist die Implementierung neuer Features im bestehenden Konzept bzw. innerhalb des entwickelten Prototyps beschrieben, sowie die Erstellung von Testfällen für denselben.

3.1 Idee, Aufbau und Ablauf

Die Arbeit wie auch die Forschungsfrage selbst findet ihren Ursprung in der Idee einer mobilen App. Wie bereits in den einleitenden Worten zur Arbeit erwähnt, beruht die Idee auf einer Entscheidung der Firma INTECO GmbH, eine App für Marketingstrategien anzubieten. Aus der Anforderung heraus wurden von der Softwareabteilung alle möglichen Szenarien dafür geprüft. Diese Szenarien haben vor allem die Entwicklung der App hinsichtlich der mobilen Plattformen betroffen. Um möglichst präsent zu sein, sollten daher die wichtigsten Plattformen iOS, Android und Windows Phone abgedeckt werden. Nur so kann der eigentliche Zweck, mithilfe der App Marketing zu betreiben, erfüllt werden. Die Szenarien und Kalkulationen zeigten einheitlich auf, dass die Entwicklung drei nativer Apps für die jeweiligen Plattformen zu teuer ist. Eine Entscheidung für nur eine Plattform würde hingegen den Zweck einschränken. Eine Web-App, basierend auf den Technologien HTML5, CSS3 und JavaScript, wurde als die Beste aller Optionen entschieden. Die mobile Web-App für das Unternehmen INTECO GmbH als Kontext gibt der Arbeit auch ihren praxisorientierten Bezug. Unternehmen derselben Größe, oft als Klein und Mittelunternehmen bezeichnet, haben oft ähnliche Anforderungen, welche mit einem guten Konzept in Form einer Web-App anstelle mehrerer nativer Apps gelöst werden können.

Bezugnehmend auf die Anforderungen des Unternehmens an die Web-App, werden diese in kurzer Form vorgestellt. Mit der Web-App soll ein Quiz entstehen, bei dem die TeilnehmerInnen durch eine Art Schatzsuche alle Messestände des Unternehmens bei Messeauftritten durchlaufen. Ziel des Spiels ist es, alle definierten Fragen möglichst schnell zu beantworten. Um eine Frage zu beantworten, wird die/der SpielerIn auf die Suche nach der richtigen Antwort geschickt. Jederzeit besteht auch die Möglichkeit, von der App Hinweise zu einer Frage zu

erhalten. Jeder Hinweis wird jedoch mit einer vordefinierten Strafzeit belegt. Am Ende, nach Absolvierung aller Fragen, wird die benötigte Zeit und additiv die in Anspruch genommenen Strafminuten in eine Punktezahl verwandelt, nach der die TeilnehmerInnen gereiht werden. Alle Fragen sowie Hinweise und deren Gewichtung bezüglich Strafzeit sollen für jeden Messeauftritt neu gestaltet werden können. TeilnehmerInnen am Spiel sind vorher zu registrieren, um im Sinne des Marketings auch eventuelle Gewinnspiele veranstalten zu können. Daher bekommen TeilnehmerInnen einen Benutzernamen und ein Passwort für den Zugang zum Spiel.

Die Prototypenentwicklung ist die Basis für die Evaluierung der Architektur bzw. des Konzepts. Der grundsätzliche Ablauf der Arbeit beginnt mit dem Aufsetzen der Entwicklungsumgebung. Für die Entwicklung wird auf Open-Source-Technologien zurückgegriffen, was sowohl für das Betriebssystem (Linux Distribution Fedora 21) gilt, als auch für die verwendeten Softwarekomponenten. Diese werden nachfolgend bei Erläuterung des Aufbaus näher erklärt. Der Prototyp evaluiert schließlich das Konzept mit dem Framework AngularJS im Detail. Gemäß den Gegebenheiten von AngularJS wird demnach ein Rohkonzept der App erstellt. Bei der Konzepterstellung gilt es grob zu definieren wie Module modeliert werden bzw. wie die Architektur der App grundsätzlich zu erstellen ist. Dabei fließen Coding Standards und andere Guidelines, welche bei der Entwicklung mit diesem Framework üblich sind, ein. Mithilfe des erstellten Konzepts beginnt die Phase der Implementierung der Applikation. Für die Programmierung wird lediglich der Editor *gedit*[1] mit entsprechendem Syntax-Highlighting verwendet. Die wichtigen Punkte der Arbeit bezüglich Erweiterbarkeit und Testbarkeit sind die darauf folgenden letzten Schritte. Für den bereits funktionierenden Prototypen wird ein Modul als Erweiterung angebracht. Während der Implementierung und auch nach der Erweiterung durch ein neues Modul werden für die App Testfälle erstellt. Diese werden in Form von Unit-Tests in einer Jasmine-Umgebung demonstrativ implementiert.

Der Aufbau der Applikation als gesamtheitliches Software-System ist in Abbil-

[1] https://wiki.gnome.org/Apps/Gedit

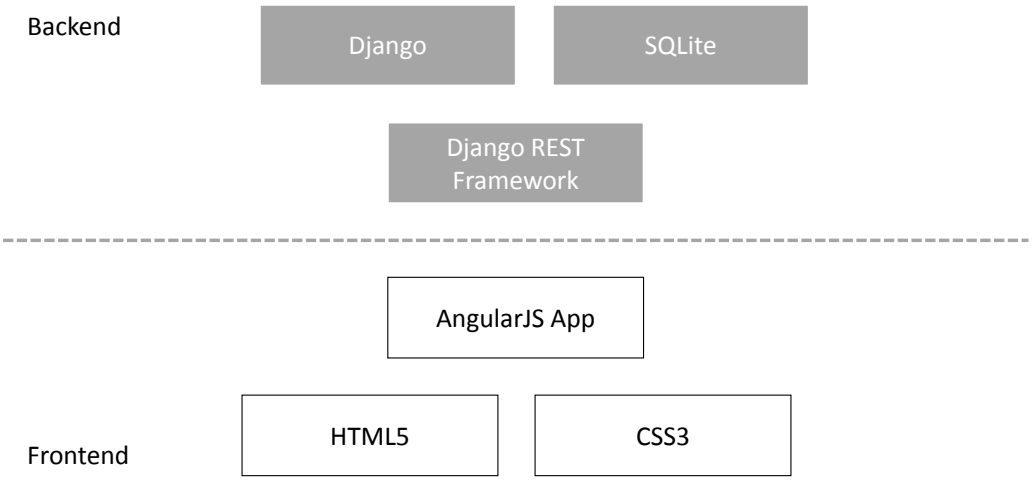

Abbildung 3.1: Prototyp Web-Applikation System Ansicht

dung 3.1 zu sehen. Im folgenden Teilabschnitt wird das Backend des Systems in kurzen Worten erläutert. Das Backend ist grundsätzlich aber nicht Gegenstand der Arbeit, ist jedoch notwendig um eine lauffähige Web-Applikation entwickeln zu können. Die Backend-Applikation ist Datenlieferant der Web-App und ist in Python entwickelt. Diese ist mithilfe der Django[2]-Bibliothek aufgesetzt. Nicht genutzt werden dabei die dynamischen Web-Komponenten, da das Frontend eine JavaScript-Single-Page-Application darstellt. Das Frontend greift nun mittels einer Restfull-Schnittstelle auf das Backend zu. Das REST-API ist mittels der AddOn-Bibliothek „‘Django-Rest-Framework[3]“ programmiert. Das Django-Backend wird somit lediglich über die REST-Schnittstelle angesprochen, ohne dabei selbst einen HTML-Code serverseitig aufzubereiten. Alle benötigten Daten werden vom Backend in eine SQLite[4]-Datenbank gespeichert bzw. wieder von dieser gelesen. Django übernimmt dabei das OR-Mapping, das den Übergang zwischen relationaler Datenbank und objektorientierter Repräsentation darstellt. Vertiefendere technische Details bezüglich der Backend-Server-Applikation werden bewusst ausgespart, da sie nicht Untersuchungsgegenstand der Arbeit sind.

[2]https://www.djangoproject.com
[3]http://www.django-rest-framework.org
[4]https://www.sqlite.org

Sehr wohl Gegenstand und damit im Mittelpunkt der Arbeit ist die Client-Applikation. Daher darf auf die Darstellung des Aufbaus der Web-App im Client-Bereich nicht vergessen werden. Im Übergang zwischen Backend und HTML-Benutzeroberfläche befindet sich die JavaScript-Applikation, wie ebenfalls auf Abbildung 3.1 zu sehen ist. Die Präsentationsschicht ist dabei durch HTML5 und CSS3 gekennzeichnet. Der Web-Client kann unabhängig von seinem Backend als statischer Inhalt auf einem Webserver ausgeliefert werden. Zur einfachen Entwicklung wird dieser als statischer Inhalt vom Django-Entwicklungsserver bereitgestellt.
Durch die Darstellung des Gesamtsystems und dessen Aufbau ist verständlicher inwieweit die Frontend-Applikation darin eingebettet ist und vor allem welcher Teil Gegenstand der Arbeit ist. In den folgenden Abschnitten werden die Konzeption sowie die Implementierung des Prototyps mit anschließender Erweiterung und Testerstellung erläutert.

3.2 Konzept und Implementierung

Der folgende Abschnitt beschäftigt sich ausgehend vom Grundaufbau der Applikation mit dem Konzept und der Architektur und letztlich mit der Implementierung des Prototypen.
Zu Beginn gilt es aus den Anforderungen an die Applikation ein **Konzept** zu erarbeiten. Wie bereits Eingangs des Kapitels erwähnt, handelt es sich um einen Prototypen, der in der Praxis zum Einsatz kommen soll. Aufgrund aller Anforderungen wurde für die Konzepterstellung ein grobes Prozessablaufdiagramm erstellt, dargestellt in Abbildung 3.2. Die Web-App beginnt dementsprechend bei einem Log-In Workflow, welcher die UserInnen authentifizieren soll. Bei nicht korrekten Anmeldedaten wird eine entsprechende Meldung dargestellt. Nach erfolgreicher Anmeldung gelangt die Benutzerin/der Benutzer zum Kontrollzentrum der App, in Form einer Übersichtsmaske, oft auch als Dashboard bezeichnet. Von seinem Kontrollzentrum aus können die BenutzerInnen ein neues Spiel starten und es soll außerdem möglich sein ein bestehendes Spiel fortsetzen zu können. Dies soll

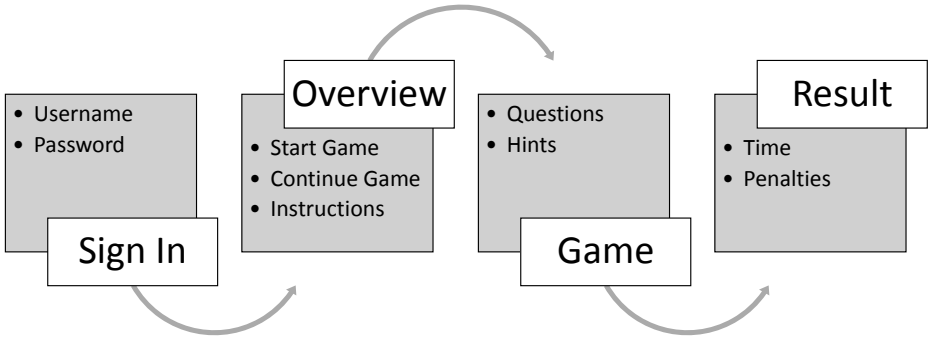

Abbildung 3.2: Prozessablauf der Quiz-App im Erstentwurf

im Falle einer Unterbrechung jeglicher Art die Erwartungshaltung der TeilnehmerInnen erfüllen, das Spiel nicht wieder neu starten zu müssen. Des Weiteren kann vom Dashboard aus zu einer Spielanleitung navigiert werden. Bei den Bedienelementen Spielstart oder Spielfortsetzung wird der entsprechende Workflow gestartet. Durch die Rahmenbedingung, dass die Quiz-App für verschiedene Messen immer wieder neu gestaltet werden können muss, ist der Workflow SSpiel" konfigurierbar konzipiert. Das bedeutet, dass die Summe aller Fragen in einer Matrix aus Fragen und Levels abgebildet sind. Das Spiel wird erfolgreich bestritten, wenn jeweils eine Frage aller Levels richtig beantwortet wird. Dadurch ist es möglich immer wieder neue Wege durch das Spiel zu konfigurieren. Weiters können für jede Frage keine oder beliebig viele Hinweise erzeugt werden. Zugehörig zu den Hinweisen kann auch eine Strafzeit für denselben festgelegt werden. Wird das letzte Level mit einer richtigen Antwort abgeschlossen, ist der Workflow *Spiel* beendet und geht zum *Ergebnis* über. Im abschließenden Prozessschritt wird die Gesamtplatzierung und die benötigte Zeit dargestellt. Die Punktzahl für die Platzierung errechnet sich aus der Summe der benötigter Spielzeit plus der verwendeten Strafzeit durch Hinweise.

Ausgehend von den Anforderungen und dem Prozessablauf dreht sich im nächsten Schritt alles um die Architektur der Web-App. AngularJS selbst bietet dem Entwickler einige Freiheiten die richtig eingesetzt werden wollen. Der Aufbau der App in einzelnen Modulen wird jedoch nicht in Frage gestellt. Wie bereits im Stand

der Technik erläutert, sind Module gekapselte Funktionalitäten der Anwendung. Grundsätzlich hat jede AngularJS Applikation das sogenannte App-Modul als Einstiegspunkt fix verankert. Alles weitere innerhalb der Anwendung hängt von den Anforderungen und der Auslegung der Architektur ab. Bevor jedoch vertiefender in das Modellieren von Modulen eingestiegen werden kann, muss ein weiterer wichtiger Aspekt der Anwendung betrachtet werden. Sie soll sich möglichst verhalten wie eine native App. Um diesem Ziel einen Schritt näher zu kommen, muss am Grundkonzept von Web-Applikationen gedreht werden. Im herkömmlichen Sinne führen Anfragen bzw. Änderungen am UI zu einer neuen Anfrage am Server, welcher nach einer gewissen Latenzzeit HTML als Antwort zur Verfügung stellt. Im Gegensatz dazu stehen sogenannte SPAs. Verweisend auf den Stand der Technik bringt eine SPA dementsprechend den Vorteil, dass das UI am Client zusammengestellt wird und die Daten asynchron vom Server geladen werden. Dadurch wird bei der Bedienung vielmehr der Eindruck einer nativen App vermittelt als durch stetiges Laden einer Website. Das Grundkonzept der Quiz-App ist daher eine SPA. Dieses Konzept wird ebenfalls durch AngularJS entsprechend unterstützt. Zurückkommend auf die Module in AngularJS schließt sich der Kreis, da Module für sich stehende Einheiten von HTML bis Datendienst sein können. In der AngularJS Community gibt es aber zwei grundsätzlich verschiedene Ansätze, Module zu modellieren. Mit diesen Ansätzen geht sehr oft auch die Verteilung des Quellcodes in Datien bzw. deren Struktur in Ordnern einher. Die beiden häufigsten Ansätze werden in folgenden Punkten kurz dargestellt:

1. Module nach Komponententyp

   Module werden nach technisch beeinflussten Aspekten in Komponententypen eingeteilt und dementsprechend aufgebaut. Dies entspricht dabei einer Aufteilung nach den Schichten in einer MVC-Architektur. Im Literaturteil wird diese Form als *Specific-Stereotype-Style* bezeichnet. Dabei werden die Controller, Directives und Services der AngularJS-App in jeweils einem separaten Modul zusammengefasst.

2. Module nach Verhalten bzw. Funktion

Module werden nach ihrem Verhalten bzw. ihrem Mehrwert für die Applikation eingeteilt. Dies bedeutet, dass ein größeres Feature einer App in einem Modul zusammengefasst wird. Hierbei ist wichtig, dass technisch gesehen alle Komponenten in einem Modul vereint sind. Das HTML-Template, dessen Controller oder einem etwaigen Service befinden sich in einem Modul und meist auch physikalisch in einem Ordner wieder. Wiederum vergleichend mit dem AngularJS Literaturteil entspricht diese Organisation dem *Domain-Style*.

Module sind gemäß ihrer Zugehörigkeit zu Komponententypen einfacher zu definieren. Dem liegt bereits der Grundsatz *Seperation-of-concerns* zugrunde. Die generelle Trennung in einer AngularJS Applikation in Schichten bzw. Zuständigkeitsbereiche ist eines der wichtigsten Bestandteile der Architektur des Prototyps. Verweisend auf die Literatur in Kapitel 2.3 werden AngularJS Applikationen meist in eine Datenzugriffsschicht in Form von Diensten (Services), die Controller für die Ablauflogik und das UI eingeteilt. Der Controller interagiert dabei mit dem UI. Für die Architektur der Quiz-App bedeutet dies, es gibt Dienste, welche mit der REST-Schnittstelle des Backend kommunizieren und die erhaltenen Daten bereitstellen. Die Dienste werden von den Controllern angesprochen und verarbeiten die Ein- und Ausgaben. Dadurch ist konzeptionell die Logik, welche mit dem Backend interagiert, getrennt von jener Logik, welche mit dem UI operiert. Bereits in einer frühen Phase des Projekts werden zwei Dienste schnell klar: ein Dienst für die Authentifizierung und ein weiterer Dienst für den Ablauf des Spiels an sich. Abgesehen von der Einteilung in Services und Controller wird für die Entwicklung des Prototypen ein modularer Aufbau gemäß dem *Domain-Style* gewählt. Abbildung 3.3 zeigt die Einteilung der Quiz-App in Module und definiert ebenso bereits benötigte Bestandteile. Dabei fällt auf, dass im Modul *Dashboard* auf einen eigenen Dienst verzichtet wird. Da in der Übersicht grundsätzlich nur zu den nächsten Workflows navigiert werden kann, ist kein eigener Datendienst notwendig. Das Fortführen eines bestehenden Spiels bringt jedoch eine Abhängigkeit zum Modul *Game* und dessen Datendienst mit sich. Beziehungen solcher Art werden vom Framework ebenso unterstützt und stellen daher

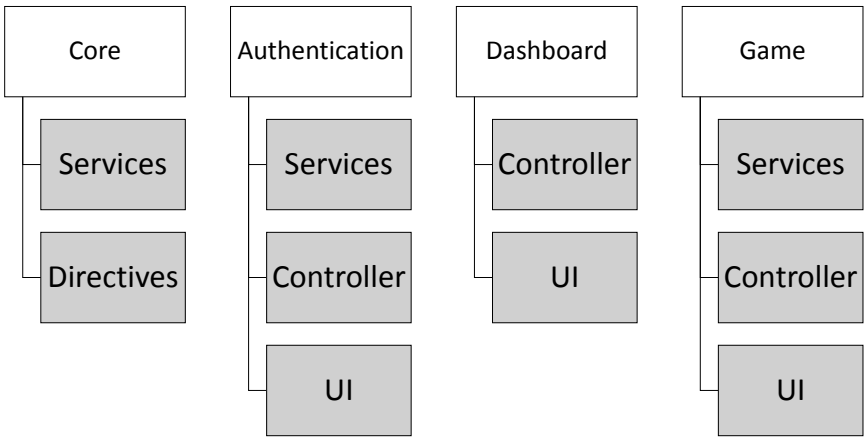

Abbildung 3.3: Module und Struktur des Prototypen

keinen Bruch des Konzepts dar. Das fehlende Service im Modul Dashboard ist jedoch nicht das einzig auffallende. Das Diagramm zeigt ein Modul *Core*, welches im Vergleich zu den anderen keine Funktionalität der App beinhaltet, zumindest nicht auf den ersten Blick. Das Modul beinhaltet jedoch immer wieder benötigte Komponenten und Dienste innerhalb der Anwendung. Daher besitzen alle anderen Module eine Abhängigkeit auf den *Core*. Aus der Abbildung der Module kann ebenso bereits die physikalische Abbildung des Quellcodes abgelesen werden. Einzelne Module finden sich in ihren gleichnamigen Ordner wieder. Jede technisch getrennte Komponente bekommt darin ein eigene Quelldatei mit der festgelegten Namenskonvention <moduleName>.<technicalDetail>.js. Alle Dienste im Modul *Core* befinden sich also in der Datei core.service.js. Dies gewährleistet ein schnelles Zurechtfinden im Projekt und im Quellcode selbst.

Die Architektur in Bezug auf das UI und dem SPA-Ansatz ist mit dem AngularJS-Template-Mechanismus abgebildet. Vereinfacht bedeutet dies, dass die Benutzeroberfläche grundsätzlich nur in der index.html-Datei deklariert wird. Alle weiteren Inhalte werden durch sogenannte Templates in die Hülle eingesetzt. Dadurch ist ein Aktualisieren der Seite nicht mehr notwendig. Templates werden in der Quiz-App durch Routen zur richtigen Zeit eingesetzt. Demzufolge deklariert jedes der Module eine oder mehrere Routen zu einem bestimmten HTML-Template, im

Struktur-Diagramm als UI markiert. Das Template wird von AngularJS, ausgelöst durch eine Navigation, geladen und angezeigt. Dadurch wird automatisch der zugehörige Controller des Templates instanziiert und das Modul ist bereit Eingaben der BenutzerInnen zu verarbeiten bzw. Ausgaben des Servers anzuzeigen.
Nach dem Entwurf des Konzeptes und des Software-Designs für die App wird deren **Implementierung** beschrieben. In der Praxis ist dies kein sequentieller Ablauf, wie die aufeinander folgenden Darstellungen etwa suggerieren. Vielmehr ist es ein iteratives Vorgehen, bei dem nach der Konzeption eines Teilbereichs zumindest ein *Proof-of-Concept* entwickelt wird. Die getrennte Beschreibung von Konzept und Implementierung in der Arbeit soll die einzelnen Bereiche hervorheben und nicht ein Vorgehen nach dem Wasserfallmodell abbilden. Im folgenden Abschnitt werden die Implementierungen der angestrebten Konzepte im Detail erläutert. Am Beginn soll die Betrachtung des Gesamtsystems dabei helfen, die geplante Architektur mit dem Framework *AngularJS* innerhalb der Quiz-App näher zu bringen. Darauf folgt die Detailbeschreibung wie einzelne Konzepte implementiert sind, sowie eine Erläuterung der Entwicklung bis zum Prototyp in einer ersten Version.
Um die geplante Architektur in Quellcode zu verwandeln bzw. das Vorgehen zu beschreiben, ist eine grobe Sicht auf das System von Nutzen. Abbildung 3.4 zeigt schematisch einen Durchlauf durch das Design der Applikation im Querschnitt. Dies wird oft als *Durchstich* vom UI bis hin zum Modell, sei es Datenbank oder gekapselt im Backend, bezeichnet. Ebenfalls zeigt es die Abhängigkeiten der Schichten im System, welche durch die MVW-Architektur entsteht entsteht. Verstanden werden kann das Diagramm wie folgt: Beginnend von einer View in Form eines HTML-Template wird eine Aktion am zugehörigen Controller ausgelöst. Dies impliziert bereits die Instanziierung eines Controller zu jeder View. Das Erzeugen geschieht automatisch durch das Framework sobald eine Route zu einem Template und einem Controller führt. Durch das *Data-Binding* von AngularJS werden Controller bzw. dessen Methoden und Daten an die View gebunden. Dies geschieht für die Applikation mit der *Controller-as*-Syntax, anstelle der Deklaration auf dem `$scope`-Objekt. Controller selbst oder auch Links in HTML können

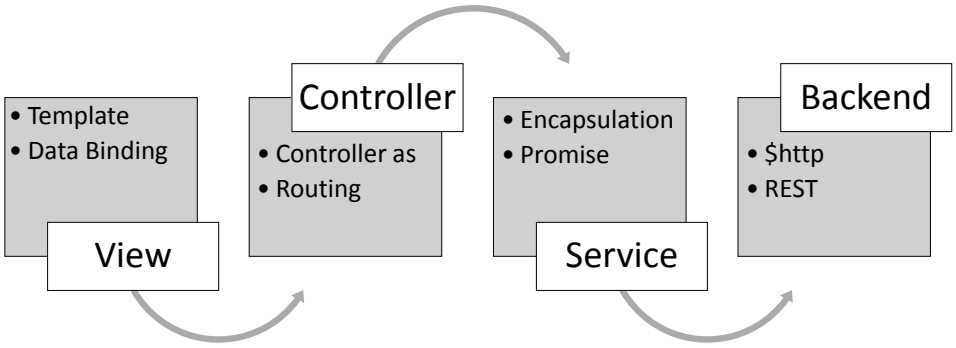

Abbildung 3.4: Abhängikeiten und Beziehungen der Quiz-App

eine Navigation zu einer Route anstoßen, mehr dazu in der Detailbeschreibung des Routing-Konzepts der App im Verlauf des Abschnitts Implementierung. Auf Ebene der Controller findet sich im Grunde die Ablauflogik der App wieder. Kommunikation intern sowie nach außen werden durch Dienste gekapselt. Dies bedeutet, dass den Controllern eine Auswahl an Diensten zur Verfügung steht. In einer MVC-Architektur ausgedrückt, ist dies der Zugriff auf das Modell. Dienste wiederum haben den Anspruch Funktionalität bzw. Abhängigkeiten zu kapseln. Die Quiz-App bietet in Folge dessen Services an, welche wiederum Services aus dem AngularJS-Framework kapseln. Dadurch wird die Austauschbarkeit der Angular-Services ermöglicht. Ebenso wird auf Service-Ebene der Datenzugriff zentralisiert entwickelt. Als Paradigma werden asynchrone Zugriffe über sogenannte *Promise* Objekte aus dem Angular-Framework abgebildet. Um den Durchstich zu komplettieren, verwenden Dienste, um auf das Backend zuzugreifen, den AngularJS-Dienst `$http`. Dieser kann mittels einer REST-URL Resourcen ansprechen und liefert ebenfalls ein *Promise*-Objekt zurück.

Nach dem Überblick über die Implementierung in Form eines Ablaufes bzw. der Darstellung der Beziehungen werden die einzelnen Konzepte der App im Detail erläutert. Am Beginn davon stehen die Beschreibungen der Modultrennung und des *Core*-Moduls. Wie bereits im Architekturteil erwähnt bilden Module gekapselte Funktionaltität ab. Die Applikation selbst besitzt neben den Modulen *Authentication*, *Game*, *Dashboard* auch ein *Core*-Modul, welches die Implementierungen aus

Route	Pfad	Modul
Log In	/auth	Authentication
Dashboard	/dashboard	Dashboard
Instructions	/instructions	Game
Game	/dashboard	Game
Game Result	/finished	Game

Tabelle 3.1: Quiz-App Basis Routenkonfiguration und Modulzugehörigkeit

gemeinsam verwendeten Komponenten umfasst. Dies sind einerseits Dienste wie ein Logging- oder Event-Service sowie auch eine Direktive. Das *Core*-Modul ist vom zeitlichen Verlauf der Prototypenentwicklung erst nach einer Weile entstanden, als immer wiederkehrende Komponenten darin zusammengeführt wurden. Der Einsprungspunkt der Angular-App ist allerdings keines der bisher genannten Module. Die Quiz-App besitzt wie jede Angular-App an oberster Ebene ein *App*-Modul. Bei der Deklaration der App werden alle benötigten Module als Abhängigkeiten angegeben. Um nun die Betrachtung weiter ins Detail zu führen, wird die Implementierung der weiteren Konzepte in den folgenden Punkten genauer erläutert:

Templates

Die Quiz-App ist unter Verwendung der AngularJS-Template-Engine entwickelt. Dies bedeutet im Grunde, dass zur App eine index.html-Datei vorhanden ist. Darin befindet sich der Rumpf des UI und in der Mitte darin wird ein -Tag dafür verwendet, die HTML-Templates einzubetten. Im Rumpf befinden sich im Fall der App das Menü an oberster sowie eine Statusleiste an untester Stelle.

Routing

Eines der ersten und wichtigsten Bestandteile der entstehenden App ist die Routenkonfiguration. Diese ist Teil des Konzeptes und wird im *App*-Modul beschrieben. In der Quelldatei app.routes.js werden die in Tabelle 3.1 abgebildeten Routen deklariert. Besonders zu beachten ist dabei, dass ein Modul nicht nur ein Template-Controller-Paar enthalten kann sondern beliebig viele. Im Modul *Ga-*

me finden sich daher auch die Routen zur Spielanleitung, dem Spiel selbst und der Ergebnisübersicht nach erfolgreich absolviertem Spiel wieder.

Services

Dienste sind, wie eingangs bereits erwähnt, Objekte, welche den Verwendern eine bestimmte Dienstleistung anbieten. In den meisten Fällen werden Dienste für die Kapselung des Datenzugriffs entwickelt. Abgesehen von diesem Fall finden Dienste in einer Angular-App noch einen größeren Einsatzbereich. Vom Event-Mechanismus bis hin zum Authentifizierungskonzept werden Dienste in der Quiz-App eingesetzt, deren Hauptaufgabe nicht der Zugriff nach außen zum Backend ist. Somit werden in den Modulen *Core*, *Authentication* und *Game* Services für die App entwickelt. Diese dienen auch der Zusammenfassung von Zuständigkeitsbereichen nach dem *Single-Responsibility*-Prinzip. Das authenticationService fasst beispielsweise alle Anfragen bezüglich User-Management zusammen, das gameService all jene bezüglich des Spielablaufs.

Authentifizierung

Neben den Routen und dem Hauptmodul selbst befindet sich auch die Datei app.config.js. Diese verfolgt das Ziel, die Konfiguration der Quiz-App zu kapseln. Dabei ist festzustellen, dass die Routentabelle selbst auch Teil der Konfiguration ist. Dennoch ist es ein praktikables und häufig angewandtes Pattern, die Routenkonfiguration getrennt von der restlichen Konfiguration in einer eigenen Quelldatei zu verwalten. Im Konzept der Quiz-App befindet sich in der Datei app.config.js die allgemeine Konfiguration, wohingegen die app.routes.js-Datei dem Template-Mechanismus dient. Der Konfigurationsteil für die Quiz-App beinhaltet Einstellungen für den $httpProvider und dient dem Authentifizierungskonzept. Für die Authentifizierung der BenutzerInnen gibt es für Web-Apps viele Ansätze. Dies ist unabhängig von der Verwendung des AngularJS-Framework. Es können dabei moderne Technologien wie Json-Web-Token ebenso verwendet werden wie die Session-Authentifzierung mit einem clientseitigen Cookie. Für den Prototyp wurde die Methode der Cookie-Authentifizierung gewählt, da dies vom Django-Backend ohne zusätzlichen Programmieraufwand unterstützt wird und der Authentifizierungsmodus nicht Kernthema der Arbeit ist. Wie die Metho-

de in die Applikation eingebettet wird, ist hingegen von konzeptioneller Bedeutung. Im Hinblick auf die Austauschbarkeit ist es wichtig, dass in weiterer Folge die Methode der Authentifizierung verändert werden kann ohne viel Logik der App verändern zu müssen. Dies führt zurück zur Konfiguration des $httpProvider. Die Konfiguration beinhaltet zwei wesentliche Aspekte. Zum einen werden in der Eigenschaft defaults des Provider die Namen des Cross-Site-Request-Forgery (CSRF)-Cookies und des CSRF-Headers gesetzt. Diese beiden Einstellungen führen in weiterer Folge dazu, dass bei jedem Request über HTTP das gespeicherte CSRF-Cookie mitgeschickt wird. Ohne diesem würde das Backend den Request grundsätzlich ablehnen. Zum anderen besteht ein wesentlicher Teil der Konfiguration daraus, dem Provider einen sogenannten *Interceptor* hinzuzufügen. Dieser Interceptor ist eine Möglichkeit des Frameworks in alle Aktivitäten über HTTP (Request/Response) einzugreifen. Dies wird hier genutzt um, die Authentifizierung durchzuführen. Der Name des Interceptors wird der Eigenschaft $httpProvider.interceptors angehängt. Die eigentliche Funktionalität befindet sich hingegen im *Authentication* Modul.

```
1  function authInterceptor($rootScope, $q) {
2    ...
3      function response(response) {
4          if(response.status === 401 || response.status === 403) {
5            // not authorized
6            $rootScope.$broadcast('authorizationFailure', response);
7          }
8      };
9      function responseError(response) {
10         ...
11     };
12 };
```

Listing 3.1: Authentication Interceptor

In Listing 3.1 wird die Funktionsweise durch den Code-Ausschnitt des Interceptors angedeutet. Jede Antwort vom Server wird vom Interceptor abgefangen und in den entsprechenden Methoden response(...) oder responseError(...) behandelt. Über den Status lässt sich anschließend leicht feststellen, ob die Anfrage erlaubt oder unerlaubt stattgefunden hat. Im Fehlerfall wird vom Interceptor ein Broadcast-Event an die Applikation über den $rootScope versandt. Lose Kopplung ist ein hohes Gut, welches insbesondere für Erweiterbarkeit steht und auch in diesem Fall gewahrt werden soll. Die Behandlung des Fehlers selbst obligt somit nicht mehr dem *Authenticaton*-Modul, sondern demjenigen in dessen Zuständigkeitsbereich der Fehler fällt. Konkret wird in der Methode run(...) des Moduls *App* auf das „authorizationFailure“ Event reagiert, indem direkt zum Login-Workflow navigiert wird.

Events

Innerhalb der Applikation gibt es immer wieder die Anforderung, auch modulübergreifend auf Ressourcen bzw. Komponenten zuzugreifen. Im konkreten Fall von einem Controller eines Moduls auf den Controller eines anderen. Diese Anforderung ist ähnlich der des Authentifizierungskonzeptes beim Fehlerfall. Dort wird im Fall einer ungültigen Anmeldung ein Event verschickt. Die Lösung dieses übergreifenden Problems ist ebenfalls ähnlich der des Authentifizierungskonzepts. Das *Core*-Modul bietet ein eventService, bei dem die Möglichkeit besteht, sich für eine Nachricht zu registrieren. Dementsprechend wird die registrierte Komponente bei Auftreten des Events benachrichtigt. Listing 3.2 zeigt die Implementierung des Dienstes. Bei genauerer Inspektion fällt auf, dass der eigene Event Dienst wiederum den bekannten $rootScope aus dem Framework kapselt. Dies schafft einerseits Unabhängigkeit und andererseits, anhand des Logging-Dienstes zu sehen, die Möglichkeit des Tracings. Generell wäre es daher in einem Refactoring anzustreben, den $rootScope im Interceptor durch den Event-Dienst zu ersetzten.

```
1 function eventService($rootScope, loggingService) {
2     var service = {
3         broadcast: broadcast,
4         subscribe: subscribe
```

```
5     };
6     return service;
7     /* service declarations */
8     function broadcast(eventName, payload) {
9         loggingService.debug('broadcast(): ' + eventName);
10        $rootScope.$broadcast(eventName, payload);
11    };
12    function subscribe(eventName, action) {
13        loggingService.debug('subscribe(): ' + eventName);
14        $rootScope.$on(eventName, action);
15    };
16 };
```

Listing 3.2: Event Service aus dem Core der App

Controller As

Jedes HTML-Template, aufgeteilt auf die verschiedenen Module, wird durch Navigation zu einer bestimmten Route vom Framework geladen. Durch die Routenkonfiguration wird zum Template die entsprechende Controller-Funktion aufgerufen. Die beiden Komponenten sind über Data-Binding miteinander verbunden. Dies geschieht für den Prototyp nicht durch die $scope-Variante, sondern über die *Controller-as*-Syntax. Bei der Routenkonfiguration ist dabei zu beachten, nicht nur den Name für den Controller zu setzen, sondern auch das Feld controllerAS mit einer Variablenbezeichnung zu belegen. In der App ist dies entsprechend einer Guideline immer „vm“, was für den Begriff *ViewModel*, aus dem erwähnten MVVM Design Pattern steht.

```
1 function DashboardController($location, loggingService,
     eventService, authenticationService, gameService) {
2     var vm = this;
3
4     /* view model properties */
5
```

```
6     vm.user = null;
7     vm.hasCurrentGame = false;
8     vm.state = {
9         header: 'IQuest'
10    };
11    ...
12 };
```

Listing 3.3: Ausschnitt des Dashboard Controller der App

Listing 3.3 zeigt den wesentlichen Ausschnitt des *DashboardController*, um die Syntax zu erläutern. Es wird keine $scope Instanz in den Controller injiziert. Stattdessen deklariert dieser eine Variable mit dem Namen ‘vm“ und setzt diese auf den this-Pointer. Dieser Name muss mit dem oben erwähnten Namen der controllerAs-Einstellung im Routing korrelieren. Dannach können die Daten mit dem Data-Binding Konstrukt {{vm.user.username}} an die View gebunden werden.

Directives

Ebenfalls für die Quiz-App zum Einsatz kommt das Konzept der Direktiven. Dabei werden Komponenten inklusive Template in wiedervendbaren Einheiten zusammengefasst. Die wichtigste Größenordnung der Quiz-App während des Spiels ist einerseits die verbrauchte Zeit und andererseits die Strafzeit für Hinweise. Um die Zeit im selben Format (hh:mm:ss) immer gleich darzustellen, ist eine Direktive dafür entwickelt worden. Diese Direktive beinhaltet jedoch kein eigenes HTML-Template, sondern besitzt lediglich einen eigenen Controller. Der Controller wiederum verfügt mittels *Isolated-Scope* über einen eigenen Bereich für Data-Binding. Listing 3.4 zeigt die Verwendung der Direktive im HTML-Code. Des Weiteren kann diese auf verschiedene Arten verwendet werden. Die einfachste Variante ist das Berechnen einer statischen Zeitdauer. Diesen Wert bekommt der Controller der Direktive übergeben und errechnet daraus die Anzeige im Format 05:30:00. Auch kann die Direktive über den Parameter *run* gestartet werden, um von einem Zeitstempel aus die vergangenen Stunden, Minuten und Sekunden zu berechnen. Dieser Parameter ist per Default aktiviert. Bei der Implementie-

rung wird der AngularJS Dienst $interval verwendet, welcher auf Sekundentakt konfiguriert ist. Durch jedes Erreichen einer Sekunde werden die Stunden, Minuten und Sekunden im Controller neu berechnet. Wie in der Verwendung gezeigt, kann innerhalb der Direktive auf diese Werte gebunden werden. Dadurch entsteht letztlich der Effekt, dass die Zeit am UI nach oben zählt.

```
1 ...
2 <span class="label label-primary pull-right">
3     <time-component start-time="vm.game.start_time">{{ hhours
          }}:{{ mminutes }}:{{ sseconds }}</time-component>
4 </span>
5 ...
```

Listing 3.4: Ausschnitt aus der Verwendung der time-component Direktive

Im folgenden Abschnitt wird nach der Detailbeschreibung einzelner Komponenten der Ablauf bis zum funktionalen Prototypen zusammengefasst. Nach dem *Authentication*-Modul werden die fehlenden Module *Dashboard* und *Game* entwickelt. Dies geschieht auch vom zeitlichen Ablauf her parallel, da das Dashboard einerseits der weiteren Navigation dient und andererseits vom *gameService*-Objekt abhängig ist. Das *gameService* selbst verfolgt ebenso den Ansatz, alle Backend-Zugriffe bezüglich der Quiz-Logik zu kapseln. Der *DashboardController* leitet indessen das Routing für die Workflows *Spiel erstellen* und *Spiel fortsetzen* ein. In beiden Fällen übernimmt der *GameController* die Arbeit. Da bereits über das Dashboard zu einem bestimmten Workflow navigiert wurde, kann der *GameController* über den *gameService*-Dienst die entsprechenden Quiz-Daten laden. Damit wird auch der Zyklus der Fragen gestartet. Die Frage wird durch Eingabe und Bestätigung in Form einer Antwort über den Dienst an den Server geschickt. Dies wird mit einem true oder false bestätigt. Bei einer positiven Rückmeldung wird der Dienst um die nächste Frage gebeten. Ist es jedoch so, dass vom Spielablauf her bereits die gestellte Frage des letzten Levels beantwortet wurde, so wird vom Server der HTTP Status 400 Bad Request geschickt. Der Status wiederum wird vom Dienst durch einen entsprechenden Returnwert ersetzt und an den

Controller übergeben. Dadurch weiß der *GameController*, dass der Spielablauf beendet ist und zur Ergebnisübersicht navigiert werden kann.

3.3 Erweiterbarkeit und Testbarkeit

Das Konzept und die Implementierung in Summe ergibt einen Prototypen, welcher die Funktionen und Anforderungen and die Quiz-App in erster Version erfüllt. Die Software wurde in einer Architektur und einem Konzept entworfen, welches nun auf die Erweiterbarkeit bzw. Testbarkeit überprüft werden muss. Grundsätzlich ist dabei festzuhalten, dass Applikationen in allen Ausprägungen erweitert werden können. Die Zielsetzung in Bezug auf die Quiz-App liegt dabei besonders darauf, auf welche Art und Weise Erweiterungen und auch Tests abgebildet werden können. Dies soll in weiterer Folge mehr Aussage über die Stabilität und die Wandelbarkeit des Konzeptes darlegen.
Am Beginn jeder **Erweiterung** steht eine neue bzw. veränderte Anforderung an das Softwaresystem, in diesem Fall an die Quiz-App. Um TeilnehmerInnen einen besseren Überblick über das Spiel bzw. vielmehr den eigenen Spielverlauf zu gewähren, soll die App eine Statistik zu bereits gespielten Quiz-Durchgängen anzeigen. Dabei sollte eine Tabelle der Spiele angezeigt werden, in welcher die letzten 10 Spiele, gereiht nach der erreichten Punktzahl, gelistet sind. Die Informationen, wann das Spiel begonnen wurde sowie verbrauchte Spielzeit und benötigte Strafzeit, sind dabei zu visualisieren. Aus den Anforderungen heraus ist nun zu entscheiden wie die Erweiterung in die bestehende App eingebettet wird. Da sich die Anforderung von den bestehenden Funkionalitäten abhebt, fällt die Entscheidung darauf, die Applikation um ein neues Modul *Statistik* zu erweitern. Das Modul soll wiederum wie bei der Implementierung der anderen Module querschnittlich alle benötigten Komponenten abdecken. Dabei wird schnell klar, dass die Daten für die Statistik vollumfänglich die Spiele an sich betreffen. Um dabei das bestehende *gameService* unangetastet zu belassen, fällt die Entscheidung darauf, den Datenzugriff ebenfalls im neuen Modul zu implementieren. Eingebettet bzw. erreichbar durch Navigation soll die Statistik vom Dashboard aus sein. Aus

diesen Informationen ergeben sich Änderungen bzw. Entwicklungen für folgende Bereiche:

1. Statistik-Modul

   Das Statistik-Modul stellt eine Neuentwicklung HTML-Template bis hin zum *statisticService*-Dienst dar. Dabei entsteht lediglich eine Abhängigkeit zum bestehenden Modul *Core*.

2. Dashboard-Modul

   Hierbei sind Anpassungen des Dashboard-HTML-Templates notwendig, um auf die neue Statistik navigieren zu können. Ebenso wurde ein Refactoring des Controllers durchgeführt, um mit einer bereitgestellten Methode navigateTo(...) zu allen Routen navigieren zu können. Der Methode wird im HTML-Template die Route als *String*-Parameter übergeben.

3. App-Modul

   Erweiterung des App-Moduls durch eine neu entstandene Abhängigkeit auf das Statistik-Modul. Des Weiteren wird der Routenkonfiguration eine neue Route (/statistics) in der app.routes.js-Datei hinzugefügt.

Abbildung 3.5 zeigt das Dashboard auf einem Windows Phone inklusive der Erweiterung um das Statistik Modul. Damit zeigt der Screenshot auch die finale Version der Quiz-App. Um die Applikation bzw. die richtige Funktionsweise zu gewährleisten, ist eine hohe **Testbarkeit** unerlässlich. Kleine Testeinheiten in Form von Unit-Tests helfen dabei, den Code nach Veränderungen oder Erweiterungen erneut ohne Aufwand auf seine Funktionalität zu überprüfen. Das AngularJS Framework bietet durch die lose Kopplung mittels *Dependency Injection* eine gute Möglichkeit, die Komponenten unabhängig voneinander zu testen. Die Unit Tests werden dabei mithilfe des Jasmine Framework erstellt. Technische Details zum Framework sind im Literaturteil beschrieben. Der Aufbau der Tests für die App besteht aus einer test.html-HTML-Datei, welche neben der index.html zu finden ist. Die einzelnen Testfälle, in *Jasmine* auch Specifications (Specs) genannt, werden in eigenständigen Dateien abgelegt. Die Namenskonvention dafür ist an die Konvention der bstehenden Quelldateien angepasst und sieht einen Postfix für

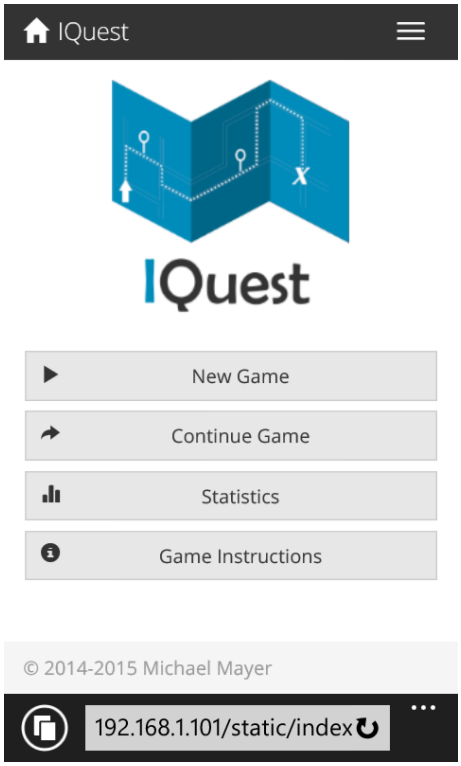

Abbildung 3.5: Screenshot der Quiz Web-App

die Specification vor <moduleName>.<technicalDetail>.spec.js. Für die Struktur bzw. die Organisation der Specification-Dateien wird von der Community empfohlen diese direkt neben den Quelldateien, die getestet werden, abzulegen. Dadurch soll vor allem Bewusstsein geschaffen werden, dass bei Anpassungen am Quellcode auch die Testspezifikation angepasst werden muss. Durch die Organisation von Modulen durch Gliederung von Features bzw. Feature Scopes ist die Verwahrung der Specifications im selben Ordner auch inhaltlich korrekt. Ebenfalls entspricht sowohl jede Testspezifikation als auch die einzelnen Testfälle darin einer für EntwicklerInnen lesbaren Beschreibung. Für den Prototyp wurden zur Demonstration der Testbarkeit, ausgewählt aus jeder MVC-Schicht, folgende Testspezifikationen erstellt:

1. Dienste

   In diesem Bereich wurden Spezifikationen für die Dienste im *Core* wie auch für das *gameService* erstellt. Dabei kommen Konzepte wie die Mock-Objekte zum Einsatz. Durch den Einsatz des httpBackend-Mock-Objekts

aus dem Angular-Framework sind die Datenzugriffe in den Services getestet.

2. Direktiven

   Für das Core-Modul wurde eine Spezifikation für die Direktive implementiert. Die Spezifikation testet konkret die time-component-Direktive mithilfe von Mock-Objekten, um die Zeitintervalle des AngularJS-Dienstes $interval zu simulieren.

3. Controller

   Um die Ablauflogik innerhalb der Controller zu testen, wurden 2 Controller ausgewählt. Aus dem Authentication-Modul ist eine Spezifikation für den *AuthenticationController* und aus dem Game-Modul eine für den *GameController* implementiert. Dabei kommt das Konzept von Teststubs zum Einsatz, um nicht nur AngularJS-Dienste, sondern auch die eigenen *gameService*- oder *authenticationService*-Dienste für die Testfälle zu ersetzen.

Die erstellten Spezifikationen werden neben der *Jasmine*-Umgebung als Scripte in die *test.html*-Datei eingebunden. Die Quiz-App Quelldateien werden ebenfalls bereitgestellt, damit sie von der Testumgebung geladen werden kann. Durch Aufruf der Test-Webseite im Browser werden die Testfälle ausgeführt und das Ergebnis davon in einem Report dargestellt.

3.4 Ergebnisse

Im folgenden Abschnitt werden die Ergebnisse der Arbeit dargestellt. Diese beruhen auf den Auswertungen des Prototypen hinsichtlich der Zieldefinition. Aus diesem Grund sind die genannte Ziele im Folgenden noch einmal kurz zusammengefasst. Die Arbeit verfolgt den Aufbau einer Web-Applikation für mobile Endgeräte. Die Methodik der Prototypenentwicklung dient der Begleitung der Phasen von Konzepterstellung und Architekturentwurf über die Implementierung bis hin zur Testerstellung. Nach Abschluss der ersten lauffähigen Version der App wurde durch eine neue Anforderung eine Erweiterung an der bestehenden Software

vorgenommen. Damit durchläuft der Prototyp alle notwendigen Schritte, um das Konzept der Web-App mit den Zielen im Bereich der Codequalität gegenüber zu stellen. Aus diesem großen Bereich werden konkret die Erweiterbarkeit und die Korrektheit in Form der Testbarkeit im Detail betrachtet.
Das Konzept und die Architektur des Prototypen wurde unter dem Rahmenwerk des AngularJS JavaScript Framework entwickelt. Das Framework beansprucht bereits im Vorfeld die im Literaturteil vorgestellten Qualitätsmerkmale für sich. Im Allgemeinen hat die Entscheidung für den Einsatz von AngularJS einige Auswirkungen. Zu beachten ist die nicht unerhebliche Einarbeitungszeit in das Framework. In Zahlen ausgedrückt, fließen geschätzt 10 Tage vor der Konzepterstellung in das Kennenlernen des Frameworks und seiner Umgebung. Der Einarbeitungszeit steht jedoch eine Vielzahl an Beispielen der Community gegenüber, wodurch der Einstieg in AngularJS vereinfacht wird.
Die Ergebnisse im Bereich Erweiterbarkeit sind auf gemessene Softwaremetriken zurückzuführen. Verweisend auf den Literaturteil sind dies die Lines-of-Code (ohne Leerzeilen), die zyklomatische Komplexität und der Maintainability-Index. Diese Kennzahlen wurden mit JSComplexity gewonnen und ausgewertet. Im Bereich der Testbarkeit sind unter Verwendung der Bibliothek *Jasmine* Unit-Tests erstellt worden. Der Prototyp wird dementsprechend gemessen, inwiefern es möglich ist, kleine Teile zu testen, um eine hohe Testabdeckung zu erzielen. Die Detailergebnisse der beiden Bereiche werden in folgender Aufzählung dargestellt:

1. Erweiterbarkeit

   Der JavaScript Source Code wurde mit JSComplexity ausgewertet und weist die in Tabelle 3.2 - Spalte „Erste Version“ - erreichten Werte auf. Dabei wurde lediglich der für den Prototypen entwickelte Source Code herangezogen, kein Bestandteil eines Frameworks. Um eine bessere Auswertung zu erhalten, werden die gemessenen Metriken der ersten Version (Spalte Erste Version) denen der zweiten Version (Spalte Zweite Version) gegenübergestellt. Daraus können nun folgende Resultate herausgelesen werden: Die logischen LOC zeigen einen Zuwachs durch die Erweiterung von 43 Zeilen. Dies entspricht einem relativen Wachstum der Codebasis um 7%. Die zy-

Softwaremetrik	Erste Version	Zweite Version
Logical LOC	599	642
Cyclomatic Complexity	36	37
Cyclomatic Complexity density	6%	6%
Maintainability Index	126	126

Tabelle 3.2: Metriken des Prototypen nach JSComplexity

klomatische Komplexität wurde im Fall der ersten Version auf 36 gemessen. Dieser Wert zeigt eine durchaus erhöhte Komplexität des Projekts. Durch JSComplexity lassen sich die Übeltäter schnell ausfindig machen. Die beiden Ausreißer in der App sind die Direktive *time-component* in der Datei core.direcitve.js mit einer zyklischen Komplexität von 15, gefolgt von der game.controller.js-Datei mit einem Wert von 8. Dies ergibt sich vor allem daraus, dass mehrere Controller in der Datei definiert sind. Dennoch könnte in diesen beiden Bereichen ein Refactoring dazu führen, die Komplexität weiter zu senken. Der nächste Wert in Tabelle 3.2 stellt die zyklomatische Komplexität ins Verhältnis zu den LOC. Dadurch erhält der in Prozent ausgedrückte Wert mehr Bedeutung auf das Gesamtprojekt bezogen als die reine zyklomatische Komplexität. Die letzte Kennzahl der Tabelle ist der Maintainability Index. Dieser beläuft sich auf 126 wobei 171 das Maximum darstellt. Der Index von 126 bekommt erst Gewicht, wenn man Aussagen von Oman und Hagemeister aus Originaldokument heranzieht, wo festgehalten wird, dass ca. ein Index von 65 der Schwellwert zwischen wartbarem und nicht mehr wartbarer Architektur ist. Weiters zeigen die Ergebnisse der Auswertung eines im besonderen: Die Erweiterung der Codebasis um 7% haben die zyklomatische Komplexität um 1 erhöht, jedoch ist die Komplexitätsdichte (6%) sowie der Maintainability Index (126) gleich geblieben. In Summe zeigt die Metrik, dass eine Erweiterbarkeit einer App mit AngularJS möglich ist, sofern man Konzepte wie Module, Dienste und Controller, welche allesamt zur Modularisierung beitragen, auch nutzt.

Test Suite Beschreibung	Test Spezifikationen
core.directive.spec.js	3
core.filter.spec.js	2
core.service.spec.js	5
authentication.controler.spec.js	4
game.service.spec.js	4

Tabelle 3.3: Jasmine Unit Test des Prototypen

2. Testbarkeit

   Der im Bereich der Erweiterbarkeit gelegte Grundstein ist auch die Basis für ein gutes Maß an Testbarkeit. Nicht nur die horizontale Trennung in domainspezifisch zusammengefasste Module, sondern auch die vertikale Trennung in technisch modulare Komponenten, ermöglichen im Konzept den Einsatz von Unit-Tests mit Jasmine. Wie bereits erwähnt sind sowohl ein modulares Design als auch lose gekoppelte Komponenten Indikatoren für gute Testbarkeit. AngularJS unterstützt diese Anforderung mit seinem *Dependency-Injection* Konzept. Dabei definiert jede Komponente, ob Dienst, Controller oder Interceptor, seine Abhängigkeiten als Parameter der Funktion selbst. Dies ist beispielsweise vergleichbar mit einem *Container* und Constructor Injection im J2EE Bereich. Dieses Konzept ermöglicht während der Ausführung einer Unit-Test-Spezifikation die Injektion von Mock- oder Spy-Objekten. Diese dienen am Ende der Testspezifikation dazu, den Status bzw. den gewünschten Ablauf zu validieren. Diese Konzepte schaffen die Grundvoraussetzung einer hohen Testabdeckung im Unit-Test Segment. Um diese Konzepte der Testbarkeit mit einem Beweis zu unterlegen, ist für den Prototyp mindestens für jeden Komponententyp ein Unit-Test erstellt worden. Tabelle 3.3 zeigt, dass für das Core-Modul, welches den höchsten Wiederverwendungsgrad besitzt, alle sich darin befindlichen Komponenten getestet werden konnten. Des Weiteren wurde demonstrativ dargestellt, das auch Controller vollständig getestet werden können. Dies er-

möglicht vor allem auch das *Two-Way-Data-Binding*-Konzept welches den Controller nur lose mit dem DOM verbindet. Um mit den Diensten alle architektonischen Schichten abzudecken, wurde eine Test-Suite für den gameService-Dienst erstellt. Für die Testspezifikationen im Service wurde ein Mock-Objekt zum Ersetzen des $http-Dienstes verwendet. Dadurch wurde das Backend für den Unit-Test simuliert und das Verhalten des Service konnte vollständig validiert werden.

Zusammenfassend ist auch für die Testbarkeit festzustellen, dass mit Jasmine und AngularJS vieles im Bereich Unit-Tests ermöglicht wurde. Dennoch ist anzufügen, dass je komplexer eine Komponente ist, desto komplexer werden auch die Tests. Dies hat sich besonders bei der Direktive, diese erreichte eine zyklomatische Komplexität von 15, wiedergespiegelt. Dennoch war es letzten Endes möglich, unter Verwendung von Mock-Objekten, den von der Direktive erzeugten HTML-Code zu validieren.

Kapitel 4

Resümee

Durch moderne Frameworks sowie den Support auf vielen Smartphones der jüngsten Generation ist die Entwicklung einer mobilen Web-App mehr als eine Alternative zu einer nativen App. Die Vorteile von Web-Apps liegen unter anderem in der Plattformunabhängigkeit und an den neuen Möglichkeiten des UI Designs, welche an jene von nativen Apps herankommen. Die Bachelorarbeit ermittelt die Vor- und Nachteile moderner Konzepte und Architekturen mit Hilfe des AngularJS Frameworks.

Die **App** ist Form einer Web-App mit den Technologien HTML5, CSS3 und JavaScript als sogenannte Single-Page-Application (SPA) konzipiert. Durch die Verwendung des Prototyps für das Unternehmen INTECO GmbH erhalten der Prototyp und die Arbeit einen praxisorientierten Bezug. Die erste Phase der Arbeit ist die Konzepterstellung und Architektur der Web-App. Das Konzept beruht auf dem Einsatz von AngularJS als JavaScript-Client-Framework. Die Struktur des Projekts ist im *Domain-Style* angelegt. Dies bedeutet, dass größere Features der Applikation zu Modulen zusammengefasst werden und in gleichnamigen Ordnern verwaltet sind. Im Vergleich zur *Stereotype*-Struktur, bei welcher alle Dienste, Controller oder Direktiven separat in einem Ordner zusammengefasst werden, sind in der *Domain*-Struktur alle Komponenten *Layer*-übergreifend in einem Modul vorhanden. Daraus ergibt sich bereits in gewissem Maße die modulare Architektur der Web-App. Der Prototyp umfasst in der ersten Version 4 Module (Core, Authentication, Dashboard und Game). Um den Anforderungen

einer SPA gerecht zu werden, wird der von AngularJS angebotene clientseitige Template Mechanismus verwendet. Dies bedeutet, dass das User Interface (UI) erst am Client zusammengebaut und verändert wird. Mittels *Routing* wird zwischen den Modulen und Views der App navigiert. Routing ist ebenso Bestandteil von AngularJS und sorgt dafür, dass beim Navigieren zu einer Route das dahinter konfigurierte HTML-Template geladen wird. Hinter jeder View, also jedem Template, kann sich in einer Angular-App ein Controller verbergen. Dieser übernimmt die Abläufe auf dem UI und delegiert Anfragen an Dienste weiter. Innerhalb einer Model-View-Controller (MVC) Architektur, würden sich diese Dienste im Bereich zwischen Controller und Modell wiederfinden. Die Controller können sich dabei einer Vielzahl an Diensten bedienen. Diese Services dienen der Kapselung von Funktionalitäten. Eine davon ist die zentrale Verwaltung von Datenzugriffen zu einem Backend. Der Prototyp definiert einige Services, welche mit einem Restful-API kommunizieren und asynchron Daten an die Aufrufer (meist Controller) zurückliefern. Durch das *Data-Binding* Konzept von AngularJS werden die Daten anschließend am UI angezeigt. Zur Darstellung der benötigten Zeit im Format *hh:mm:ss* innerhalb eines Spieles der Quiz-App wurde eine sogenannte Direktive entwickelt. Direktiven in AngularJS bieten die Möglichkeit, die HTML-Syntax um eigene Elemente zu erweitern. Nach der letzten Phase der Entwicklung wurde hinsichtlich der Zielsetzung die App um ein neues Modul erweitert, sowie Unit-Tests mit Jasmine entwickelt. Das neue Statistik-Modul bietet den AnwenderInnen eine Übersicht über bereits bestrittene Spiele.

Die **Ergebnisse** der Arbeit sind in Bezug auf die Erweiterbarkeit mit Softwaremetriken gemessen. Die wichtigsten Kennzahlen, welche bei der Auswertung verwendet wurden, sind die logischen Codezeilen (LOC), zyklomatische Komplexität, prozentuelle zyklomatische Komplexität sowie der Maintainability-Index. Gemäß Literatur ist die Wartbarkeit des Projektes besser je höher dieser Wert liegt, wobei 171 das Maximum darstellt. Der Prototyp erreicht bei der Auswertung einen Index von 126. Nach der Erweiterung des Quellcodes um das Statistik-Modul, was einem Zuwachs der Codebasis von 7% entspricht, bleibt der Index auf demselben Niveau von 126. Dies ist ein solider Hinweis auf eine modulare, strukturierte und

erweiterbare Architektur.

Mit Hilfe des Jasmine-Framework wurden Testspezifikationen für die App entwickelt, um die Testbarkeit derselben zu untersuchen. Dabei entstand eine Gesamt-Testabdeckung des Modules *Core*. Dies ist besonders wichtig, da es in jedem anderen Modul referenziert wird. Übergreifend über alle anderen Module wurde beispielhaft jeweils eine Test-Suite für Controller und Service erstellt. Eine Test-Suite umfasst dabei alle Testspezifikationen für einen Bereich. Dadurch sind beispielsweise alle Spezifikationen für den gameService-Dienst in einer Test-Suite zusammengefasst. AngularJS selbst bietet eine Vielzahl an Mock-Objekten für das Testen mit Unit-Tests. Vor allem jedoch das *Dependency-Injection* Konzept sorgt dafür, dass jede Komponente für sich getestet werden kann und wenn notwendig, Abhängigkeiten durch entsprechende Mock-Objekte ersetzt werden können.

Daraus ergeben sich folgende **Schlussfolgerungen**: AngularJS als Framework bietet durch seinen Umfang die Möglichkeiten, modulare, erweiterbare und testbare Web-Apps zu entwickeln. Dadurch kann bei der Entscheidung für eine Web-App die Codequalität gesteigert werden und somit ein besserer und auch kostensparender Applikations-Lebenszyklus gewährleistet werden. Aus den Erfahrungen mit AngularJS während der Entwicklung des Prototyps ist zu sagen, dass der Aufbau der Ordnerstruktur sehr stark von der Größe des Projektes abhängt. Des Weiteren ist zu empfehlen, bei größeren Projekten die Testspezifikiationen im Sinne eines Test-Driven-Development von Anfang an zu berücksichtigen.

Abschließend ist anzumerken, dass Web-Apps auf dem mobilen Sektor zweifelsfrei eine immer stärker werdende Alternative darstellen. Mit den Möglichkeiten von JavaScript und Frameworks wie AngularJS können die Anforderungen einer modernen Web-App mit neuen Konzepten strukturierter abgebildet werden.

Literaturverzeichnis

angularjs.org (2015), 'AngularJS - Superheroic JavaScript MVW Framework', `https://angularjs.org/`. [Online; accessed 03-Jan-2015].

backbonejs.org (2014), 'Backbone.js', `http://backbonejs.org/`. [Online; accessed 28-Dez-2014].

Bednarski, W. (2013), *Learning JavaScriptMVC learn to build well-structured JavaScript web applications using JavaScriptMVC*, Packt Publishing Ltd, Birmingham, UK.

Branas, R. (2014), *AngularJS Essentials*, Packt Publishing Ltd, Birmingham.

Clark, R., Studholme, O., Murphy, C. & Manian, D. (2012), *Beginning HTML5 and CSS3 - The Web Evolved*, Apress, New York.

ECMA International (2011), 'Standard ecma-262 - ecmascript language specification'.
URL: *http://www.ecma-international.org/publications/standards/Ecma-262.htm*

Fenton, N. & Bieman, J. (2014), *Software Metrics: A Rigorous and Practical Approach, Third Edition*, CRC Press.

Fink, G. (2014), *Pro single page application development : using Backbone.js and ASP.NET*, Apress, Berkeley, CA New York, NY.

Forgue, M.-C. & Hazaël-Massieux, D. (2012), Mobile web applications: Bringing mobile apps and web together, *in* 'Proceedings of the 21st International

Conference Companion on World Wide Web', WWW '12 Companion, ACM, New York, NY, USA, pp. 255–258.
URL: *http://doi.acm.org.acm.perm.fh-joanneum.at/10.1145/2187980.2188022*

Freeman, A. (2012), *Pro JavaScript for web apps*, Apress Distributed to the book trade worldwide by Springer, Berkeley, CA New York.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns*, Addison Wesly, Reading, MA.

Green, B. & Seshadri, S. (2013), *AngularJS*, O'Reilly Media, Sebastopol, CA.

Hahn, E. (2013), *JavaScript testing with Jasmine*, O'Reilly Media, Sebastopol, CA.

Hales, W. (2012), *HTML5 and JavaScript Web Apps*, O'Reilly Media, Beijing Sebastopol, CA.

Hogan, B. (2013), *HTML5 and CSS3: Level Up with Today's Web Technologies*, The Pragmatic Bookshelf, Dallas.

Jungmayr, S. (2005), 'Testbarkeit.de', `http://www.testbarkeit.de/TestbarkeitDefinition.html`. [Online; accessed 21-Apr-2015].

Knol, A. (2013), *Dependency injection with AngularJS*, Packt Publishing Ltd, Birmingham, UK.

Liang, Y. (2010), *JavaScript testing beginner's guide test and debug JavaScript the easy way*, Packt Pub, Birmingham, UK.

Lloyd, I. (2008), *The ultimate HTML reference*, Sitepoint, Collingwood, Vic.

Lubbers, P., Albers, B. & Salim, F. (2011), *Pro HTML5 Programming*, 2. aufl. edn, Apress, New York.

Mikowski, M. S. & Powell, J. C. (2013), *Single Page Web Applications - JavaScript End-to-end*, Manning, Birmingham.

Nicolaou, A. (2013), 'Best practices on the move: Building web apps for mobile devices', *Commun. ACM* **56**(8), 45–51.
URL: *http://doi.acm.org.acm.perm.fh-joanneum.at/10.1145/2492007.2492023*

Oman, Hagemeister, e. a. (1991), 'A definition and taxonomy for software maintainability', Software Engineering Test Laboratory, University of Idaho, Moscow, ID, USA, Tech. Rep. #91-08-TR.

Osmani, A. (2012*a*), *Developing Backbone.js Applications*, O'Reilly Media, Beijing Sebastopol.

Osmani, A. (2012*b*), *Learning JavaScript Design Patterns*, O'Reilly Media, Sebastopol, CA.

Roemer, R. (2013), *Backbone.js Testing*, Packt Publishing Ltd, City.

Rogat, A. (1998), 'Qualitätskriterien für software', `http://www2.math.uni-wuppertal.de/~axel/skripte/oop/oop1_3.html`. [Online; accessed 18-Apr-2015].

Severance, C. (2012), 'Javascript: Designing a language in 10 days', *Computer* **45**(2), 7–8.
URL: *http://dx.doi.org/10.1109/MC.2012.57*

Tripathy, P. & Naik, K. (2014), *Software Evolution and Maintenance*, Wiley.

Xanthopoulos, S. & Xinogalos, S. (2013), A comparative analysis of cross-platform development approaches for mobile applications, *in* 'Proceedings of the 6th Balkan Conference in Informatics', BCI '13, ACM, New York, NY, USA, pp. 213–220.
URL: *http://doi.acm.org.acm.perm.fh-joanneum.at/10.1145/2490257.2490292*