Hochschule Bremen
Fakultät für Elektrotechnik und Informatik

Untersuchung und Evaluierung der Möglichkeiten für die Realisierung automatisierter Tests für AngularJS-Webanwendungen

Bachelorthesis
zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
im Dualen Studiengang Informatik

Autor Nikolas Schreck <nikolasschreck@gmail.com>

Version vom: 10. Mai 2017

Erstprüfer Prof. Dr.-Ing. Heide-Rose Vatterrott
Zweitprüfer Dipl.-Inf. Jochen Schwitzing

Sperrvermerk

Die vorliegende Prüfungsarbeit enthält vertrauliche Daten der Commerz Systems GmbH, die der Geheimhaltung unterliegen. Die Prüfungsarbeit wird an der Hochschule Bremen ausschließlich solchen Personen zugänglich gemacht, die mit der Abwicklung des Prüfungsverfahrens betraut sind und zur Verschwiegenheit verpflichtet sind. Es wird darauf hingewiesen, dass, sofern der Verfasser die Bewertung seiner Arbeit angreift, die Arbeit gegebenenfalls dem Widerspruchsausschuss zugeleitet werden muss, wobei die Mitglieder des Widerspruchsausschusses zur Verschwiegenheit verpflichtet sind. Wird die Bewertung der Arbeit gerichtlich angegriffen, so ist die Arbeit als Teil des Verwaltungsvorgangs dem Gericht zu übermitteln. Veröffentlichung und Vervielfältigung der vorliegenden Prüfungsarbeit – auch nur auszugsweise und gleich in welcher Form – bedürfen der schriftlichen Genehmigung der Commerz Systems GmbH.

Lorem ipsum dolorem sit amet.

Nikolas Schreck

Abstract

In dieser Bachelor-Thesi

Inhaltsverzeichnis

Abbildungsverzeichnis **8**

Tabellenverzeichnis **8**

Listingverzeichnis **9**

1 Motivation **10**

2 Grundlagen **11**
2.1 Test . . . 11
2.1.1 Komponententest . . . 11
2.1.2 Integrationstest . . . 12
2.1.3 Systemtest . . . 13
2.2 AngularJS . . . 14
2.2.1 Architektur . . . 14
2.2.2 Two-Way-Databinding . . . 15
2.2.3 Scopes . . . 16
2.2.4 Dependency Injection . . . 16
2.3 Node.js . . . 18
2.3.1 Laufzeitumgebung . . . 18
2.3.2 npm . . . 18

3 Auswahl von Testsoftware **21**
3.1 Softwarerecherche . . . 21
3.1.1 Karma . . . 21
3.1.2 Jasmine . . . 22
3.1.3 Protractor . . . 23
3.1.4 Mocha . . . 25
3.1.5 QUnit . . . 25
3.1.6 PhantomJS . . . 27
3.1.7 Chai . . . 27
3.1.8 Cucumber . . . 27
3.1.9 Unit JS . . . 27
3.1.10 Buster.js . . . 27
3.1.11 Sinon . . . 27
3.1.12 ngMock . . . 27
3.1.13 Intern . . . 27
3.1.14 Ava . . . 27
3.2 Anforderungsanalyse . . . 27
3.2.1 Anforderungen aufgrund Commerzbank-Richtlinien . . . 27
3.2.2 Weitere Anforderungen . . . 27
3.3 Auswahlentscheidung . . . 27

4 Evaluierung **28**
4.1 Projektumfeld . . . 28
4.2 Implementierung . . . 28
4.3 Auswertung . . . 28

5 Ausblick **29**
5.1 Einsatz von automatisierten Test im Commerzbank-Konzern 29
5.2 Mögliche Verbesserungen durch Einsatz von Angular2 29
5.3 Fazit 29

Literaturverzeichnis **30**

Stichwortverzeichnis **33**

Anhang **34**

Eidesstattliche Erklärung **34**

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1 Beispiel eines AngularJS-Templates, adaptiert nach [BT14] 15
2 Beispielhafter AngularJS-Controller, adaptiert nach [Goo17d] 15
3 Beispielhafter AngularJS-Controller mit Dependency Injection, adaptiert nach [Goo17a] 17
4 Beispiel einer package.json 19
5 Beispiel von Skripten in einer package.json (aus [Cir14]) 20
6 Beispiel der Watch-Funktionalität in einer package.json (aus [Cir14], angepasst durch den Autor) 20
7 Beispiel einer `karma.conf.js` (adaptiert nach [Kar17a; Kar17b]) 22
8 Beispiel einer Jasmine-Testsuite (adaptiert nach [Jas17b]) 23
9 Beispiel einer Protractor-Konfiguration (adaptiert nach [Pro17c]) 24
10 Beispiel einer Spec für Protractor (aus [Pro17c]) 25
11 Beispiel mehrerer Tests für QUnit (adaptiert nach [The17a]) 26

1 Motivation

Die Commerzbank AG ist Deutschlands zweitgrößtes Finanzinstitut[Sta14, S. 2]. Als solches stellen sich für ihre IT besondere Herausforderungen an Daten-, Ausfallsicherheit und Stabilität. Regulatorische Vorgaben durch die BaFin[Bun16], interne Programmierrichtlinien[Com17a; Com17b], das Book of Standards [Com16] und Weitere führen zu einem trägen und wenig innovativem IT-Umfeld. Somit ist es nicht überraschend, dass zur Entwicklung von Webanwendungen noch immer auf alte und etablierte Technologien wie JavaServer Pages und jQuery zurückgegriffen wird. Quelle!

Viele Unternehmen nutzen bereits seit einigen Jahren Angular als Technologiebasis für clientseitige Webanwendungen, beispielsweise *Gmail*, *PayPal* oder *Youtube*. In quellen der Commerzbank wurde Angular bisher nicht berücksichtigt; erst in jüngerer Vergangenheit wird es in vereinzelten Projekten eingesetzt. Hierbei werden automatisierte Entwicklertests in JavaScript aufgrund von Unwissenheit über die Möglichkeiten meist stiefmütterlich behandelt. Die Anwendungen werden stattdessen per Hand im Webbrowser getestet. Die sich hieraus ergebende Testabdeckung steht im Gegensatz zu den Anforderungen an Sicherheit und Stabilität im Bankenumfeld.

In dieser Bachelorthesis sollen daher die Möglichkeiten zur Realisierung von automatisierten Tests in AngularJS-Webanwendungen untersucht werden. ...?

2 Grundlagen

2.1 Test

Der Test von Software dient dazu, mögliche Fehler aufzudecken und dadurch die Qualität zu erhöhen. Der Nachweis von Fehlerfreiheit ist unmöglich, daher muss der Testaufwand verhältnismäßig zum Ergebnis sein[SL12, S. 14 ff.].

Beim Testen werden üblicherweise vier Teststufen unterschieden[SL12, S. 42 f.]:

- Komponententest
- Integrationstest
- Systemtest
- Abnahmetest

Bei Komponenten-, Integrations- sowie bedingt bei Systemstests handelt es sich um Entwicklertests, weshalb diese im Rahmen dieser Bachelorthesis relevant sind. Der Abnahmetest wird daher nicht betrachtet.

2.1.1 Komponententest

Ein Komponententest überprüft die einzelnen Bausteine der entwickelten Software erstmalig und unabhängig von anderen Bausteinen. Es wird überprüft, ob die Komponente den Anforderungen sowie dem definierten Softwaredesign entspricht. Außerdem kann auch der Quellcode analysiert und zur Erstellung der Testfälle herangezogen werden; dann handelt es sich um einen *Whitebox-Test.*[SL12, S. 44] In JavaScript ist die kleinste testbare Komponente üblicherweise eine Funktion[Zae12]. Die zu testende Komponente muss nicht zwingend atomar sein, d. h. die Funktion kann aus weiteren Funktionen zusammengesetzt sein, jedoch sollte nur die komponenteninterne Funktionsweise getestet werden[SL12, S. 45]. Beim Test von AngularJS-Anwendungen sind die Komponenten beispielsweise Controller, Services, Direktiven und Filter.

Ein Komponententest hat spezifische Testziele. Das Wichtigste ist die Sicherstellung, dass die Funktion der Anforderung in der Spezifikation entspricht. Hierdurch wird die Komponente wie gefordert mit anderen Komponenten zusammenarbeiten und somit in die Gesamtsoftware integriert werden können. Wichtig ist auch der Test auf Robustheit: Bei falschem Aufruf, also einem Verstoß gegen die Vorbedingungen, sollte die Komponente sinnvoll reagieren und den Fehler abfangen. Testfälle lassen sich in Positiv- und Negativtests unterteilen: Positivtests sind die Überprüfung von vorgesehenem Verhalten der Komponente, Negativtests der Test von nicht vorgesehenen, unzulässigen oder explizit ausgeschlossenen Sonderfällen.[SL12, S. 48].

Im Komponententest können auch nicht funktionale Qualitätseigenschaften getestet werden. Zu nennen sind hier beispielsweise Speicherverbrauch oder Antwortzeit, sowie

statische Tests auf Wartbarkeit, wie beispielsweise vorhandene Quelltextkommentare oder die Einhaltung von Programmierrichtlinien.[SL12, S. 49 f.].

evtl hier Ref auf unter-kapitel

2.1.2 Integrationstest

Der Integrationstest folgt nach den Komponententests und basiert auf getesteten Komponenten. Diese Komponenten werden zu größeren Komponenten oder Teilsystemen zusammengesetzt. Der Integrationstest dient dann dazu zu überprüfen ob alle Einzelteile korrekt zusammenarbeiten und soll Fehler in Schnittstellen und im Zusammenspiel aufdecken. [SL12, S. 52 f.] Beispielsweise kann in einem Integrationstest auch die Anbindung an externe Komponenten, wie Datenbanken oder REST-APIs überprüft werden. Diese werden im Komponententest durch *Mocks* ersetzt und emuliert.

Somit ist das aufdecken von Schnittstellenfehlern ein Testziel des Integrationstest, zum Beispiel wegen von der Spezifikation abweichender Schnittstellen. Außerdem ist ein Testziel unerwünschte Wechselwirkungen zwischen den Einzelkomponenten aufzudecken, welche das Zusammenspiel unmöglich machen. [SL12, S. 56]

Der Integrationstest ist jedoch kein Ersatz für den Komponententest, da er mit Nachteilen verbunden ist. Es ist schwer bis unmöglich, die tatsächliche Fehlerursache herauszufinden, da oft nicht klar ist in welcher Teilkomponente der Fehler aufgetreten ist, sondern dieser sich nur in einem abweichenden Gesamtverhalten äußert. Manche Fehler werden möglicherweise gar nicht gefunden, da regelmäßig kein vollumfänglicher Zugriff auf Einzelkomponenten besteht.[SL12, S. 57]

Es existieren verschiedene Integrationsstrategien, die Auswirkungen auf die Integrationstests haben:[SL12, S. 59 f.]

- Bei der Top-Down-Integration beginnt der Test mit der obersten Systemkomponente, von der alle Anderen aufgerufen werden. Sukzessive werden die weiteren Komponenten von oben nach unten integriert und getestet, wobei die untergeordneten Komponenten zunächst durch Platzhalter ersetzt werden. Vorteilhaft ist, dass keine aufwändigen Testtreiber zum Aufruf benötigt werden. Jedoch müssen Platzhalterkomponenten implementiert werden, was einen zusätzlichen Overhead beim Test bedeuten kann.

- Bei der Bottom-up-Integration werden zunächst die unteren, atomaren Komponenten integriert und getestet. Erst nach und nach werden größere Teilsysteme aus getesteten Komponenten integriert. Der Vorteil hierbei ist, dass keine Platzhalter implementiert werden müssen. Jedoch müssen hier aufwändige Testtreiber erstellt werden, welche die obergeordneten, aufrufenden Komponenten emulieren.

- Bei der Ad-hoc-Integration werden Komponenten integriert, sobald sie fertiggestellt sind. Nachteilig hierbei ist, dass sowohl Platzhalter als auch Testtreiber implementiert werden müssen. Allerdings bietet sich ein Zeitgewinn, da jede Komponente so früh wie möglich integriert wird.

- Bei der wenig empfehlenswerten Big-Bang-Integration werden alle Komponenten auf einmal integriert. Sie bietet ausschließlich Nachteile: Es wird Zeit verschwendet, da bis zur Fertigstellung der letzten Komponente gewartet wird. Auch treten alle Fehlerwirkungen gesammelt auf, so dass es schwierig ist, die Fehler zu finden.

2.1.3 Systemtest

Der Systemtest ist der finale Entwicklertest nach den abgeschlossenen Integrationstests. Getestet wird das gesamte System, möglichst in einer produktionsnahen Umgebung. Es sind keine Testtreiber oder Platzhalter mehr vorhanden, stattdessen wird überall die finale Hard- und Software genutzt. Das Testziel ist die Validierung, ob alle funktionalen und nicht-funktionalen Anforderungen erfüllt werden.[SL12, S. 60 ff.]

Aufgrund der erforderlichen Produktionsnähe sowie der erforderlichen Datenbasis kann der Systemtest nur bedingt als Entwicklertest gesehen werden[Roi05, S. 236; oos06]. Jedoch kann und sollte ein einfacher Systemtest auch von Entwicklern durchgeführt werden. Es bietet sich hier die Durchführung von End-To-End-Tests an, mittels derer das System von der Benutzeroberfläche bis zur untersten Komponentenschicht getestet werden kann[Sof10].

2.2 AngularJS

AngularJS ist ein von Google ins Leben gerufenes JavaScript-Framework zur Entwicklung von clientseitigen Webanwendungen[Goo17e]. Der Quellcode von AngularJS steht auf Github zur Verfügung und wird dort von einer großen Entwicklergemeinschaft weiterentwickelt[Ler13, S. 9]. Da es unter der MIT-Lizenz veröffentlicht ist eignet sich AngularJS auch für den kommerziellen Einsatz[Ler13, S. 9; Pat16].

Ende 2016 wurde eine neue Version von Angular veröffentlicht: Angular2[Pre16]. Durch die Bezeichnung kann AngularJS (Version 1) klar von Angular2 (Version 2) abgegrenzt werden. Im Rahmen dieser Bachelorarbeit wird AngularJS betrachtet.

2.2.1 Architektur

Bei der Entwicklung von Webanwendungen mit AngularJS kommt das Model-View-ViewModel-Entwurfsmuster (MVVM), eine Erweiterung von Model-View-Controller (MVC), zum Einsatz[BT14, S. 21].

Die Model-Schicht, also die Datenhaltung und Geschäftslogik, liegt hierbei auf dem Server und wird durch REST- oder WebSocket-Verbindungen dargestellt. Hierzu kommen in AngularJS meist Services zum Einsatz: Vordefinierte, wie z. B. der http-Service für HTTP-Abfragen, oder Selbsterstellte[Goo17c]. Mittels dieser kann Geschäftslogik auch clientseitig umgesetzt werden.[BT14, S. 21]

Es ist erforderlich, die über die Model-Schicht ermittelten Daten zu verwalten und gegebenenfalls zu transformieren um sie der Anzeige zur Verfügung zu stellen. Hierfür wird die ViewModel-Schicht genutzt. Außerdem wird in dieser Schicht die Funktionalität definiert, welche die View-Schicht steuert und diese zur Kommunikation mit der Model-Schicht nutzt. Dabei handelt es sich um Funktionen zur Behandlung von Events, wie Buttonclicks, Texteingaben, etc. Zur Weitergabe der Daten an die Anzeige wird Two-Way-Databinding (s. Abschnitt 2.2.2) verwendet. Umgesetzt wird die ViewModel-Schicht mit Controllern und sogenannten Scopes (s. Abschnitt 2.2.3). [BT14, S. 21 f.]

Die View-Schicht wird in AngularJS mit Templates und Direktiven umgesetzt. Templates sind HTML-Dateien, in welchen zusätzliche Tags und Attribute, die sogenannten Direktiven, verwendet werden. [BT14, S. 1 ff.]Direktiven ermöglichen es wiederverwendbare Komponenten zu erschaffen, indem Template und Quelltext in einem neuen Tag oder Attribut gekapselt werden[BT14, S. 49 f.].

```
<!DOCTYPE html>
<html ng-app="testApp">
<body ng-controller="TestCtrl">
  <input type="text" ng-model="someModelField" />
  <h1>Eingabe: {{someModelField}}</h1>

  <button ng-click="setName()">Andere Eingabe</button>

  <script src="js/angular.js" />
</body>
</html>
```

Listing 1: Beispiel eines AngularJS-Templates, adaptiert nach [BT14]

Im Beispieltemplate (s. Listing 1) wird ein Eingabefeld (HTML `input`) definiert, dessen Inhalt automatisch mit der im Scope liegenden Variable `someModelField` synchronisiert wird. Ein `h1`-Element zeigt den Inhalt dieser Variablen an. Für beide Synchronisierungen wird automatisch Two-Way-Databinding (s. Abschnitt) genutzt. Weiterhin wird ein Button definiert, welcher bei Click die Controller-Funktion `setName()` aufrufen soll. Die Angabe `ng-app` im `html`-Tag gibt das AngularJS-Modul an, welches von der Anwendung verwendet werden soll. Die Angabe `ng-controller` spezifiziert den von diesem Template zu verwendenden Controller (s. Listing 2).

ref

```
var testApp = angular.module("testApp", []);

testApp.controller("TestCtrl", function($scope) {
  $scope.someModelName = "Welt";

  $scope.setName = function() {
    $scope.someModelName = "Neue" + $scope.someModelName;
  }
});
```

Listing 2: Beispielhafter AngularJS-Controller, adaptiert nach [Goo17d]

In der JavaScript-Datei wird im verwendeten Modul eine Funktion, die als Controller mit dem Namen `TestCtrl` dient, definiert. Dieser Controller spezifiziert die Funktion `setName`, welche dadurch im Template verwendet werden kann. Das Skript muss über Dateikonkatenation (npm-Package „concat“[Gor17]) oder zusätzliches Einbinden in das Template an den Browser ausgeliefert werden.

2.2.2 Two-Way-Databinding

Two-Way-Databinding ist die Datenbindung in beide Richtungen. Es dient der Aktualisierung der Model-Daten anhand von Benutzereingaben in der Ansicht sowie

die Anpassung und Aktualisierung der View bei Änderungen des zugrundeliegenden Datenmodells. Dieses Konzept ist integraler Bestandteil von AngularJS und erspart das Schreiben von Boilerplate-Code, der nicht zur Geschäftslogik beiträgt. Ohne Two-Way-Databinding wäre es erforderlich, auf jedem zu synchronisierenden DOM-Element einen ChangeListener zu registrieren, welcher Änderungen durch den Benutzer an das Datenmodell weiterreicht. Außerdem müsste Logik implementiert werden, welche bei einer Änderung von Variablen im Datenmodell die View aktualisiert. Die Datenbindung in AngularJS erhöht somit die Effizienz, da Programmcode mit weniger Overhead geschrieben werden kann.[BT14, S. 24]

2.2.3 Scopes

Scopes sind in AngularJS die Basis der Datenbindung, wobei in einem Scope die Variablen und Funktionen definiert sind, welche für einen bestimmten Teil des DOM benötigt werden. Scopes sind hierarchisch angeordnen und bilden grob die DOM-Struktur nach. Den Ursprung dieser Hierarchie bildet der Root-Scope, welcher von AngularJS standardmäßig zur Verfügung gestellt wird. Hierbei können sie entweder die Eigenschaften des jeweils übergeordneten Scopes erben oder isoliert sein. Beim Auswerten von Ausdrücken in Templates (z. B. `{{scopeVariable}}`) wird zuerst im mit dem jeweiligen Element assoziierten Scope und danach in den jeweils Übergeordneten nach der Eigenschaft gesucht.[BT14, S. 23 ff.; Goo17b]

Zur Erkennung, ob eine Variable im Datenmodell geändert wurde und eine Aktualisierung der Anzeige erforderlich ist, wird in AngularJS Dirty Checking genutzt. Hierzu wird von jedem Scope eine Kopie im Speicher gehalten, so dass bei jedem Event die gehaltene und aktuelle Version eines Scopes miteinander verglichen werden können. Bei veränderten Werten wird eine Aktualisierung der Anzeige angestoßen.[BT14, S. 24; Sym]

2.2.4 Dependency Injection

Dependency Injection ist ein Entwurfsmuster welches beschreibt, wie eine Komponente Zugriff auf benötigte Abhängigkeiten, also andere Komponenten, bekommt und wird in AngularJS durchgängig genutzt. Bei Nutzung von Dependency Injection werden die Komponenten nicht selber erzeugt sondern von außerhalb durch einen Injector geliefert. Hierfür ist es nötig, dass Services, Direktiven, Filter und Controller mit den entsprechenden Factory-Funktionen von AngularJS erzeugt werden. Diese registrieren einerseits die Komponente und ermöglichen es, diese in andere Komponenten zu injizieren, kümmern sich aber auch um die Bereitstellung der benötigten Komponenten. Ein beispielhafter Controller mit injizierten Abhängigkeiten findet sich in Listing 3.[Goo17a]

```
someModule.controller("MyController", ["$scope", "$http", "dep", function
    ($scope, $http, dep) {
  $scope.aMethod = function() {
    dep.someFunction();
    //...
  }
});
```

Listing 3: Beispielhafter AngularJS-Controller mit Dependency Injection, adaptiert nach [Goo17a]

Dependency Injection bietet gravierende Vorteile für die Testbarkeit. Es ermöglicht, eine Komponente durch ein spezielles selbst implementiertes Mock-Objekt zu ersetzen, dessen Verhalten festgelegt werden kann. Bei Tests kann das Verhalten der Abhängigkeiten festgelegt und Komponenten isoliert getestet werden.[BT14, S. 27]

2.3 Node.js

2.3.1 Laufzeitumgebung

Node.js ist eine Laufzeitumgebung, mit der JavaScript ohne Webbrowser ausgeführt werden kann[HWD12, S. 1]. Somit ist es möglich, JavaScript nicht nur für die Darstellung von Benutzeroberflächen im Webbrowser zu Nutzen, sondern auch als Backend-Sprache oder zur Unterstützung von Entwicklungsprozessen auf Continuous-Integration-Servern oder Entwicklerarbeitsplätzen.

Intern nutzt Node.js die JavaScript-Engine *Chrome V8* [Nod17], welche von Google als Open-Source-Software veröffentlicht wurde. V8 kommt auch im weitverbreiteten Webbrowser *Google Chrome* zum Einsatz und implementiert den JavaScript-Standard ECMAScript wie in ECMA-262 spezifiziert[Goo17f]. ECMA-262 ist der im Juni 2016 veröffentlichte und zurzeit aktuellste JavaScript-Standard[Ecm16]. Somit bietet Node.js alle spezifizierten und von Google Chrome unterstützten Sprachfunktionalitäten. Es eignet sich daher auch für den Test von für Webbrowser entwickelte Webanwendungen.

2.3.2 npm

Der Node Package Manager (npm) ist der zusammen mit Node.js installierte Paketmanager für JavaScript. Unter npm wird außerdem die *npm Registry*, also die zentrale Ablage von mit npm verwendeten JavaScript-Paketen, verstanden, auf welche der Node Package Manager zugreift. [npm17c] Die npm Registry enthält über 180.000 Pakete und ist damit das größte öffentliche Softwarerepository[DeB17].

Grundlegend funktioniert die Paketverwaltung mit einer JSON-Konfigurationsdatei, der `package.json` (vgl. Listing 4). Die Datei enthält den Namen sowie die Version des Pakets, für welches sie angelegt wurde, sowie optional weitere Metadaten wie Beschreibung, Autor und Referenzlinks auf Bugtracker. Außerdem werden hier Abhängigkeiten angegeben, die zur Ausführung (`dependencies`) oder zur Entwicklung (`devDependencies`) in diesem Paket benötigt werden.[npm17d] Die angegebenen Abhängigkeiten werden von npm automatisiert heruntergeladen und im Ordner `node_modules` abgelegt, von wo aus sie in die JavaScript-Anwendung eingebunden werden können. Auch transitive Abhängigkeiten werden von npm aufgelöst.[npm17a]

```
{
  "name": "test_package",
  "version": "1.0.0",
  "dependencies:" {
    "my_dependency": "1.1.0"
  },
  "devDependencies": {
    "some_test_framework": "0.1.0"
  }
}
```

Listing 4: Beispiel einer package.json

Neben der Paketverwaltung kann npm auch zum Build als Taskrunner eingesetzt werden. Hiermit kann der Buildprozess eines Paketes automatisiert werden, z. B. durch die automatisierte Ausführung von Tests oder dem Aufrufen von Compilern. Hierzu werden in der `package.json` Skripte angegeben. Diese bestehen aus einem Skript-Namen und dem auszuführenden Befehl. Im Beispiel (siehe Listing 5) werden drei Skripte definiert:[Cir14]

- „lint“ führt das Kommando `jshint **.js` aus. Dies dient dem Überprüfen von JavaScript-Dateien auf statische Programmierfehler[Wal+17].
- „build“ führt das Kommando `browserify [...]` aus. Dieses dient dem Zusammenfügen von mittels `require` eingebundenen JavaScript-Dateien in eine konkatenierte Datei[Hal+17].
- „test“ führt das Kommando `mocha [...]` aus. Mocha ist ein Test-Runner (siehe auch Abschnitt 3.1.4).

Angegebene Skripte können auch automatisch in sogenannten Hooks (*Pre* und *Post Hooks*) ausgeführt werden. Im Beispiel (siehe Listing 5) sind folgende Hooks definiert:[Cir14]

- „prepublish“ wird vor der Ausführung von `publish`, welches ein npm Standard-Skript ist und das Paket in der npm Registry veröffentlicht[npm16], das benutzerdefinierte Skript `build` sowie dadurch `prebuild` und `postbuild` ausführen.
- „prebuild“ wird vor Ausführung des `build`-Skripts das Paket durch Ausführung von `test` überprüfen.
- „pretest“ wird vor Ausführung von `test` mittels `lint` das Paket auf statische Fehler untersuchen.

```
"scripts": {
  "lint": "jshint **.js",
  "build": "browserify index.js > myproject.min.js",
  "test": "mocha test/",

  "prepublish": "npm run build # also runs npm run prebuild",
  "prebuild": "npm run test # also runs npm run pretest",
  "pretest": "npm run lint"
}
```

Listing 5: Beispiel von Skripten in einer package.json (aus [Cir14])

Die wohl populärste Funktion von Taskrunnern ist das automatisierte Beobachten des Dateisystems auf Änderungen. Häufig ist es wünschenswert, dass bei einer Dateiänderung automatisch ein entsprechender Buildprozess oder die Tests ausgeführt werden. Diese Funktionalität bietet npm im Gegensatz zu anderen Taskrunnern wie *Gulp* oder *Grunt* nicht nativ, sondern nur mithilfe eines Zusatzpakets. Ein entsprechendes Beispiel, welches bei Veränderung einer Datei im Paket-Ordner die JavaScript-Module zu einer Datei konkateniert[Hal+17] und den JavaScript-Code des Paketes überprüft, findet sich in Listing 6.[Cir14]

```
"scripts": {
  "lint": "jshint **.js",
  "build:js": "browserify assets/scripts/main.js > dist/main.js",
  "build": "npm run build:js",
  "build:watch": "watch 'npm run build && npm run lint' .",
}
```

Listing 6: Beispiel der Watch-Funktionalität in einer package.json (aus [Cir14], angepasst durch den Autor)

3 Auswahl von Testsoftware

3.1 Softwarerecherche

Im folgenden Kapitel werden verschiedene Tools und Frameworks zur Realisierung von automatisierten Tests untersucht. Einen Überblick über zur Verfügung stehende Tools bieten dabei Jon [Jon17] und Weitere.

3.1.1 Karma

Karma ist ein Test-Runner für die Ausführung von JavaScript-Tests. Er wurde vom AngularJS-Team ins Leben gerufen und wird auf GitHub von einer Open-Source-Gemeinschaft weiterentwickelt.[Kar17g] Karma liegt als Paket `karma` im npm-Repository[Kar17c].

Karma ermöglicht die Nutzung diverser Testframeworks, wie Jasmine (s. Abschnitt 3.1.2), Mocha (s. Abschnitt 3.1.4) oder QUnit (s. Abschnitt 3.1.5). Auch Continuous Integration Server wie Jenkins oder Travis werden unterstützt.[Kar17c]

Grundlegend basiert Karma auf einem Client-Server-Prinzip, wobei Karma einen Webserver startet, welcher alle verbundenen Browser fernsteuert und in diesen die Tests ausführt. Ein Browser kann hierbei entweder manuell, also durch Aufruf der vom Karma-Server bereitgestellten URL, oder automatisiert, also indem der Browser durch Karma gestartet wird (vgl. Listing 7), verbunden werden. In jeder Testumgebung wird der Quelltext mittels IFrame eingebunden, der Test ausgeführt und danach die Ergebnisse an den Server gesendet. Dort werden die Ergebnisse aufgearbeitet präsentiert oder automatisiert von übergeordneten Buildprozessen verarbeitet. Das verwendete Prinzip stammt aus einer Masterthesis [Jín13]; tieferes Verständnis ist jedoch für die reine Nutzung von Karma nicht erforderlich.[Kar17d]

Für die Konfiguration wird eine JavaScript-Datei, die `karma.conf.js`, genutzt. Ein Beispiel findet sich in Listing 7. Mit dieser beispielhaften Konfiguration wird für die Testausführung das Framework Jasmine (s. Abschnitt 3.1.2) genutzt. Der Parameter `files` gibt an, welche Dateien von Karma ausgeliefert und beobachtet werden, und somit bei Änderung welcher Dateien die Tests automatisch erneut ausgeführt werden. Außerdem ist konfiguriert, dass bei Testdurchführung automatisch Firefox gestartet werden und in diesem die Tests ausgeführt werden soll.[Kar17a; Kar17b]

```
module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    files: [
      'build/js/**/*.js',
      'build/js/**/*.test.js'
    ],
    browsers: ['Firefox']
  });
};
```

Listing 7: Beispiel einer `karma.conf.js` (adaptiert nach [Kar17a; Kar17b])

Die Funktionalität von Karma lässt sich mit Plugins erweitern. Sie werden für die Einbindung von Testframeworks, ein verändertes Ausgabeformat der Testergebnisse, Präprozessoren (z. B. für die Auslieferung von in JavaScript eingebettetem HTML oder die Ermittlung der Code Coverage)[Kar17f] oder die Einbindung von Browsern wie Firefox, Chrome oder PhantomJS (s. Abschnitt 3.1.6) benötigt. Jedes Plugin ist ein npm-Paket, daher werden Plugins über npm installiert. Karma bindet alle installierten Pakete mit dem Namen `karma-*` automatisch ein.[Kar17e] Im npm-Repository liegen über 1300 Karma-Plugins.[npm17b]

3.1.2 Jasmine

Jasmine ist ein Behavior Driven Development Framework zum Test von JavaScript[Jas17b]. Es liegt unter `jasmine` im npm-Repository[Jas17a]. Jasmine bietet eine saubere und einfache Syntax zur Beschreibung von Testfällen. Die Tests bestehen aus drei Ebenen: Testsuites, Spezifizierungen („Specs“) und Erwartungen, also den eigentlichen Testassertions[Jas17b]. Ein beispielhafter Test findet sich in Listing 8.

Eine Testsuite beginnt auf oberster Ebene mit dem Aufruf der globalen JavaScript-Funktion `describe(string, function)`. Der String ist hierbei der Name der Testsuite, üblicherweise wird hier das Testsubjekt benannt. Die Funktion implementiert die Testsuite und besteht aus Specs.[Jas17b]

Ein Spec wird durch Aufruf der globalen Funktion `it(string, function)` angelegt. Der String enthält eine Beschreibung des Testfalls; nach dem BDD-Modell also eine Beschreibung des erwarteten Verhaltens. Die Funktion dient zum Überprüfen dieses Verhaltens und enthält Assertions, welche entweder `true` oder `false` ergeben. Liefern alle Assertions `true` so gilt die Spec als bestanden, ansonsten als durchgefallen.[Jas17b]

Eine Assertion besteht in Jasmine aus der Funktion `expect(object)`, welcher der tatsächliche Wert übergeben wird. Diese wird mit einer Matcher-Funktion verkettet, welche den erwarteten Wert übergeben bekommt und die beiden Werte vergleicht und

auswertet. Es wird eine Vielzahl an vorgefertigten Matchern mitgeliefert: `toEqual`, `toContain`, `toBeTruthy`, und Weitere[Jas17b; Ban12].

```
describe("Sample Test", function() { //Suite
  it("should add integers correctly", function() { //Spec #1
    var result = 13 + 2;

    expect(result).toBe(15); //Expectation
  });

  it("should compare e and pi correctly", function() { //Spec #2
    var e = 2.78;
    var pi = 3.1416;

    expect(e).toBeLessThan(pi); //Expectation #1
    expect(pi).not.toBeLessThan(e); //Expectation #2
  })
})
```

Listing 8: Beispiel einer Jasmine-Testsuite (adaptiert nach [Jas17b])

Specs können als *pending* deklariert werden. Sie werden dann nicht ausgeführt, aber im Ergebnis angezeigt. Hierfür kann beim Aufruf der `it`-Funktion die Übergabe einer Funktion weggelassen werden, stattdessen die `xit`-Funktion aufgerufen werden oder im Funktionskörper die `pending`-Funktion genutzt werden.[Jas17b]

3.1.3 Protractor

Protractor ist ein speziell für Angular-Anwendungen entwickeltes Framework für End-to-End-Tests. Die Tests werden in Browsern direkt gegen die Anwendungsoberfläche durchgeführt und simulieren somit das Verhalten eines echten Benutzers. Es liegt im npm-Repository mit der ID `protractor` und ist dadurch einfach zu installieren.[Pro17a]

Für die Steuerung des Browsers greift Protractor auf Selenium zurück[Pro17a], welches den W3C WebDriver-Standard implementiert und als Proxyserver zwischen Protractor und dem Browser agiert[Tol+16]. Selenium unterstützt alle großen Webbrowser: aktuell die aktuellsten Versionen von Firefox, Internet Explorer ab Version 7, Safari ab Version 5.1, Opera und Chrome[Sel17]. Vom Einsatz von PhantomJS (s. Abschnitt 3.1.6) zusammen mit Protractor wird ausdrücklich abgeraten, da es hier Berichten zufolge häufig zu Abstürzen und abweichendem Verhalten kommt[Pro17b]. Laut eigener Aussage wird Selenium automatisch zusammen mit Protractor installiert und ist nach Aufruf von `webdriver-manager update` und `webdriver-manager start` ohne weitere Konfiguration lauffähig[Pro17a].

Protractor nutzt als Framework für die Testbeschreibung standardmäßig Jasmine (s. Abschnitt 3.1.2), unterstützt out-of-the-box aber auch Mocha (s. Abschnitt 3.1.4). Die

nachfolgenden Beispiele nutzen daher auch Jasmine. Das eingesetzte Testframework, die Adresse unter welcher der Selenium-Server angesprochen wird, Testdateien, Timeouts, für den Test zu verwendende Browser und weitere Feineinstellungen werden in einer Konfigurationsdatei (s. Listing 9) konfiguriert.

```
exports.config = {
  framework: 'jasmine',
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['js/e2e/**/*.js'],
  multiCapabilities: [
    {browserName: 'firefox'},
    {browserName: 'chrome'}
  ]
}
```

Listing 9: Beispiel einer Protractor-Konfiguration (adaptiert nach [Pro17c])

Üblicherweise hat jede zu testende Seite eine eigene Testsuite und jeder Testfall ist eine eigene Spec (s. Listing 10). Vor der eigentlichen Testdurchführung muss die jeweilige Seite aufgerufen werden: hierzu dient die durch Protractor bereitgestellte Funktion `browser.get(url)`. Es bietet sich an, diese in `beforeEach()` auszuführen, einer Funktion die durch Jasmine vor jedem Spec aufgerufen wird. Auf Elemente kann mit der Funktion `element` zugegriffen werden, welcher ein Locator übergeben wird. Locator sind ein durch Protractor definiertes Konstrukt und beschreiben, wie das Element gefunden werden kann. Um mit den gefundenen Elementen zu interagieren werden verschiedene Funktionen bereitgestellt: beispielsweise `sendKeys` zur Zeicheneingabe, `click` zum Simulieren eines Mausklicks oder `getText` um den Elementinhalt zu ermitteln.

```
describe('Protractor Demo App', function() {
  beforeEach(function() {
      browser.get('http://juliemr.github.io/protractor-demo/');
  });

    it('should have a title', function() {
      expect(browser.getTitle()).toEqual('Super Calculator');
    });

    it('should add one and two', function() {
      element(by.model('first')).sendKeys(1);
      element(by.model('second')).sendKeys(2);

      element(by.id('gobutton')).click();

      expect(element(by.binding('latest')).getText()).toEqual('3');
    });
});
```

Listing 10: Beispiel einer Spec für Protractor (aus [Pro17c])

3.1.4 Mocha

Mocha ist ein Testframework und Test-Runner, welches sowohl in Node.js als auch in Browsern lauffähig ist. Es liegt als Paket `mocha` im npm-Repository und kann darüber installiert werden.[Moc17]

Tests bestehen in Mocha aus drei Ebenen. Die Oberste sind die Testsuites, welche weitere Testsuites oder Testfälle enthalten können. Testfälle bestehen aus funktionalem Code sowie Assertions als eigentliche Testüberprüfung. Es werden verschiedene Stile zur Testbeschreibung unterstützt: BDD, TDD, QUnit und weitere, welche sich nur in ihrem Aussehen unterscheiden und Entwicklern ermöglichen, ihren eigenen Stil zur Definition von Tests zu wählen.[Moc17]

Für Assertions können in Mocha verschiedene Frameworks genutzt werden. In [Moc17] wird beispielsweise die Nutzung von *should.js* bei Verwendung des BDD-Stils, *expect.js* oder *chai* (s. Abschnitt **??**) empfohlen. Es ist einem Entwickler somit möglich, Mocha auf die eigenen Vorlieben anzupassen.[Moc17]

Auf eine genauere Vorstellung und Codebeispiele wird an dieser Stelle aufgrund der Vielseitigkeit verzichtet.

Konfigu
synchro
asynchr
Repor-
ter,
Src-Bsp

3.1.5 QUnit

QUnit ist ein Framework für automatisierte Komponententests mit JavaScript und wird von jQuery und einer Vielzahl weiterer Projekten genutzt. Es liegt unter `qunitjs` als

Paket im npm-Repository. Es kann sowohl in Browsern als auch in Node.js ausgeführt werden.[The17b]

In QUnit geschriebene Tests ähneln denen vieler Testframeworks populärer anderer Sprachen, wie beispielsweise JUnit in Java. Ein Testfall wird mittels Aufruf von `QUnit.test(string, function)` definiert. In der übergebenen Funktion kann Testcode aufgerufen werden und das Ergebnis mit Assertions validiert werden. Wenn mindestens eine Assertion fehlschlägt, gilt der Test als fehlgeschlagen; sonst als bestanden. QUnit liefert Assertions mit: beispielsweise `assert.ok`, welche einen truthy Wert erwartet, oder `assert.equal`, welches zwei als gleich angesehene Werte erwartet.[The17a]

Tests können in durch Aufruf von `QUnit.module(string)` erzeugten Modulen gruppiert werden (s. Listing 11). In Modulen kann Code ausgelagert werden, indem die vor und nach jedem Test aufgerufenen Funktionen `beforeEach` und `afterEach` definiert werden.[The17a]

```
QUnit.module("group a");
QUnit.test("a basic test example", function(assert) {
  assert.ok(true, "this test is fine");
});
QUnit.test("a basic test example 2", function(assert) {
  var sum = 1 + 2;
  assert.equal(sum, 3, "sum equals 3");
});
```

Listing 11: Beispiel mehrerer Tests für QUnit (adaptiert nach [The17a])

3.1.6 PhantomJS

3.1.7 Chai

3.1.8 Cucumber

3.1.9 Unit JS

3.1.10 Buster.js

3.1.11 Sinon

3.1.12 ngMock

3.1.13 Intern

3.1.14 Ava

3.2 Anforderungsanalyse

3.2.1 Anforderungen aufgrund Commerzbank-Richtlinien

3.2.2 Weitere Anforderungen

3.3 Auswahlentscheidung

4 Evaluierung

4.1 Projektumfeld

4.2 Implementierung

4.3 Auswertung

5 Ausblick

5.1 Einsatz von automatisierten Test im Commerzbank-Konzern

5.2 Mögliche Verbesserungen durch Einsatz von Angular2

5.3 Fazit

Literaturverzeichnis

[Ban12] Łukasz Kazimierz Bandzarewicz. *Jasmine Cheat Sheet*. 8. März 2012. URL: http://blog.bandzarewicz.com/blog/2012/03/08/jasmine-cheat-sheet/ (besucht am 06. 05. 2017).

[BT14] Robin Böhm und Philipp Tarasiewicz. *AngularJS: Eine praktische Einführung in das JavaScript-Framework*. 27. Mai 2014.

[Bun16] Bundesanstalt für Finanzdienstleistungsaufsicht. *BaFin - Banken & Finanzdienstleister*. 22. März 2016. URL: https://www.bafin.de/DE/Aufsicht/BankenFinanzdienstleister/bankenfinanzdienstleister_node.html (besucht am 02. 04. 2017).

[Cir14] Keith Cirkel. *How to Use npm as a Build Tool*. 9. Dez. 2014. URL: https://www.keithcirkel.co.uk/how-to-use-npm-as-a-build-tool/ (besucht am 10. 04. 2017).

[Com16] Commerzbank AG. »Book of IT-Standards«. internes Dokument. 15. Juni 2016.

[Com17a] Commerzbank AG. »IT-Richtlinie: Allgemeine Programmierrichtlinien«. Version 4. internes Dokument. 1. Apr. 2017.

[Com17b] Commerzbank AG. »IT-Richtlinie: Programmierrichtlinien JavaScript«. internes Dokument. 8. Feb. 2017.

[DeB17] Erik DeBill. *Modulecounts*. 9. Apr. 2017. URL: http://www.modulecounts.com (besucht am 09. 04. 2017).

[Ecm16] Ecma International. *Standard ECMA-262. ECMAScript 2016 Language Specification*. Version 7. Juni 2016. URL: https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (besucht am 06. 04. 2017).

[Goo17a] Google. *AngularJS: Developer Guide: Dependency Injection*. 2017. URL: https://docs.angularjs.org/guide/di (besucht am 21. 04. 2017).

[Goo17b] Google. *AngularJS: Developer Guide: Scopes*. 2017. URL: https://docs.angularjs.org/guide/scope (besucht am 21. 04. 2017).

[Goo17c] Google. *AngularJS: Developer Guide: Services*. 2017. URL: https://docs.angularjs.org/guide/services (besucht am 20. 04. 2017).

[Goo17d] Google. *AngularJS: Developer Guidee: Controllers*. 2017. URL: https://docs.angularjs.org/guide/controller (besucht am 24. 04. 2017).

[Goo17e] Google. *AngularJS: Miscellaneous: FAQ*. 2017. URL: https://docs.angularjs.org/misc/faq.

[Goo17f] Google Developers. *Chrome V8*. 2017. URL: https://developers.google.com/v8/ (besucht am 06. 04. 2017).

[Gor17] Konstantin Gorodinskii. *concat*. 2017. URL: https://www.npmjs.com/package/concat (besucht am 21. 04. 2017).

[Hal+17] James Halliday u. a. *browserify*. 5. Apr. 2017. URL: https://github.com/substack/node-browserify/blob/master/readme.markdown (besucht am 10. 04. 2017).

[HWD12] T Hughes-Croucher, M Wilson und T Demmig. *Einführung in Node.js*. O'Reilly, 2012. ISBN: 9783868997972.

[Jas17a] Jasmine. *Getting Started*. 2017. URL: https://jasmine.github.io/pages/getting_started.html (besucht am 04. 05. 2017).

[Jas17b] Jasmine. *introdution.js*. 2017. URL: https://jasmine.github.io/edge/introduction.html (besucht am 04. 05. 2017).

[Jín13] Vojtěch Jína. »JavaScript Test Runner«. Magisterarb. Czech Technical University in Prague Faculty of Electrical Engineering Department of Computer Science und Engineering, 30. Juni 2013. URL: https://github.com/karma-runner/karma/raw/master/thesis.pdf (besucht am 03. 05. 2017).

[Jon17] Jon. *JavaScript unit test tools for TDD*. 16. Jan. 2017. URL: http://stackoverflow.com/a/680713.

[Kar17a] Karma. *Configuration File*. 2017. URL: http://karma-runner.github.io/1.0/config/configuration-file.html (besucht am 03. 05. 2017).

[Kar17b] Karma. *Files*. 2017. URL: http://karma-runner.github.io/1.0/config/files.html (besucht am 03. 05. 2017).

[Kar17c] Karma. *Frequently Asked Questions*. 2017. URL: http://karma-runner.github.io/1.0/intro/faq.html (besucht am 03. 05. 2017).

[Kar17d] Karma. *How It Works*. 2017. URL: http://karma-runner.github.io/1.0/intro/how-it-works.html (besucht am 03. 05. 2017).

[Kar17e] Karma. *Plugins*. 2017. URL: http://karma-runner.github.io/1.0/config/plugins.html (besucht am 04. 05. 2017).

[Kar17f] Karma. *Proprocessors*. 2017. URL: http://karma-runner.github.io/1.0/config/preprocessors.html (besucht am 04. 05. 2017).

[Kar17g] Karma. *Spectacular Test Runner For Javascript*. 2017. URL: http://karma-runner.github.io/1.0/index.html (besucht am 03. 05. 2017).

[Ler13] Ari Lerner. *ng-book. The Complete Book on AngularJS*. 2013. ISBN: 978-0-9913446-0-4.

[Moc17] Mocha. *Mocha - the fun, simple, flexible JavaScript test framework*. 2017. URL: https://mochajs.org/.

[Nod17] Node.js Foundation. *Node.js*. 2017. URL: https://nodejs.org/en/ (besucht am 06. 04. 2017).

[npm16] npm, Inc. *npm-publish. Publish a package*. Nov. 2016. URL: https://docs.npmjs.com/cli/publish (besucht am 10. 04. 2017).

[npm17a] npm. *install - npm Documentation*. 2017. URL: https://docs.npmjs.com/cli/install (besucht am 30. 04. 2017).

[npm17b] npm. *karma-* - npm search*. 4. Mai 2017. URL: https://www.npmjs.com/search?q=karma-*&page=1&ranking=optimal (besucht am 04. 05. 2017).

[npm17c] npm, Inc. *npm*. 2017. URL: https://www.npmjs.com/about (besucht am 08. 04. 2017).

[npm17d] npm, Inc. *Using a package.json*. 9. März 2017. URL: https://docs.npmjs.com/getting-started/using-a-package.json (besucht am 09. 04. 2017).

[oos06] oose Innovative Informatik eG. *A-256 Testkonzept erstellen*. 6. Nov. 2006. URL: https://www.oose.de/oep/desc/a_824d.htm?tid=256 (besucht am 17. 04. 2017).

[Pat16] Bill Patrianakos. *Why I Use the MIT License*. 28. Juli 2016. URL: http://billpatrianakos.me/blog/2016/07/28/why-i-use-the-mit-license/ (besucht am 19. 04. 2017).

[Pre16] Pascal Precht. *Angular 2 Is Out - Get Started Here*. 18. Dez. 2016. URL: https://blog.thoughtram.io/angular/2016/09/15/angular-2-final-is-out.html (besucht am 19. 04. 2017).

[Pro17a] Protractor. *Protractor - end-to-end testing for AngularJS*. 2017. URL: http://www.protractortest.org (besucht am 06. 05. 2017).

[Pro17b] Protractor. *Setting Up the Browser*. 2017. URL: http://www.protractortest.org/#/browser-setup (besucht am 07. 05. 2017).

[Pro17c] Protractor. *Tutorial*. 2017. URL: http://www.protractortest.org/#/tutorial (besucht am 07. 05. 2017).

[Roi05] Erich H. Peter Roitzsch. *Analytische Softwarequalitätssicherung in Theorie und Praxis*. MV-Verlag, 2005. ISBN: 9783865822024.

[Sel17] SeleniumHQ. *Platforms Supported by Selenium*. 2017. URL: http://www.seleniumhq.org/about/platforms.jsp (besucht am 07. 05. 2017).

[SL12] Andreas Spillner und Tilo Linz. *Basiswissen Softwaretest*. Dpunkt.Verlag GmbH, 11. Sep. 2012. ISBN: 3864900247. URL: http://www.ebook.de/de/product/19361935/andreas_spillner_tilo_linz_basiswissen_softwaretest.html.

[Sof10] Software-Sanierung. *Warum End-To-End-Tests alleine mehr schaden als nützen*. 28. Feb. 2010. URL: https://softwaresanierung.wordpress.com/2010/02/28/warum-end-to-end-tests-alleine-mehr-schaden-als-nutzen/ (besucht am 17. 04. 2017).

[Sta14] Ilias Stampoulis. »Börse Frankfurt: Commerzbank schießt in die Höhe«. In: *Handelsblatt* (6. Juni 2014).

[Sym] Symetics GmbH. *AngularJS.DE -> Dirty-Checking / Updatezyklus*. URL: https://angularjs.de/buecher/angularjs-buch/dirty-checking (besucht am 21. 04. 2017).

[The17a] The jQuery Foundation. *Cookbock | QUnit*. 2017. URL: http://qunitjs.com/cookbook/ (besucht am 10. 05. 2017).

[The17b] The jQuery Foundation. *QUnit*. 2017. URL: https://qunitjs.com/ (besucht am 10. 05. 2017).

[Tol+16] Andreas Tolfsen u. a. *Selenium*. 21. Dez. 2016. URL: https://github.com/SeleniumHQ/selenium (besucht am 07. 05. 2017).

[Wal+17] Rick Waldron u. a. *About JSHint*. 28. Jan. 2017. URL: http://jshint.com/about/ (besucht am 10. 04. 2017).

[Zae12] Jörn Zaefferer. *Introduction To JavaScript Unit Testing. How To Build A Testing Framework*. 27. Juni 2012. URL: https://www.smashingmagazine.com/2012/06/introduction-to-javascript-unit-testing/ (besucht am 15. 04. 2017).

Abkürzungsverzeichnis

BaFin Bundesanstalt für Finanzdienstleistungsaufsicht
DOM Document Object Model
HTTP Hypertext Transfer Protocol
jQuery jQuery ist eine JavaScript-Bibliothek für DOM-Manipulation, Event-Handling und Animation.
JSON JavaScript Object Notation
Mock Ein Platzhalter-Objekt, welches als Attrappe verwendet wird.
MVC Model-View-Controller, ein Entwurfsmuster
MVVM Model-View-ViewModel, ein Entwurfsmuster
REST Representational State Transfer
WebSocket WebSocket ist eine bidirektionale Erweiterung von HTTP.
Whitebox-Test Ein Whitebox-Test ist ein Test, welcher unter Kenntniss der inneren Funktionsweise entwickelt wird.

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Bachelorarbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift$: $Ort, Datum$: