Hochschule München

Studiengang Bachelor Wirtschaftsinformatik

Fakultät 07

Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Studienarbeit

Rich Internet Applications mit HTML5 und JavaScript

Sommersemester 2015

Thorsten Wirth

Matrikelnummer: 05466311

Lehrbeauftragter: Michael Theis

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Webanwendungen spielen eine immer größere Rolle in Unternehmen. Durch die weite Verbreitung des Internets und einer immer höheren Bandbreite können, mithilfe von JavaScript, Anwendungen erstellt werden, die in ihrem Verhalten normalen Desktopanwendungen immer ähnlicher werden.

Die Vorteile einer solchen Webanwendung ergeben sich aus der ständigen Erreichbarkeit der Anwendung, einer hohen Komplexität bei vergleichsweise geringen Kosten bei der Entwicklung, sowie die Benutzung und Zusammenarbeit vieler Nutzer. So ist in den letzten Jahren eine Vielzahl an JavaScript Frameworks aufgetaucht, die das Erstellen solcher komplexer Anwendungen erleichtern sollen.

Heutzutage haben Webentwickler zunehmend das Problem mit der rasanten Entwicklung Schritt zu halten. Für die unterschiedlichsten Bedürfnisse an eine Webanwendung die richtigen Techniken einzusetzen, fällt immer schwerer.

Ziel dieser Studienarbeit ist es, „Rich Internet Applications“ (RIA) mit HTML5 und JavaScript unter Verwendung des JavaScript-Frameworks AngularJS vorzustellen.

Zunächst werden die Grundlagen und Konzepte des Frameworks vorgestellt, sowie der Grundaufbau einer Webanwendung unter Verwendung von AngularJS. Die einzelnen Bestandteile des Frameworks werden im Anschluss anhand eines kleinen Projektes zur Erstellung einer ToDo-Liste erläutert und eingesetzt, so dass der Leser dieser Arbeit einen kleinen Einblick in die Welt von AngularJS erhält.

Im Anschluss an das Beispielprojekt folgt ein Vergleich mit anderen JavaScript Frameworks in denen die Stärken und Schwächen der jeweiligen Frameworks herausgearbeitet werden.

2 AngularJS - Grundlagen und Konzepte

Für den Einstieg in Angular, sollte man sich zuerst einmal mit den Grundlagen dieses Frameworks beschäftigen. AngularJS folgt einigen Grundprinzipien, die man kennen muss, um die Mechanismen verstehen zu können (Tarasiewicz & Böhm, 2014, S. 19). Auf diese Prinzipien wird im Verlauf dieser Arbeit noch näher eingegangen.

Das Framework setzt weiter ein gewisses Grundverständnis in JavaScript voraus, weswegen auch in dieser Arbeit auf JavaScript nicht näher eingegangen wird.

2.1 MVC-Muster

Das Model-View-Controller-Muster ist wohl das bekannteste Architekturmuster und sollte jedem Entwickler bekannt sein. Dieses Muster wurde bereits in den achtziger Jahren durch Trygve Reenskaug für Benutzeroberflächen in Smalltalk[1] beschrieben und dient zur Trennung der Geschäftslogik (Controller), Datenhaltung (Model) und der Präsentation (View) und dazu sie in verschiedene Schichten zu separieren (Tarasiewicz & Böhm, 2014, S. 19).

Dieses Konzept in der Softwareentwicklung fand auch seinen Weg in die Webentwicklung. Bei klassischen Anwendungen stellt der Client einen HTTP-Request, den ein Controller auf dem Server verarbeitet und als Antwort wieder HTML an den Client liefert. Da die Antwort immer nur auf Anfrage des Clients folgen kann, liegt die Verantwortung der View-Schicht größtenteils auf dem Server. Bei

[1] Smalltalk gehört zu den ersten objektorientierten Programmiersprachen.

Anwendungen mit großer Interaktivität mit dem Benutzer zeigt diese Lösung ihre Nachteile, da hier die Seiten immer wieder neugeladen werden müssen.

Mittlerweile werden viele Webanwendungen als Rich Internet Application (RIA) konzipiert und bieten dem Benutzer eine große Interaktionsmöglichkeit mit der Anwendung. Hierbei wird der Anteil der Logik die auf dem Browser liegt immer größer. Der Server ist nur noch Datenquelle und Datenspeicher, welche über eine REST-Schnittstelle[2] bereitgestellt werden. Der Browser kann dann über ein JavaScript-Framework darauf zugreifen.

Da die Anwendung immer mehr zum Client verlagert wird hat sich das MVC-Muster auch hier durchgesetzt und wurde durch das MVVM-Muster erweitert.

2.2 Das MVVM-Muster

Das MVVM-Muster (Model-View-ViewModel-Muster) begründet sich darin, dass das Model über die REST-Schnittstelle zwar erreichbar ist, aber nur die Daten, die auch benötigt werden angezeigt werden sollen. Diese Aufgabe übernimmt das ViewModel. Das ViewModel ermöglicht es, Daten die z.B. von einem Server kommen, vor ihrer Ausgabe in der View noch zu transformieren. Dies geschieht mit Hilfe der Scopes und ermöglicht die Zwei-Wege-Datenbindung. Weiter lässt sich im ViewModel die Logik für die View hinterlegen. Diese Trennung erhöht die Testbarkeit der View sowie ihre Austauchbarkeit. Die folgende Abbildung (Abb. 2.1) soll dies verdeutlichen.

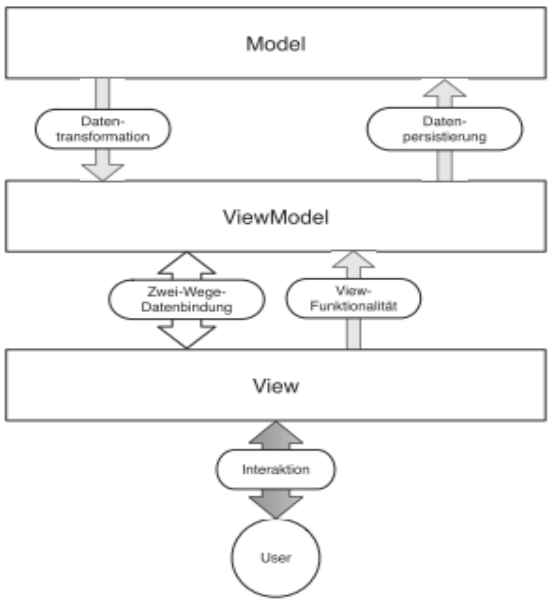

Abbildung 2.1 Das Model-View-ViewModel-Muster (Tarasiewicz & Böhm, 2014, S. 22)

2.3 Zwei-Wege-Datenbindung und Scopes

Ein weiteres Konzept von AngularJS ist die Zwei-Wege-Datenbindung. Sie ermöglicht es Änderungen, die ein Nutzer in der Benutzeroberfläche anstößt, direkt im Datenmodell abzubilden. Wiederum wird, bei Änderungen am Datenmodell, die Ansicht aktualisiert. Wie diese Datenbindung im Detail umgesetzt wird, soll später im Beispielprojekt noch gezeigt werden. Vorab sei nur erwähnt, dass dies im Wesentlichen durch die Verwendung von Scopes ermöglicht wird. Scopes sind Objekte in denen Variablen und Funktionen enthalten sind. Dieser Scope bezieht sich auf einen bestimmten Abschnitt im DOM[3]. Die Datenbindung kann auch nur bei Variablen und Funktionen hergestellt werden, die in einem Scope definiert sind. Damit AngularJS erkennt welche Daten sich geändert haben, kommt die

[2] REpresentational State Transfer
[3] Document Object Model

sogenannte „Dirty Checking“ Methode zum Tragen. Diese vergleicht die Variablen der View mit Variablen des ViewModel und synchronisiert sie bei Ungleichheit.

2.4 Dependency Injection

Ein weiteres Muster in der Softwareentwicklung ist die Dependency Injection. Hier wird die Auflösung der Abhängigkeiten an eine zentrale Stelle übergeben. Sie stellt zur Laufzeit dann die benötigten Objekte zur Verfügung. Dies erhöht die Austauschbarkeit und Testbarkeit der Anwendung. Die Dependency Injection (DI) gehört zu den Leitkonzepten in AngularJS (Tarasiewicz & Böhm, 2014, S. 27). Weiter kann man sagen, dass DI eines der Alleinstellungsmerkmale von AngularJS unter den JavaScript Frameworks ist (Wießeckel, 2015).

2.5 Testbarkeit

Wie im obigen Abschnitt bereits erwähnt, lassen sich durch die Verwendung von DI die Anwendungen gut testen. Testbarkeit ist ein großes Thema in diesem Framework. Von Haus aus hat AngularJS ein Konzept zum Erstellen von Unit-Tests an Bord. Des Weiteren lassen sich E2E-Tests[4] erstellen, mit welchen die Anwendung als Black-Box angesehen wird. Somit kann das Verhalten der ganzen Anwendung getestet werden. Die unten aufgeführte Abbildung zeigt einen solchen E2E-Test.

```
describe('User Login', function() {
  it('should successfully login the user', function() {
    input('user').enter('John');
    input('password').enter('123asd');
    element(':button').click();
    expect(element('h1').text()).toEqual('Hello, Tom!');
  });
});
```

Abbildung 2.2 E2E-Test für einen Benutzer-Login (Tarasiewicz & Böhm, 2014, S. 28)

3 Bestandteile von AngularJS

Die folgenden Abschnitte befassen sich mit den Bestandteilen oder auch Bausteinen des Frameworks, die in einer typischen AngularJS-Anwendung eingesetzt werden. Diese sollen hier nur kurz vorgestellt werden um einen groben Überblick über Funktion und Verwendungszweck zu geben. Eine feingranulare Betrachtung würde den Rahmen dieser Arbeit sprengen. Für einen tieferen Einblick sei hier die Webseite von AngularJS unter https://angularjs.org/ genannt. Dort gibt es eine ausführliche Dokumentation.

3.1 Module

Module bilden einen Rahmen, in den zusammenhängende Controller, Direktiven, Services und Filter gekapselt werden. Auch bei den Modulen wird die DI angewendet, so dass in einem Modul weitere Module eingebunden werden können.

3.2 Controller

Controller sind in AngularJS eigentlich nur ein Konstruktor für Scopes. Die Scopes stellen wiederum die Daten und Logik, die für eine bestimmte Ansicht benötigt werden, bereit. Bei Controllern sollte man, laut den Autoren von AngularJS, Tarasiewicz und Böhm, und Ingo Rammer in einem YouTube Video im Rahmen der MobileTech Conference 2013 (Rammer, 2014), sich an einige Regeln halten. Zum einen

[4] End-to-End-Tests

sollte man die Controller so schmal wie möglich halten und komplexe Logik in Services implementieren und diese via DI in den Controller einbinden. Weiter sollte der Code aussagekräftig sein. Der wichtigste Punkt und als absolutes No-Go zu sehen, sind DOM-Manipulationen. Diese sollten, wenn nicht vermeidbar und mit den Mitteln die AngularJS schon mitbringt, in eigene Direktiven umgesetzt werden. Diese Regel findet sich auch auf der Webseite von AngularJS wieder.

„Custom directives to access the DOM: In Angular, the only place where an application should access the DOM is within directives. This is important because artifacts that access the DOM are hard to test. If you need to access the DOM directly you should write a custom directive for this. The directives guide explains how to do this." (Google, 2015)

3.3 Models

Models stellen in AngularJS gewöhnliche JavaScript-Datentypen dar. Das Framework gibt hier keine besonderen Klassen vor, von denen geerbt werden müsste. Die Models werden in einem Scope definiert um die Zwei-Wege-Datenbindung herzustellen (Tarasiewicz & Böhm, 2014, S. 31).

3.4 Routen

Da AngularJS weitestgehend bei Single Page Applications zum Einsatz kommt, stellt sich die Frage, wie man später auf bestimmte Teile der Anwendung mittels der URL zugreifen kann (Deep Link). Auch die Weitergabe von Links auf bestimmte Abschnitte der Applikation würde nicht funktionieren. Hier liefert das Framework das Routing.

```
angular.module('myApp').config(function ($routeProvider) {
    $routeProvider
        .when('/', {
            templateUrl: 'templates/mainTemplate.html',
            controller: 'MainCtrl'
        })
        .when('/user/:userId', {
            templateUrl: 'templates/userDetailsTemplate.html',
            controller: 'UserDetailsCtrl'
        })
        .when('/user', {
            templateUrl: 'templates/userOverviewTemplate.html',
            controller: 'UserOverviewCtrl'
        })
        .otherwise({
            redirectTo: '/'
        });
});
```

Abbildung 3.1 Routendefinitionen für eine AngularJS-Anwendung (Tarasiewicz & Böhm, 2014, S. 32)

Wie in Abbildung 3.1 zu sehen ist, kann man auf einer bestimmten URL ein Template und einen Controller definieren. Der URL können auch Parameter mitgegeben werden. Diese werden mit einem vorangestellten Doppelpunkt eingeleitet. Die otherwise-Funktion liefert hier einen Fallback, der ausgelöst wird, falls eine Anfrage nicht zu den definierten Routen passt.

3.5 Templates und Expressions

Templates sollten mittlerweile jedem Entwickler etwas sagen, weswegen es in dieser Arbeit nicht näher vertieft werden soll. Nur so viel, dass sie Teile des HTML-Codes enthalten, um mit ihnen die Webseite zusammen zu setzen.

Wichtiger ist die Betrachtung der Expressions, die für die Zwei-Wege-Datenbindung unerlässlich sind.

```
angular.module('myApp')
.controller('MyCtrl', function ($scope) {
  $scope.user = {
    name: 'John Doe',
    age: 27,
    email: 'john@doe.com'
  };

  $scope.family = [
    'James Doe', 'Clarissa Doe', 'Ted Doe'
  ];

   $scope.loggedIn = true;
});


<div>
  <p>Hello, {{user.name}}!</p>
</div>
```

Abbildung 3.2 UnserMyCtrl-Controller (Tarasiewicz & Böhm, 2014, S. 34-35)

Wie in obiger Abbildung zu sehen ist, werden zunächst erstmal Daten in einem Scope definiert. Dieser Scope gehört zu dem Controller „myCtrl“. Weiter unten in der Abbildung ist ein kleines Template mit der Expression, die den Namen aus dem Scope ausgibt. Expressions werden in zwei geschweifte Klammern geschrieben.

3.6 Filter

Filter werden in AngularJS eingesetzt um Expressions vor der Ausgabe zu transformieren. Es gibt zum einen die Formatierungsfilter und zum anderen die Collection-Filter. Formatierungsfilter, wie der Name schon vermuten lässt, formatieren die Expression vor ihrer Ausgabe z.B. in Großbuchstaben oder Kleinbuchstaben (lowercase, uppercase). Collection-Filter finden ihre Anwendung in Verbindung mit der Direktive ng-repeat. So lassen sich Collections nach bestimmten Parametern filtern.

```
<div>
  <p>
    Hello, {{user.name | uppercase}}!<br>
    Five years ago you were
    {{user.age - 5}} years old.
  </p>
</div>
```

Abbildung 3.3 Eine Expression mit angewandtem Filter (Tarasiewicz & Böhm, 2014, S. 36)

Abbildung 3.3 zeigt die Verwendung eines Formatierungsfilters, der in diesem Fall die Ausgabe in Großbuchstaben transformiert. Filter werden in AngualrJS mit einem Pipe („|“) eingeleitet, welches z.B. aus der Shell-Syntax bekannt ist.

```
<div>
    Family members of {{user.name}}:
    <ul>
        <li ng-repeat="member in family | filter:'clarissa'">
          {{member}}
        </li>
    </ul>
</div>
```

Abbildung 3.4 Verwendung von Filtern innerhalb von ng-repeat (Tarasiewicz & Böhm, 2014, S. 39)

In Abbildung 3.4 wird ein Collection-Filter eingesetzt, welcher in Verbindung mit der Direktiven „ng-repeat“, uns nur Familienmitglieder ausgibt, in dessen Namen „Clarissa“ vorkommt. Wie in Abbildung 3.2 zu sehen war, ist im Scope ein family Array definiert, welches sich mit „ng-repeat“ durchlaufen lässt und der foreach-Schleife aus Programmiersprachen ähnelt.

3.7 Services

Wie unter 3.2 schon erwähnt wurde sind Controller in AngularJS schmal zu halten und komplexe Logik sollte ausgelagert werden. Die würde ihren Platz in den Services finden. Services lassen sich vom Controller problemlos einbinden und Services können sogar andere Services einbinden. Eine häufige Verwendung der Services ist es, sie als DAO[5] einzusetzen die einen REST-Endpunkt ansprechen um z.B. Daten zu speichern.

Services werden in einer Anwendung genau einmal instanziert (Singleton-Pattern) und eignen sich dadurch, Daten und Logik in mehreren Controllern zu verwenden. Die folgende Abbildung zeigt die Definition eines Log-Services.

```
angular.module('myApp').factory('log', function() {
  // Service Implementation
  var log = function(level, message) {
    console.log('[' + level + '] ' + message);
  };

  // Public API
  return {
    error: function(message) {
      log('ERROR', message);
    },
    info: function(message) {
      log('INFO', message);
    },
    debug: function(message) {
      log('DEBUG', message);
    }
  };
});
```

Abbildung 3.5 Definition eines log-Service (Tarasiewicz & Böhm, 2014, S. 43)

Wie man in Abbildung 3.5 sieht, wird hier der Service mit der Funktion „factory()“ definiert und nicht mit einer Funktion namens „service()“. Es gibt insgesamt fünf Möglichkeiten einen Service zu

[5] Data Access Object

definieren, die in der nächsten Abbildung 3.6 kurz vorgestellt werden. Hier sei noch erwähnt, dass die „factory()“-Funktion wohl die am häufigsten verwendete Funktion ist.

Servicedefinition	API	Beschreibung
Provider	`provider(...)`	Grundlegendes API zur Servicedefinition. Kann in der Konfigurationsphase konfiguriert werden. Muss eine Funktion `$get()` bereitstellen, die die entsprechende Serviceinstanz zurückgibt. Kann mit einer Konstruktorfunktion oder mit einem Objekt aufgerufen werden.
Factory	`factory(...)`	Baut auf dem Provider-API auf. Abstrahiert von dem Aspekt der Konfigurierbarkeit. Erlaubt somit eine bequemere Servicedefinition, falls Konfigurierbarkeit keine Rolle spielt.
Konstruktorfunktion	`service(...)`	Baut auf dem Factory-API auf. Instanzierung der Serviceinstanz mithilfe des `new`-Operators in Verbindung mit der übergebenen Konstruktorfunktion. Somit lässt sich ein Objekt einer bestehenden Klasse als Service registrieren.
Wert als Service	`value(...)`	Baut auf dem Factory-API auf. Erlaubt die Registrierung eines Wertes als Service. Dabei kann der Wert ein primitiver Datentyp oder ein Objekt oder eine Funktion sein.
Konstante als Service	`constant(...)`	Registrierung eines konstanten Wertes als Service. Dieser Wert kann zur Laufzeit nicht mehr verändert werden.

Abbildung 3.6 Die fünf Möglichkeiten zur Servicedefinition (Tarasiewicz & Böhm, 2014, S. 44)

3.8 Direktiven

An dieser Stelle soll nun der letzte Baustein, die Direktiven, vorgestellt werden. Direktiven sind ein sehr komplexes Thema in AngularJS und sind in keinem anderen Framework zu finden. Mithilfe von Direktiven kann HTML beliebig um weitere Tags oder Attribute erweitert werden.

So kann z.B. hinter einem Tag oder Attribut ein ganzer Unterteil des DOM mit entsprechender Logik stehen (Tarasiewicz & Böhm, 2014, S. 50). AngularJS bringt schon eine ganze Palette von Direktiven mit. Die „ng-repeat“ ist z.B. eine von Vielen. Mittlerweile gibt es schon einige gut gefüllte Bibliotheken mit zusätzlichen Direktiven, wie z.B. auf der Webseite von Angular-UI, die unter https://angular-ui.github.io/ zu erreichen ist.

Bei der Erstellung eigener Direktiven sei noch auf die Namenskonvention hingewiesen. Wie in Abbildung 3.7 zu sehen ist, fängt bei der Definition einer Direktive der Direktivenname mit einem Kleinbuchstaben an und folgt dann der CamelCase-Notation, d.h. sollte der Name aus mehreren Worten bestehen, wird der Anfangsbuchstabe des nächsten Wortes groß geschrieben. Im Template hingegen werden die Worte mit Bindestrich („-„), Doppelpunkt („:“) der Unterstrich („_“) getrennt.

```
angular.module('myApp').directive('colorPicker', function () {
  return function(scope, element, attrs) {
    console.log("It's me, colorPicker!");
  }
});

<div color-picker></div>
<div color-picker></div>
<div color-picker></div>
```

Abbildung 3.7 Definition der colorPicker-Direktive durch Rückgabe einer Link-Funktion und Verwendung im Template (Tarasiewicz & Böhm, 2014)

Weiter ist in Abbildung 3.7 zu sehen, dass die Direktivendefinition eine Funktion zurückgibt. Das ist eine der zwei Möglichkeiten und nennt sich Link-Funktion. Die zweite Möglichkeit ist es, ein Direktiven-Definitions-Objekt (DDO) zurück zu geben. In diesem können Verhaltensweisen detaillierter festgelegt werden. Aus diesem Grund ist diese für Anfänger in AngularJS auch wesentlich komplexer. Die folgende Abbildung 3.8 zeigt eine Liste mit den Eigenschaften eines solchen DDO.

DDO-Eigenschaft	Mögliche Werte	Beschreibung
`link`	Funktion mit max. 4 Parametern	Definition der Link-Funktion, die für jede Direktiveninstanz genau einmal aufgerufen wird. AngularJS übergibt der Funktion beim Aufruf 3 oder 4 Parameter, abhängig von der `require`-Eigenschaft: `scope` – Scope, `element` – HTML-Element, `attrs` – Zugriff auf die HTML-Attribute, `ctrl` – Controller (falls `require` gesetzt ist).
`compile`	Funktion mit max. 3 Parametern	Definition der Compile-Funktion, die für jede Direktive (nicht Instanz!) genau einmal aufgerufen wird. AngularJS übergibt der Funktion beim Aufruf 3 Parameter: `tElement` – Template-Element, `tAttrs` – Zugriff auf Template-Attribute, `transclude` – Link-Funktion für die Transklusion.
`restrict`	String	Beschränkung des Geltungsbereiches auf Attribute (`A`), Elemente (`E`), CSS-Klassen (`C`), Kommentare (`M`) oder eine Kombination daraus
`template`	String	Definition des HTML-Templates
`templateUrl`	String	Definition der URL zum HTML-Template, falls das Template komplexer ist und in einer separaten Datei liegt
`scope`	boolean oder Objekt	Scope-Definition entsprechend der besprochenen Möglichkeiten
`replace`	boolean	Soll bei der Nutzung der Direktive das Direktiven-Tag durch das Direktiven-Template ersetzt werden (`true`) oder nicht (`false`)?
`require`	String	Definition einer Controller-Abhängigkeit einer anderen Direktive
`controller`	Funktion	Komplexere Direktiven können eigene Controller haben.
`transclude`	boolean	Definition einer Transklusion, falls wir beim Aufruf der Direktive im Direktiven-Tag einen `innerHTML`-Bereich erlauben wollen. Nutzung in Verbindung mit der `ngTransclude`-Direktive.
`terminal`	boolean	Festlegung, ob Direktiven im Template der Direktive verarbeitet werden sollen (`false`) oder nicht (`true`)
`priority`	Number	Festlegung der Verarbeitungsreihenfolge, falls mehrere Direktiven auf ein und dasselbe DOM-Element wirken. Direktiven mit höherer Priorität haben in der Verarbeitungsreihenfolge Vorrang gegenüber den Direktiven mit niedrigerer Priorität. Bei Direktiven mit gleicher Priorität ist die Verarbeitungsreihenfolge *nicht* definiert.

Abbildung 3.8 Überblick über die DDO-Eigenschaften einer Direktive (Tarasiewicz & Böhm, 2014, S. 71)

4 Ein kleines Projekt

In diesem Abschnitt soll nun eine kleine App entwickelt werden, die zeigen soll, was man mit relativ wenig Code in AngularJS, dank der Datenbindung, umsetzten kann. Entwickelt werden soll eine ToDo-App in der Tasks hinzugefügt werden können und auch wieder entfernt werden können. Dieses Beispiel ist sehr stark an das AngularJS Tutorial von John Lindquist angelehnt (Lindquist, 2012). Hier wurden

nur kleine Änderungen in Bezug auf den verwendeten Filter eingefügt. Weiter wurde der Controller in eine separate JS-Datei ausgelagert.

4.1 Vorbereitung

Für die Entwicklung wurde die IDE WebStorm verwendet, weil diese für JavaScript ausgelegt ist und auch für AngularJS einen guten Support bietet.

Damit die App auch auf einem lokalen Webserver laufen kann wurde nun von der Seite https://nodejs.org/ Node.JS heruntergeladen und installiert. Nach der Installation kann mit Hilfe des Konsolenbefehls „npm install -g http-server“ das HTTP-Server-Modul von NodeJS installiert werden.

Jetzt kann man, in dem Verzeichnis, in dem die Applikation liegen soll, mit dem Konsolenaufruf „http-server“ den Webserver starten. Dieser ist unter http://localhost:8080 erreichbar (s. Abb. 4.1).

```
C:\Users\Thorsten\WebstormProjects\todo\app>http-server
Starting up http-server, serving ./ on: http://0.0.0.0:8080
Hit CTRL-C to stop the server
```

Abbildung 4.1 Starten des Webservers

Im Anschluss wurde die Verzeichnisstruktur angelegt. Wie die Abbildung 4.1 zeigt, wurde ein Verzeichnis mit dem Namen des Projekts angelegt. In diesem Verzeichnis wurde nun ein Ordner mit dem Namen „app“ erstellt. Dies ist auch der Ort, an dem der Webserver gestartet wird und der die Index.html enthält.

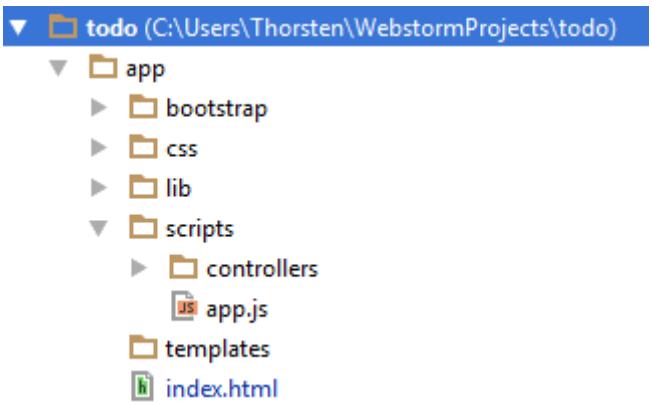

Abbildung 4.2 Verzeichnisstruktur der todoApp

Im Unterordner „bootstrap“ legen wir die Dateien vom Framework Bootstrap ab, welches verwendet wird, um die Anwendung mittels den CSS-Klassen von Bootstrap ansehnlicher zu gestalten. Bootstrap in der hier verwendeten Version ist unter http://getbootstrap.com/2.3.2/index.html herunterladbar.

Bootstrap ist eines der bekanntesten CSS-Frameworks. Es bietet einige Vorlagen für Buttons, Tabellen und Formulare. Bootstrap bietet auch Javascript-Erweiterungen an. Es wird oft für die Gestaltung im „responsive Webdesign“ verwendet. In diesem kleinen Projekt werden nur ein paar CSS-Klassen benötigt, die die Anwendung ansprechender machen sollen.

Im Ordner „lib“ legen wir im Unterverzeichnis angular die Datei angular.js ab. Den übrigen Verzeichnissen wird sich im Verlauf dieser Beschreibung gewidmet.

4.2 Die index.html

Im nächsten Schritt soll nun die index.html erstellt werden. Wie in Abbildung 4.3 zu sehen ist, werden die Skripte von AngularJS und Bootstrap schon eingebunden. An dem HTML-Tag ist zu sehen, dass innerhalb dieses Elements die Angularanwendung laufen soll. Es ist natürlich möglich Angular nur innerhalb eines „div-Containers“ laufen zu lassen. Weiter wird dem ersten „div“ schon ein Controller zugewiesen. In den nächsten Schritten werden das Modul „todoApp“ und der Controller TodoCtrl erstellt.

```
<!DOCTYPE html>
<html ng-app="todoApp">
<head lang="en">
    <meta charset="UTF-8">
    <link rel="stylesheet" href="bootstrap/css/bootstrap.css">
    <title>ToDoListe - Eine AngularJS Einführung<</title>
</head>
<body>
<div ng-controller="TodoCtrl">

</div>


<!-- Scripts -->
<script src="lib/angular/angular.js"></script>
<script src="bootstrap/js/bootstrap.js"></script>
</body>
</html>
```

Abbildung 4.3 Einstieg mit der index.html

4.3 Das Modul „todoApp“

In der folgenden Abbildung wird das Modul erstellt. Für dieses kleine Projekt sind sonst keine weiteren Schritte, die das Modul betreffen, notwendig (s. Abb. 4.4). Dieser Code wird in der app.js im Unterordner „scripts“ gespeichert.

```
var todoApp = angular.module('todoApp', []);
```

Abbildung 4.4 Erstellen des Moduls

4.4 Der Controller „ToDoCtrl“

Im Ordner „scripts“ wird nun ein weiterer Unterordner „controllers“ erstellt, in dem die Datei „todo.js“ aufgerufen wird. Dort wird der Controller für das Modul „todoApp“ definiert. Im Scope des Controllers wird nun ein Array mit den ersten „Todos“ erzeugt (s. Abb. 4.5). Weiter erhält der Scope noch die Funktion „getTotalTodos“ welcher die Gesamtzahl an Aufgaben zurückgeben soll. Eine weitere Funktion in diesem Scope ist die „clearCompleted“. In dieser wird ein Filter definiert, welcher nur die Aufgaben zurückgibt, in denen das Attribut „done“ mit dem Wert „false“ belegt ist.

Zuletzt folgt noch die Funktion, die es ermöglicht eine neue Aufgabe hinzuzufügen. Hier ist die Zwei-Wege-Datenbindung gut zu sehen, denn die Variable „formTodoText“ ist nicht im Controller definiert, sondern stammt aus der index.html (s. Abb. 4.6) und wurde dort an das Modul gebunden.

```
todoApp.controller("TodoCtrl",function($scope){


    $scope.todos = [
        {text: 'Einkaufen',done:false},
        {text:'Lernen',done:false}
    ];

    $scope.getTotalTodos = function(){
        return $scope.todos.length;
    };
    $scope.clearCompleted = function(){

        $scope.todos = $scope.todos.filter(function(item){
            return !item.done;

        })

    }
    $scope.addTodo = function(){
        $scope.todos.push({text:$scope.formTodoText,done:false});
        $scope.formTodoText = "";
    };
});
```

Abbildung 4.5 finaler ToDoCtrl

Wie in u.a. Abbildung 4.6 gezeigt, kommt hier die Direktive „ng-repeat“ zum Einsatz. Diese läuft durch das Array, welches im Scope definiert wurde und erstellt für jedes Element eine „checkbox“ und einen „span“ der den Text ausgibt. Dies wird durch die Verwendung der Expressions ermöglicht. Auch die Gesamtzahl der Aufgaben wird durch die Verwendung der Funktion „getTotalTodos“ in einer Expression dargestellt.

```
<div class="wrapperTodoApp">
<div ng-controller="TodoCtrl">
    <h2>Aufgaben: {{getTotalTodos()}}</h2>
    <ul class="unstyled">
        <li ng-repeat=" todo in todos">
            <input type="checkbox" class="icon-check" ng-model="todo.done">
            <span class="done-{{todo.done}}">{{todo.text}}</span>
        </li>
    </ul>
    <form class="form-horizontal">
        <input type="text" ng-model="formTodoText" ng-model-instant>
        <button class="btn" ng-click="addTodo()"><i class="icon-plus"></i>Add </button>

    </form><br>
    <button class="btn-large" ng-click="clearCompleted()"><i class="icon-trash"></i>Entferne erledigete Aufgaben</button>
</div>
</div>
```

Abbildung 4.6 TodoCtrl in der index.html

Zum Ende der Vorstellung des Projekts ist in der nächsten Abbildung (s. Abb. 4.7) das Resultat im Browser zu sehen und zeigt, dass sich mit diesem wenigen Code, dank der Datenbindung, nützliche Anwendungen erstellen lassen.

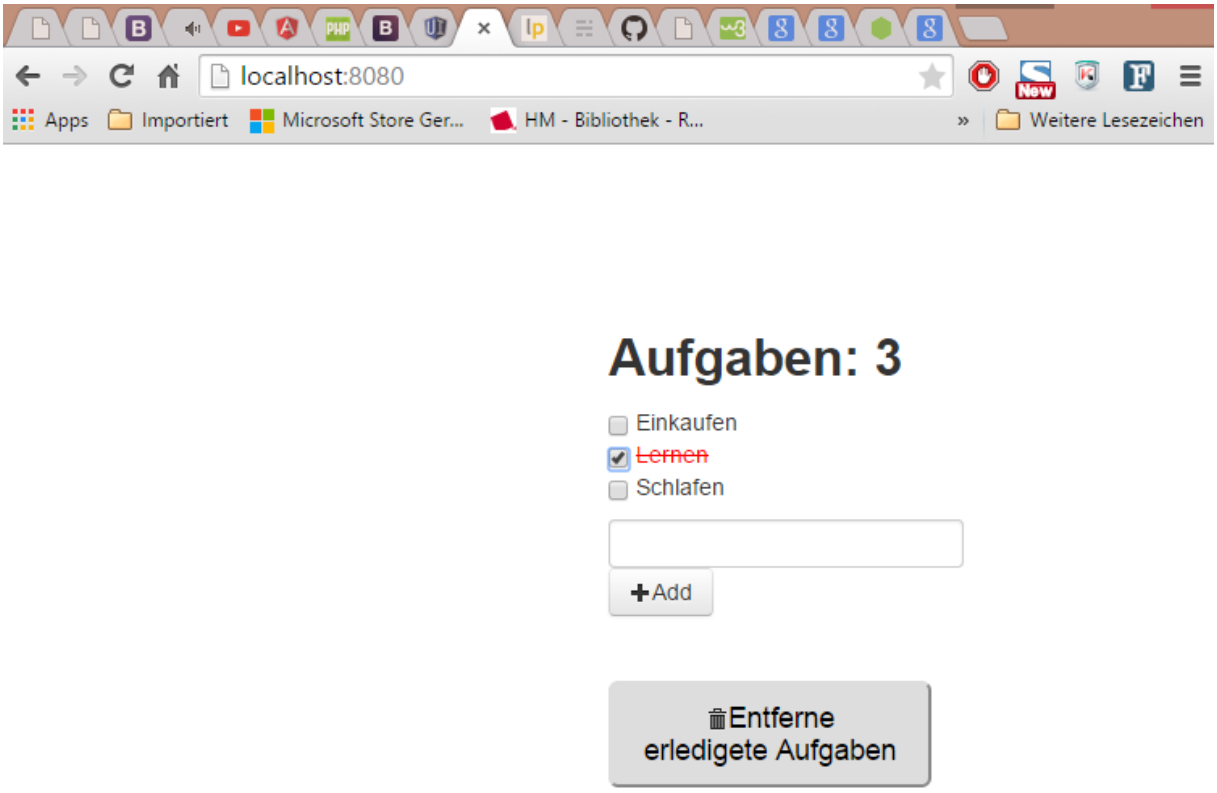

Abbildung 4.7 Darstellung der App im Browser

5 Fazit und Ausblick

Das Ziel dieser Arbeit war es Rich Internet Applications mit HTML5 und JavaScript unter Verwendung des Frameworks AngularJS vorzustellen. In der ersten Phase der Einarbeitung in das Framework konnte ich bei mir eine steile Lernkurve feststellen, so dass meine Motivation sehr hoch war. Nun wurde im Vorwort des Buches „AngularJS“ schon darauf hingewiesen, dass die Verwendung des Frameworks Kenntnisse in JavaScript voraussetzt. Diese, so muss ich gestehen, sind bei mir nicht besonders ausgeprägt, da ich vorher JavaScript nur unter Verwendung von jQuery gebraucht habe.

Aus diesem Grund wurde es für mich zunehmend schwieriger tiefer in das Thema vorzudringen, da in den Beispielen zur Erstellung eigener Direktiven, Filter oder Services zum Teil komplexere JavaScript-Funktionen zum Einsatz kamen. Diesen konnte ich nur schwer folgen. Auch das Zusammenwirken von Modulen, Controllern und Services forderte doch einiges an Zeit, um diese nachvollziehen zu können.

Ein positiver Aspekt war allerding die Zwei-Wege-Datenbindung, die den Code sehr schlank hält und man sich keine Gedanken darüber machen muss, wie man seine Daten aktualisieren kann. Weiter lassen sich alle Bausteine in separaten Dateien einfach auslagern, was die Übersichtlichkeit deutlich steigert.

In dieser Arbeit wäre ich gerne noch auf das Thema der Testbarkeit eingegangen, welches ich aus Platzgründen aber leider nicht mehr mit aufnehmen konnte. Gerade die E2E-Tests sind ein sehr interessanter Punkt. Leider fehlten mir die nötigen Kenntnisse, diese in das Beispielprojekt zu integrieren.

Einen Vergleich zu anderen Frameworks kann ich in dieser Arbeit nicht aufstellen, da dies mein erstes JavaScript Framework war, wenn man jQuery nicht hinzunimmt.

Abschließend kann ich sagen, dass AngularJS auf jeden Fall mein Interesse geweckt hat. Dies liegt zum einen daran, dass es einen sehr ausgereiften Eindruck hinterlassen hat und die Applikation sehr aufgeräumt ausschauen lässt und zum anderen, dass es eine sehr große Community im Internet gibt und eine Fülle von Tutorials in Form von Videos und Beispielcodes. Allerdings werde ich mich vorher noch näher mit JavaScript befassen müssen, um alle Beispiele nachvollziehen zu können.

AngularJS eignet sich sicher nicht für jede Art einer Webanwendung. Meiner Meinung nach ist das Framework für Single-Page-Applikationen ausgelegt, in denen eine Fülle von Interaktionen mit dem Benutzer stattfindet.

6 Literaturverzeichnis

Google. (05. Mai 2015). *AngularJS: Developer Guide: Conceptual Overview*. Von https://docs.angularjs.org/guide/concepts#model abgerufen

Lindquist, J. (04. 03 2012). *Youtube.* Abgerufen am 05. 05 2015 von https://www.youtube.com/watch?v=WuiHuZq_cg4

Rammer, I. (10. 07 2014). *AngularJS -- Schmerzlos mit HTML5 und JavaScript.* (MobileTechCon, Hrsg.) Abgerufen am 05. 05 2015 von https://www.youtube.com/watch?v=ayG-xFe5seU

Tarasiewicz, P., & Böhm, R. (2014). *AngularJS.* Heidelberg: dpunkt.verlag GmbH.

Wießeckel, T. (05. Mai 2015). *AngularJS: Über Konzepte und die Zukunft von Web-Apps*. Von http://phpmagazin.de/artikel/angularjs-web-apps-interview-176381 abgerufen

7 Anhänge

7.1 Index.html

```
<!DOCTYPE html>
<html ng-app="todoApp">
<head lang="en">

    <link rel="stylesheet" href="bootstrap/css/bootstrap.css">
    <link rel="stylesheet" href="css/todo.css">
    <meta charset="UTF-8">
    <title>ToDoListe - Eine AngularJS Einführung</title>
</head>
<body>
<div class="wrapperTodoApp">
<div ng-controller="TodoCtrl">
    <h2>Aufgaben: {{getTotalTodos()}}</h2>
    <ul class="unstyled">
        <li ng-repeat=" todo in todos">
            <input type="checkbox" class="icon-check" ng-model="todo.done">
            <span class="done-{{todo.done}}">{{todo.text}}</span>
        </li>
    </ul>
    <form class="form-horizontal">
        <input type="text" ng-model="formTodoText" ng-model-instant>
        <button class="btn" ng-click="addTodo()"><i class="icon-plus"></i>Add </button>

    </form><br>
    <button class="btn-large" ng-click="clearCompleted()"><i class="icon-trash"></i>Entferne erledigete Aufgaben</button>
</div>
</div>

<!-- Scripts -->

<script src="lib/angular/angular.js"></script>
<script type="text/javascript" src="scripts/app.js"></script>
<script type="text/javascript" src="scripts/controllers/todoController.js"></script>



</body>
</html>
```

7.2 app.js

```
var todoApp = angular.module('todoApp', []);
```

7.3 todoController.js

```
todoApp.controller("TodoCtrl",function($scope){


    $scope.todos = [
        {text: 'Einkaufen',done:false},
        {text:'Lernen',done:false}
    ];

    $scope.getTotalTodos = function(){
        return $scope.todos.length;
    };
    $scope.clearCompleted = function(){

        $scope.todos = $scope.todos.filter(function(item){
            return !item.done;

        })

    }
    $scope.addTodo = function(){
        $scope.todos.push({text:$scope.formTodoText,done:false});
        $scope.formTodoText = "";
    };
});
```

7.4 todo.css

```
.done-true{
    text-decoration: line-through;
    color: red;
}

.wrapperTodoApp{
    position: absolute;
    width: 200px;
    height: 150px;
    top: 20%;
    left: 50%;
}
```

8 Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Studienarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Wirth, Thorsten

Fürstenfeldbruck, den 08.05.2015