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:	18. Juni 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.

Zusammenfassung

In dieser Bachelorthesis werden die Möglichkeiten zur Realisierung automatisierter Tests von AngularJS-Anwendungen untersucht. Hierbei werden speziell die Erfordernisse des Commerzbank-Konzern beachtet.

Zunächst wird eine Einführung in die allgemeinen Themen Testen, Angular und Node.js gegeben und Anforderungen an Testframework und -tools definiert. Anschließend werden die verschiedene Programme untersucht und mit Blick auf die Erfüllung der aufgestellten Anforderungen bewertet. Eine Kombination von Karma, Mocha, Chai und Protractor, sowie Sinon, ngMock und Istanbul, wird für den Einsatz im Commerzbank-Konzern als am geeignetsten betrachtet und ausgewählt.

Eine prototypische Realisierung von Tests in einem exemplarischen, realen Projekt mit der Auswahl zeigt, dass die Realisierung von Komponententests mit Karma, Mocha und Chai problemlos funktioniert und einfach anzuwenden ist. Es zeigt sich jedoch auch, dass eine Durchführung von automatisierten End-To-End-Tests an Commerzbank-Arbeitsplätzen aufgrund verschiedener regulatorischer Vorgaben und Einschränkungen nicht möglich ist und die ausgewählte Software Protractor für diesen Zweck nicht funktioniert.

Abschließend wird ein Ausblick auf mögliche Verbesserungen bei automatisierten Tests durch den Einsatz neuerer Versionen von Angular vorgenommen und ein positives Fazit gezogen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

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 [Com16b] 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 jQuery und JavaServer Pages, welche von Oracle als veraltet erklärt wurden [Ora13], zurückgegriffen wird [Com16a].

Viele Unternehmen nutzen bereits seit einigen Jahren Angular als Technologiebasis für den clientseitigen Teil von Webanwendungen, beispielsweise *Gmail*, *PayPal* oder *Youtube* [Hom16]. In 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 Systemtests 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.].

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 der Überprüfung ob alle Einzelteile korrekt zusammenarbeiten oder Fehler in Schnittstellen existieren. [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 neben vielen Vorteilen auch Nachteile mit sich bringt. So ist es beispielsweise 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 entsteht dadurch 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 Benutzungsoberfläche 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 [Goo17l]. 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 [Goo17k]. 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 2.2.2) 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).

```
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 [Goo17g]

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 angeordnet 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.; Goo17j]

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 andererseits, diese in andere Komponenten zu injizieren. Des Weiteren kümmern sie sich aber auch um die Bereitstellung der benötigten Komponenten. [Goo17h]

Im Beispiel in Listing 3 wird im Modul `someModule` ein neuer Controller `MyController` angelegt, in welchen der Scope, sowie die zwei Services `http`, welcher

```
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 [Goo17h]

standardmäßig von AngularJS zur Verfügung gestellt wird, sowie `dep`, welcher benutzerdefiniert erstellt wurde, injiziert werden. Diese Abhängigkeiten werden durch Übergabe als Funktionsparameter bereitgestellt.

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 serverseitig, also 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 [Goo17n]. 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 Anwendungen.

2.3.2. Node Package Manager (npm)

Der Node Package Manager (npm) ist der zusammen mit Node.js installierte Paketmanager sowie Buildmanager 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 3.2.2).

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] und das benutzerdefinierte Skript `build` ausführt. Durch die Ausführung von `build` werden automatisch auch `prebuild` und `postbuild` ausgeführt.
- „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

Im folgenden Kapitel wird eine Auswahl verschiedener Tools und Frameworks zur Realisierung von automatisierten Tests anhand von vorher zu definierenden Anforderungen getroffen.

3.1. Anforderungsanalyse

An die auszuwählenden Frameworks oder Tools beziehungsweise eine Kombination Mehrerer stellen sich die folgenden Anforderungen:

A1 Es muss die Durchführung von Komponententests möglich sein.

A2 Es muss die Durchführung von End-To-End-Tests möglich sein.

A3 Es muss der Test von AngularJS-Anwendungen möglich sein.

A4 Es muss die Testausführung von Komponententests im Browser aus der Konsole heraus möglich sein.

A5 Sie muss Open-Source und für den kommerziellen Einsatz freigegeben sein.

A6 Sie soll durch eine Open-Source-Community aktiv gewartet werden; Fehler in ihr sollen behoben werden.

A7 Der Testcode soll leicht lesbar sein[Com17a, S. 7]; hierzu bietet sich der BDD-Stil an, den das Testframework somit unterstützen soll.

A8 Die Software soll eine geringe Einarbeitungszeit erfordern und problemlos zu verwenden sein.

A9 Sie soll dem Entwickler Flexibilität beim Schreibe der Testfallimplementationen und Assertions bieten, damit dieser möglichst nah an seinen persönlichen Präferenzen arbeiten kann.

A10 Die Ausgabe der Testergebnisse muss konfigurierbar sein, um sie gegebenenfalls in Prozessen oder anderen Tools weiterverwenden zu können.

A11 Bei der Durchführung von End-To-End-Tests soll es möglich sein, die Ergebnisse mit Screenshots zu dokumentieren.

A12 Der Einsatz von Spies, Stubs und Mocks muss bei Unit-Tests möglich sein.

Aufgrund diverser regulatorischer Vorgaben, Prozesse und vorhandener technischer Systeme innerhalb des Commerzbank-Konzerns ergeben sich weitere Anforderungen:

CB1 Die Software muss unter Windows 7 funktionieren.

CB2 Es darf keine zusätzliche Software zur Ausführung erforderlich sein; lediglich Node.js und Java sind zulässig, da diese über die Standardsoftwareverteilung der Commerzbank verfügbar sind.

CB3 Wenn die Software Node.js benötigt, muss sie mit Version 6.9.2 lauffähig sein.

CB4 Die Software muss über den Node Package Manager installierbar sein.

CB5 Gemäß dem SEF der Commerzbank müssen Testfälle eindeutig beschrieben werden; diese Beschreibung auch kann durch Implementierung der Tests erfolgen[Com17c]. Das Testframework soll daher eine Beschreibung der Tests forcieren.

CB6 Gemäß dem SEF sollen Komponententests „einen möglichst großen Teil der Funktionalität abdecken“[Com17c]. Zur Überprüfung dessen muss eine Möglichkeit zur Ermittlung der Code Coverage bestehen.

CB7 Das Bearbeiten aller Dateien in der TFS-Quellcodeverwaltung soll möglich sein, es dürfen somit keine proprietären Binärdateien zum Einsatz kommen.

Zur Erfüllung der Anforderung A1 werden ein *Testrunner*, ein *Testframework* zum Anlegen und Beschreiben der Testfälle sowie eine *Assertion-Bibliothek* benötigt. Zur Erfüllung von Anforderung A2 wird zusätzlich ein *Tool zur Browsersteuerung* benötigt.

3.2. Softwarerecherche

Einen Einstiegspunkt und ersten Überblick über zur Verfügung stehende Software bietet [Jon17]. Diese Software wird in den folgenden Abschnitten untersucht. Querverweisen auf neue Software wird dabei nachgegangen.

3.2.1. Karma

Karma ist ein Testrunner. 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 selbst bietet keine Möglichkeit zur Implementierung von Testfällen und benötigt daher ein Testframework wie Jasmine (s. Abschnitt 3.2.6), Mocha (s. Abschnitt 3.2.2) oder QUnit (s. Abschnitt 3.2.4). Auch Continuous Integration Server wie Jenkins oder Travis werden unterstützt. [Kar17c]

Grundlegend basiert Karma auf einem Client-Server-Prinzip. Karma startet einen Webserver, 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 Testframework Jasmine (s. Abschnitt 3.2.6) 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 wird und in diesem die Tests ausgeführt werden sollen. [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 3.2.9) 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]

3.2.2. Mocha

Mocha ist ein Testframework und Testrunner, 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 Testsuites, welche weitere Testsuites oder Testfälle enthalten können. Für jede Testsuite und jeden Testfall muss ein Name vergeben werden, durch welchen der Test beschrieben wird. Testfälle bestehen aus funktionalem Code sowie Assertions als eigentliche Testüberprüfung. Es werden verschiedene Stile zur Testbeschreibung unterstützt: BDD, TDD, QUnit (s. Listing 10 in Abschnitt 3.2.4) und weitere, welche sich nur in ihrem Aussehen unterscheiden und Entwicklern ermöglichen, ihren eigenen Stil zur Definition von Tests zu wählen. [Moc17]

```
describe('#indexOf()', function() {
  context('when not present', function() {
    it('should not throw an error', function() {
      (function() {
        [1,2,3].indexOf(4);
      }).should.not.throw();
    });

    it('should return -1', function() {
      [1,2,3].indexOf(4).should.equal(-1);
    });
  });
  context('when present', function() {
    it('should return the index where the element first appears in the
    array', function() {
      [1,2,3].indexOf(3).should.equal(2);
    });
  });
});
```

Listing 8: Beispiel eines Tests mit dem BDD-Stil in Mocha (adaptiert nach [Moc17])

In Listing 8 findet sich ein beispielhafter Test unter Verwendung des BDD-Stils. Die Funktionen `describe` und `context` definieren Testsuites; sie unterscheiden sich nur zwecks besserer semantischer Lesbarkeit des Tests. Beim TDD-Stil werden stattdessen die Funktionen `suite` und `test` zur Definition von Testsuites bzw. -fällen verwendet. [Moc17]

Für Assertions können in Mocha verschiedene Frameworks genutzt werden. In [Moc17] wird beispielsweise die Nutzung von *should.js*, *expect.js* oder *chai* (s. Abschnitt 3.2.7) 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.

Mocha ermöglicht es, die Testausgabe beliebig zu konfigurieren, so dass beispielsweise eine Ausgabe in der Konsole, als HTML-Datei, als JSON oder im XML-Format möglich ist. Bei besonderen Anforderungen können eigene Reporter erstellt werden, z. B. zur Einbindung in Continuous-Integration-Tools.

3.2.3. AVA

AVA ist ein komplettes Test-Tool, also Testrunner und Testframework und beinhaltet eine Assertion-Bibliothek. Die Besonderheit ist, dass die Tests simultan ausgeführt werden und somit deutliche Performanceverbesserungen möglich sind. Es steht unter `ava` im npm-Repository zur Verfügung. [Sor+17]

AVA läuft ausschließlich in Node.js; es gibt also keine Möglichkeit den Testrunner im Browser aufzurufen. Es eignet sich somit nur bedingt für den Test von AngularJS, da hierfür meist ein DOM benötigt wird. Ein Test wird über den Aufruf einer Funktion, welche aus dem Node-Modul *ava* importiert wird, definiert. Dieser Funktion wird ein Funktionskörper übergeben, welcher den Test spezifiert und Assertions durchführt. Ein Beispiel findet sich in Listing 9. Die Assertion-Funktionen werden über das übergebene Testobjekt `t` bereitgestellt. Es ist nicht vorgesehen andere Assertion-Bibliotheken zu nutzen. Mit den Funktionen `before`, `after`, `beforeEach` und `afterEach` können Funktionskörper definiert werden, welche vor oder nach jedem Test oder einmalig allen Tests ausgeführt werden. [Sor+17]

```
import test from 'ava';

test(t => {
  var testObject = [1, 2];
  t.deepEqual(testObject, [1, 2]);
});
```

Listing 9: Beispiel eines Tests mit AVA (adaptiert nach [Sor+17])

3.2.4. QUnit

QUnit ist ein Test-Tool für automatisierte Komponententests mit JavaScript und wird von jQuery und einer Vielzahl weiterer Projekten genutzt. Es beinhaltet Testrunner, Testframework und eine Assertion-Bibliothek. 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. 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 10). 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 10: Beispiel mehrerer Tests für QUnit (adaptiert nach [The17a])

3.2.5. Intern

Intern ist ein Tool für automatisierte Tests in JavaScript. Es bietet sowohl Möglichkeiten für Komponenten- als auch für End-To-End-Tests und besteht aus Testrunner, Testframework, Assertion-Bibliothek und einer Schnittstelle zur Browsersteuerung. Intern ist flexibel und bietet dem Entwickler Möglichkeiten, seinen eigenen Stil zu verfolgen. Es steht als Paket `intern` über npm zur Verfügung. [Int17]

Intern stellt verschiedene Interfaces zur Definierung von Tests bereit; es können auch eigene definiert werden. Das *Object*-Interface ist eine einzelne Funktion, welcher ein Objekt übergeben wird, welches alle definierten Tests enthält. In diesem Objekt werden außerdem Funktionen die vor oder nach jedem Test oder der Testsuite ausgeführt werden sollen definiert. Ein Beispiel findet sich in Listing 11. [Int17]

```
define(function (require) {
  var registerSuite = require('intern!object');

  registerSuite({
    name: 'Suite name',
    beforeEach: function (test 3.0) {
      // executes before each test
    },
    afterEach: function (test 3.0) {
      // executes after each test
    },
    'Test foo': function () {
      // a test case
    },
    'Test bar': function () {
      // another test case
    },
  });
});
```

Listing 11: Beispiel einer Testsuite in Intern (aus [Int17])

Die Interfaces *BDD* und *TDD* ähneln einander und unterscheiden sich nur durch die Benennung einzelner Funktionen. Sie verfolgen gegenüber dem Object-Interface einen prozeduraleren Ansatz, basieren also auf verschachtelt aufgerufenen Funktionen statt auf Objekten. Auch hier können Funktionen definiert werden, welche vor oder nach jedem Test oder der Suite aufgerufen werden. Ein Beispiel ist in Listing 12 zu finden. [Int17]

```
define(function (require) {
  var bdd = require('intern!bdd');

  bdd.describe('the thing being tested', function () {
    bdd.beforeEach(function () {
      // executes before each test
    });
    bdd.afterEach(function () {
      // executes after each test
    });
    bdd.it('should do foo', function () {
      // a test case
    });
    bdd.it('should do bar', function () {
      // another test case
    });
  });
});
```

Listing 12: Beispiel einer Testsuite in Intern (aus [Int17])

Es steht ein Interface zur Verfügung, welches QUnit nachempfindet und somit die Verwendung von in QUnit geschriebenen Tests (s. Listing 10 in Abschnitt 3.2.4) in Intern ermöglicht. Für alle Tests gilt, dass ein Test fehlschlägt, wenn in ihm ein Error auftritt, also eine Assertion fehlschlägt. Ansonsten gilt er als bestanden. Die Chai-Bibliothek ist in Intern enthalten, es ist jedoch auch die Nutzung von beliebigen anderen Assertion-Frameworks möglich. Durch Aufruf der Funktion `skip` können Tests übersprungen werden. [Int17]

End-To-End-Tests werden genau wie Unit-Tests definiert, jedoch in der Konfiguration in `functionalSuites` und nicht in `suites` aufgeführt. Hierdurch werden sie im Kontext des Test-Runners und nicht in der zu testenden Umgebung ausgeführt. Für die Interaktion mit dem Browser verwendet Intern mit *leadfoot* einen Wrapper für Selenium. Über das in `this.remote` zur Verfügung gestellte Leadfoot-Command-Objekt können Befehle im Browser ausgeführt werden. [Int17]

```
define(function (require) {
  var registerSuite = require('intern!object');
  var assert = require('intern/chai!assert');

  registerSuite({
    name: 'index',

    'greeting form': function () {
      return this.remote
        .get(require.toUrl('index.html'))
        .setFindTimeout(5000)
        .findByCssSelector('body.loaded')
        .findById('nameField')
          .click()
          .type('Elaine')
          .end()
        .findByCssSelector('#loginForm input[type=submit]')
          .click()
          .end()
        .findById('greeting')
          .getVisibleText()
          .then(function (text) {
            assert.strictEqual(text, 'Hello, Elaine!',
              'Greeting should be displayed when the form is submitted');
          });
    }
  });
});
```

Listing 13: Beispiel eines End-To-End-Tests in Intern (aus [Int17])

Im Beispiel in Listing 13 wird ein Testfall „greeting form“ definiert, in welchem die Index-Seite geladen wird und auf dieser in einem Eingabefeld der Wert „Elaine“ eingegeben und der Submit-Button geklickt wird. Abschließend wird überprüft ob das Element mit der ID „greeting“ den korrekten Inhalt enthält.

Intern ermittelt standardmäßig beim Ausführen von Tests die Code Coverage, also die Abdeckung von Codezeilen, Funktionen, Zweigen und Anweisungen. Die Ausgabe sowohl von Code Coverage als auch der Testergebnisse ist konfigurierbar. [Int17]

3.2.6. Jasmine

Jasmine ist ein Testframework und Testrunner für Tests im BDD-Stil und beinhaltet eine Assertion-Bibliothek [Jas17b]. Es liegt als `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 Assertions [Jas17b]. Ein beispielhafter Test findet sich in Listing 14.

Eine Testsuite beginnt auf oberster Ebene mit dem Aufruf der globalen JavaScript-Funktion `describe`, wobei ein beschreibender String und eine Funktion zur Implementierung der Suite übergeben wird. [Jas17b]

Testsuites enthalten eine beliebige Menge von Specs, welche durch Aufruf der globalen Funktion `it` angelegt werden. Ein zu übergebender 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. [Jas17b]

Eine Assertion besteht in Jasmine aus der Funktion `expect`, 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 14: Beispiel einer Jasmine-Testsuite (adaptiert nach [Jas17b])

3.2.7. Chai

Chai ist eine Assertion-Bibliothek, welche mit jedem Testframework kombiniert werden kann. Chai bietet verschiedene Assertion-Stile, so dass der Entwickler seinen eigenen Stil wählen kann. [Cha17c] Chai ist unter der ID `chai` über npm verfügbar und kann so installiert und in Projekte eingebunden werden [Cha17d].

Der Assert-Stil ähnelt klassischeren Testframeworks wie QUnit oder dem Assert-Modul in Node.js. Über das `assert`-Objekt werden Funktionen zur Verfügung gestellt, zum Beispiel `isOk` zur Überprüfung, ob der Parameter truthy ist oder `equal`

zur Überprüfung, ob die Parameter gleich sind. Jeder Funktion kann eine optionale Nachricht übergeben werden, welche im Falle eines Fehlschlags in der Fehlermeldung angezeigt wird. [Cha17a]

Der BDD-Stil wird in zwei Varianten zur Verfügung gestellt: `expect` und `should`. Er ermöglicht es Assertions in einer natursprachlichen Form zu schreiben, welche somit gut lesbar sind. Die should-Variante hat einige Nachteile, weshalb an dieser Stelle nur die expect-Variante betrachtet wird. `Expect` ist eine Funktion, welcher der zu überprüfende Wert übergeben wird. Diese Methode wird mit weiteren Objekten und Funktionen verkettet, um die Assertion zu bilden. [Cha17b] Ein Beispiel hierzu findet sich in Listing 15.

```
var expect = require('chai').expect
  , foo = 'bar'
  , beverages = { tea: [ 'chai', 'matcha', 'oolong' ] };

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(beverages).to.have.property('tea').with.lengthOf(3);
```

Listing 15: Beispiel von Assertions mit dem expect-Stil von Chai (aus [Cha17b])

3.2.8. Protractor

Protractor ist ein speziell für Angular-Anwendungen entwickeltes Tool zur Browsersteuerung 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]

Protractor nutzt als Framework für die Testbeschreibung standardmäßig Jasmine (s. Abschnitt 3.2.6), unterstützt out-of-the-box aber auch Mocha (s. Abschnitt 3.2.2). 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 16) konfiguriert.

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

Listing 16: 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 17). 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 17: Beispiel einer Spec für Protractor (aus [Pro17c])

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 3.2.9) zusammen mit Protractor wird ausdrücklich abgeraten, da es hier Berichten zufolge häufig zu Abstürzen und abweichendem Verhalten kommt [Pro17b]. Selenium wird automatisch zusammen mit Protractor installiert und ist nach Aufruf von `webdriver-manager update` und `webdriver-manager start` ohne weitere Konfiguration lauffähig [Pro17a].

3.2.9. PhantomJS

PhantomJS ist ein skriptbarer WebKit-Browser ohne Benutzungsoberfläche und ist über eine JavaScript-API ansteuerbar. Er bietet native Unterstützung für Webstandards wie DOM, CSS-Selektoren, JSON, HTML-Canvas und SVG. [Hid17c] PhantomJS steht nicht als npm-Paket zur Verfügung, sondern lässt sich lediglich als Binary installierten [Hid17a]. PhantomJS ist auch eine Laufzeitumgebung für JavaScript, so dass für ihn bestimmte Skripte nicht in Node.js ausgeführt werden können, sondern nur in PhantomJS [Hid17d].

Ein großer Nutzungsbereich von PhantomJS liegt im Testen von Webanwendungen; geeignet ist es beispielsweise für den Einsatz in Kommandozeilenumgebungen oder Continuous-Integration-Systemen. PhantomJS an sich ist kein Testframework, sondern dient der Ausführung eines beliebigen Testframeworks. Es existieren Frameworks welche speziell auf PhantomJS aufbauen und komfortable Funktionalitäten für Testzwecke zur Verfügung stellen: z. B. CasperJS (s. Abschnitt 3.2.10) oder WebSpecter, welches sich allerdings noch in einer frühen Entwicklungsphase befindet. [Hid17b]

Für das Laden von Webseiten sind Page-Objekte zuständig, über welche eine URL geladen werden kann, Screenshots gespeichert werden können oder auf DOM-Eigenschaften zugegriffen werden kann. Ein Beispiel findet sich in Listing 18. JavaScript-Code kann im Kontext einer geladenen Seite mit der `evaluate`-Funktion ausgeführt werden; die Ausführung erfolgt dann in einer Sandbox und kann somit nicht auf Objekte, Variablen oder Funktionen außerhalb des Kontexts zugreifen. [Hid17d]

```
var page = require('webpage').create();
page.open('http://example.com', function(status) {
  console.log("Status: " + status);
  if(status === "success") {
    page.render('example.png');
  }
  phantom.exit();
});
```

Listing 18: Beispiel eines Seitenaufrufs mit PhantomJS (aus [Hid17d])

3.2.10. CasperJS

CasperJS ist ein Framework für Navigation und Test in PhantomJS, und lässt sich somit als Tool zur Browsersteuerung kategorisieren. Es steht unter `casperjs` im npm-Repository zur Verfügung. [Per+17a] Trotz dessen ist es nicht unter Node.js lauffähig [Per+17b], sondern benötigt Python [Per+17c].

Es ermöglicht die Erstellung von komplexen Navigationsszenarien unter Benutzung von High-Level-Funktionen. Hierzu werden u. a. die Funktionen `casper.start`, `casper.then`, `casper.thenOpen` und `casper.back` sequentiell aufgerufen. Das Szenario kann dann mittels `casper.run` aufgerufen werden und wird nacheinander abgearbeitet. Es stehen diverse Funktionen wie `casper.click`, `casper.fill` oder `casper.evaluate` zur DOM-Manipulation zur Verfügung. Für alle Browser-Operationen und DOM-Manipulationen wird als Browser PhantomJS genutzt, welcher von CasperJS gestartet wird. Die Ansteuerung von PhantomJS wird durch CasperJS vereinfacht, wodurch sich einfacher wartbarerer Code ergibt [Dur15]. Ein beispielhaftes Navigationsszenario findet sich in Listing 19. [Per+17a]

```
var casper = require('casper').create();
casper.start('http://casperjs.org/');

casper.then(function() {
    this.echo('First Page: ' + this.getTitle());
    this.click('#link-quickstart-full');
});

casper.then(function() {
    this.echo('Second Page: ' + this.getTitle());
});

casper.run();
```

Listing 19: Beispiel eines Navigationsszenarios mit CasperJS (adaptiert nach [Per+17a])

CasperJS enthält auch ein einfaches Testframework. Ein Test wird durch Aufruf der Funktion `casper.test.begin` definiert und mit `test.done` beendet. Er gilt als erfolgreich, wenn er ohne fehlgeschlagene Assertions beendet wurde. Navigationsszenarien und Tests können verschachtelt werden, so dass mit CasperJS auch End-To-End-Tests durchgeführt werden können. [Per+17a; Per+17d] Ein einfacher Test ist in Listing 20 zu finden.

```
casper.test.begin('Cow can moo', 2, function suite(test) {
    var cow = new Cow();
    test.assertEquals(cow.moo(), 'moo!');
    test.assert(cow.mowed);
    test.done();
});
```

Listing 20: Beispiel eines Tests mit CasperJS (aus [Per+17d])

3.2.11. Sinon

Sinon ist ein Framework für die Realisierung von Fakes, also Mocks, Stubs und Spies, in JavaScript und arbeitet mit jedem Unit-Test-Framework zusammen. Es ist als `sinon` über npm verfügbar. [Sin17c]

Spies sind Funktionen die relevante Aufrufdaten aufzeichnen: Argumente, Rückgabewert, geworfene Exceptions und den Aufrufer. Ein Spy kann sowohl als anonyme Funktion - durch Aufruf von `sinon.spy()` - als auch als Wrapper für existierende Methoden - dann durch Aufruf von `sinon.spy(object, 'method')` für das Überschreiben von `object.method()`. [Sin17d] Ein Beispiel für einen Spy auf einer anonymen Funktion findet sich in Listing 21.

```
it('calls the original function', function () {
    var callback = sinon.spy();
    var proxy = callFunction(callback);

    proxy();

    assert(callback.called);
});
```

Listing 21: Beispiel eines anonymen Spy in Sinon (aus [Sin17c])

Stubs sind Spies mit einem vorprogrammierten Verhalten. Hierzu verfügen sie über die komplette Spy-API und zusätzliche Methoden, mit welchen ihr Verhalten angepasst werden kann. Anders als bei Spies wird bei einem Stub, welcher eine existierende Funktion überschreibt, diese nicht ausgeführt. Sie können benutzt werden um ein bestimmtes Verhalten von Funktionen zu provozieren, z. B. durch

das Werfen von Fehlern, oder um zu verhindern, dass Funktionen ausgeführt werden, z. B. damit `XMLHttpRequest` keinen HTTP-Request absetzt. Ein Aufruf von `sinon.stub().throws()` erzeugt beispielsweise einen anonymen Stub, welcher bei Aufruf eine Exception wirft. [Sin17e]

Mocks sind Stubs, welche zusätzlich vorprogrammierte Erwartungen haben. Ein Mock ist somit ein Stub, welcher Assertions enthält. Es wird empfohlen, dass maximal ein Mock pro Unittest existiert. Ein Beispiel für einen Mock, in welchem erwartet wird, dass die gemockte Methode einmal aufgerufen wird und diese eine Exception werfen soll, findet sich in Listing 22. Das Eintreffen der definierten Erwartungen wird letztlich durch Aufruf von `verify()` überprüft - ein Nicht-Zutreffen führt wie bei Assertions zum Fehlschlag eines Tests. [Sin17a]

```
"test should call all subscribers when exceptions": function () {
    var myAPI = { method: function () {} };

    var spy = sinon.spy();
    var mock = sinon.mock(myAPI);
    mock.expects("method").once().throws();

    PubSub.subscribe("message", myAPI.method);
    PubSub.subscribe("message", spy);
    PubSub.publishSync("message", undefined);

    mock.verify();
    assert(spy.calledOnce);
}
```

Listing 22: Beispiel eines Mocks in Sinon (aus [Sin17a])

Sinon ermöglicht es, mit der Funktion `sinon.restore` alle Fakes, die auf einem übergebenen Objekt definiert wurden, zurückzusetzen. Es bietet außerdem die Möglichkeit *Sandboxes* anzulegen, in welche alle angelegten Fakes abgelegt werden. Dies erleichtert das Aufräumen und Entfernen aller Fakes, da dies nun nicht mehr einzeln geschehen muss, sondern ein Aufruf von `sandbox.restore` genügt. [Sin17b]

3.2.12. ngMock

Bei ngMock handelt es sich um ein AngularJS-Modul, mit welchem andere Komponenten in Unit-Tests injiziert und gemockt werden können. Außerdem erweitert es diverse AngularJS-Kernservices, so dass diese in Testcode kontrolliert und inspiziert werden können. Es steht im npm-Repository unter `angular-mocks` zur Verfügung und muss in der Konfiguration des verwendeten Test-Runners so eingebaut werden, dass es nach `angular.js` geladen wird. [Goo17f]

Die Funktionsweise basiert auf den Methoden `angular.mock.module`, welche die übergebenen Module lädt [Goo17b], und `angular.mock.inject`, welche eine übergebene Funktion in eine Injizierbare wrapt, eine neue Injector-Instanz erstellt und die angegebenen Abhängigkeiten injiziert [Goo17a]. Methoden von injizierten Abhängigkeiten können, beispielsweise mit Sinon (s. Abschnitt 3.2.11), mit Spies oder Mocks ersetzt werden.

```
describe('SimpleService', function() {
  var SimpleService, $log;

  beforeEach(module('app')); //load the module under test

  it('should log the message "something done!"', inject(function(
    SimpleService, $log) {
     sinon.spy($log, 'info');
     SimpleService.doSomething();
     assert($log.info.calledOnce);
     assert($log.info.calledWith('something done!'));
     $log.info.restore();
  }));
});
```

Listing 23: Beispiel eines Tests mit injizierten Abhängigkeiten mit ngMock (adaptiert nach [Wat15])

Im Beispiel (s. Listing 23) wird das Module `app` geladen und die Services `SimpleService` und `$log` in die Testfunktion injiziert. Dadurch kann ein Spy auf `$log.info` gesetzt werden und hierdurch das korrekte Logging von `SimpleService.DoSomething` validiert werden.

3.2.13. Istanbul

Istanbul ist ein Tool zur Ermittlung der Code-Coverage bei JavaScript-Ausführungen. Es ist als `nyc` im npm-Repository verfügbar. [Hen+17]

Der Aufruf von Istanbul erfolgt durch Voranstellung des Befehls `nyc` vor den eigentlichen Aufruf des Test-Befehls (z. B. `nyc mocha` zur Ermittlung der Code-Coverage bei einem Mocha-Aufruf). Istanbul bestimmt dann bei Ausführung die Zeilen-, Statement-, Funktions- und Zweigabdeckung für jede einzelne JavaScript-Datei. Das Ausgabeformat ist konfigurierbar und somit an die individuellen Bedürfnisse anpassbar. [Hen+17]

Istanbul unterstützt ES5, ES6, ES2015+ und laut eigener Aussage die meisten Testframeworks. Explizit genannt werden tap, Mocha (s. Abschnitt 3.2.2) und AVA (s. Abschnitt 3.2.3) [Hen+17], es wird jedoch auch Karma (vgl. [Kro+15]) unterstützt.

3.3. Auswahlentscheidung

Für die Entscheidungsfindung wird eine Entscheidungsmatrix (s. Tabelle 1) erstellt, in welcher die Anforderungen gegen die gefundene Software abgeglichen werden. Ein grüner Haken steht für ein Erfüllen der Anforderung; ein rotes Kreuz für einen Widerspruch. Eine leere Zelle bedeutet, dass die Anforderung bei dieser Software nicht relevant ist, da sie weder positiv noch negativ beeinträchtigt wird. Eine Anforderung wird von der ausgewählten Softwaremenge erfüllt, wenn keine Einzelsoftware der Anforderung widerspricht.

Formuli

Wie in Abschnitt 3.1 bereits beschrieben werden zur Erfüllung von Anforderung A1 ein Testrunner, ein Testframework sowie eine Assertion-Bibliothek benötigt; zur Erfüllung von Anforderung A2 zusätzlich ein Tool zur Browsersteuerung. Hierzu werden die gefundenen Programme kategorisiert. Die endgültig ausgewählte Softwaremenge muss mindestens ein Exemplar jeder Kategorie enthalten, damit die Anforderungen A1 und A2 erfüllt werden.

		Software												
		Karma	Mocha	AVA	QUnit	Intern	Jasmine	Chai	Protractor	PhantomJS	CasperJS	Sinon	ngMock	Istanbul
Kategorie	Testrunner	✓	✓	✓	✓	✓	✓				✓			
	Testframework		✓	✓	✓	✓	✓				✓			
	End-To-End					✓			✓	✓	✓			
	Assertion			✓	✓	✓	✓	✓			✓			
	Sonstige											✓	✓	✓
A3	AngularJS-Support	✓	✓	✗[1]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A4	Testausführung	✓					✓				✗			
A5	Open Source & kommerziell	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A6	Wartung[2]	✓	✓	✓	✓	✓	✓	✓	✓	✗[3]	✓	✓	✓	✓
A7	BDD-Stil		✓	✗	✗	✓	✓	✓			✗			
A8	Einarbeitung & Verwendung	✓	✓	✓	✓	✗[4]	✓	✓	✓	✓	✗[5]	✓	✓	✓
A9	Flexibilität		✓	✗	✗	✓	✗				✗			
A10	Ausgabe-Konfiguration	✓	✓	✓	✓	✓	✓				✗			✓
A11	Screenshots					✗			✓	✓	✓			
A12	Spies, Stubs, Mocks											✓	✓	
CB1	Windows 7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CB2	benötigte Software	✓	✓	✓	✓	✓	✓	✓	✓[6]	✓	✗	✓	✓	✓
CB3	Node.js 9.6.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CB4	npm	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
CB5	Forcierte Beschreibung		✓	✗	✓	✓	✓				✓			
CB6	Code-Coverage					✓								✓
CB7	keine proprietären Dateien	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabelle 1: Entscheidungsmatrix zur Softwareauswahl

Die Auswertung der Entscheidungsmatrix zeigt, dass AVA, QUnit, Intern, Jasmine, PhantomJS sowie CasperJS nicht eingesetzt werden können, da sie Anforderungen widersprechen. Die Verbleibenden bilden die Basis für die zu treffende Auswahl aufgrund der Anforderungen. Es zeigt sich, dass nur durch den Einsatz sämtlicher

[1] `https://github.com/angular/angular.js/issues/13971`
[2] Zur Auswertung dieser Anforderung wird das jeweilige Github-Repository herangezogen.
[3] 1.841 offene Issues; 2 Issues geschlossen in der letzten Woche [Git17a; Git17b]
[4] Warten auf asynchrone AngularJS-Komponenten muss bei End-To-End-Tests manuell implementiert werden.
[5] Warten auf asynchrone AngularJS-Komponenten muss manuell implementiert werden.
[6] Zur Ausführung wird zwar Selenium benötigt, dies wird aber automatisiert mitinstalliert. Die Software erfüllt die Anforderung daher.

verbleibender Software sämtliche Anforderungen abgedeckt werden können. Es wird somit folgende Auswahl getroffen:

Als Test-Runner wird Karma eingesetzt, welches die benötigten JavaScript-Dateien importiert, bereitstellt und aus der Konsole heraus die in Mocha implementierten Tests ausführt. Die sich aus der Matrix ergebende Kombination von Mocha mit der Assertionbibliothek Chai wird von Mocha [Moc17] bereits in der Softwaredokumentation aufgeführt, wodurch hier von einem problemlosen Zusammenarbeiten auszugehen ist. Für die Ermittlung der Code-Coverage wird Istanbul eingesetzt; die Kombination des Einsatzes mit Mocha wird in [Hen+17] explizit empfohlen, so dass auch hier von keinerlei Problemen auszugehen ist.

Für die Realisierung von End-To-End-Tests wird auf das vom AngularJS-Team entwickelte Protractor zurückgegriffen und somit auch der Empfehlung aus der offiziellen AngularJS-Dokumentation [Goo17i] gefolgt. Die abweichende Kombination von Protractor mit Mocha und Chai, statt Jasmine, wird von vielen Entwicklern genutzt und unter Anderem in [Asu14] und [ByV14] beschrieben, so dass auch hier von keinen Problemen auszugehen ist.

Für die Realisierung von Spies, Stubs und Mocks in Komponententests wird auf Sinon und ngMock gesetzt.

4. Evaluierung

Im folgenden Kapitel wird die getroffene Softwareauswahl evaluiert, indem für ein gegebenes, bereits existierendes AngularJS-Projekt im Commerzbank-Konzern prototypisch Tests implementiert werden. Hierbei werden nicht Tests für alle Komponenten und Funktionalitäten der Anwendung entwickelt, sondern nur so viele, bis sich die getroffene Softwareauswahl als problemlos einzusetzen herausstellt.

4.1. Projektbeschreibung

Das Projekt GFB stellt ein digitales Formular bereit, mittels welchem Gefährdungsbeurteilungen erfasst werden können. Dies sind Dokumente, welche beispielsweise von Fachkräften für Arbeitssicherheit nach Betriebsbegehungen erstellt werden und in denen festgestellte Mängel aufgeführt werden. [Com14]

Die Anwendung ist eine Single-Page-Application und wurde unter Nutzung von AngularJS entwickelt. Bereitgestellt wird sie in einer Microsoft SharePoint-Umgebung, wodurch sich einige Besonderheiten ergeben: z. B. die Nutzung von SOAP zur Abfrage von WebServices. Durch die SharePoint-Umgebung funktioniert die Anwendung ausschließlich im Internet Explorer und nicht im zweiten Commerzbank-Standardbrowser Firefox. Für die Anwendung existieren keine automatisierten Tests; jegliches Testen wird bisher auf der Oberfläche im Testsystem durchgeführt. [Com14]

4.2. Implementierung

Die Implementierung der Tests erfolgt in drei Iterationen: Zunächst wird eine Testumgebung, bestehend aus Karma, Mocha, Chai, Sinon und Istanbul, aufgesetzt und gezeigt, dass diese funktioniert. Anschließend werden prototypische Komponententests für alle verschiedenen Komponententypen (Controller, Factory, Filter, Direktive) in AngularJS implementiert. Abschließend werden Systemtests als End-To-End-Tests umgesetzt; auch dies geschieht prototypisch und hat keine hundertprozentige Abdeckung zum Ziel.

4.2.1. Testumgebung

Das bestehende Projekt konnte problemlos in ein npm-Projekt überführt werden. Alle existierenden Quelldateien wurden in einen Unterordner `src` verschoben und folgende Ordnerstruktur geschaffen:

```
/
├── coverage ..... Speicherort für Code-Coverage-Reports
│   ├── html
│   └── lcov
├── node_modules ..... Speicherort für npm-Module
├── src ..... Originaldateien der GFB-Anwendung
└── test
    └── spec
```

Über npm konnten alle für die Ausführung von Tests benötigten Komponenten installiert werden: Karma mitsamt diverser Plugins, Mocha, Chai, Istanbul, angular-mocks, Sinon und Protractor. Auch hierbei traten keinerlei Probleme auf. Die fertige `package.json`-Datei ist in Listing 29 im Anhang zu finden.

Zur Verifizierung der Einsatzbereitschaft der installierten Software wurde eine einfache Testsuite geschrieben, welche eine Assertion durchführt und ein Spy mit Sinon erstellt. Für die Verifizierung von ngMock sowie die korrekte Einbindung aller originalen Projektdateien wurde eine weitere Testsuite erstellt, in welcher in zwei Testfällen ein Service und ein Filter, welche im Originalprojekt definiert sind, von ngMock injiziert werden. Die Testsuites finden sich in Listing 32 und 33 im Anhang.

Karma wurde so konfiguriert, dass es automatisch den Internet Explorer startet und diesen zur Ausführung der Tests verwendet. Hierbei trat das Problem auf, dass der Browser zwar gestartet wird, jedoch von Karma der erfolgreiche Start nicht korrekt erkannt wird. Laut Dokumentation des `karma-ie-launcher` führt Starten des Internet Explorers im *no add-ons mode* zur Lösung dieses Problems [Ham+15]. Nach Vornehmen dieser Änderung wird der Internet Explorer korrekt durch Karma gestartet und die Tests in ihm ausgeführt.

Zur Ermittlung der Code-Coverage wurde das Plugin `karma-coverage` hinzugefügt, durch welches Karma das Code-Coverage-Tool Istanbul startet. Dieses wurde so konfiguriert, dass die erstellen Berichte über die Codeabdeckung im Unterordner `coverage` in zwei Formaten abgelegt werden. Zur manuellen Ansicht im Browser dient das HTML-Format, welches eine durchklickbare Seite bereitstellt, in welcher alle erforderlichen Statistiken angezeigt werden; beispielhafte Screenshots dieser Anzeige finden sich in Abbildung 1 und 2 im Anhang.

Die fertiggestellte Konfigurationsdatei für Karma ist in Listing 30 im Anhang zu finden. Bei Ausführung des Befehls `karma start` treten keine Fehler auf; alle Testfälle werden als bestanden markiert. Es ist daher davon auszugehen, dass die eingerichtete Testumgebung für Komponententests funktioniert.

4.2.2. Komponententests

Nach der Feststellung, dass das Testsystem korrekt eingerichtet wurde, werden prototypisch Tests für alle in der Anwendung vorhandenen AngularJS-Komponenententypen entwickelt.

Controller In der Anwendung existiert ein Controller, der `OverviewCtrl`, welcher somit für Tests ausgewählt wird. Aufgrund der hohen Komplexität der initialisierenden Controller-Funktion eignet diese sich nicht für die Erstellung eines Komponententest. Daher wird eine zum Controller gehörende Scope-Funktion getestet: `$scope.validatebegehungsdatum`. Diese validiert, ob zu einer Begehung ein Datum eingegeben wurde und gibt dann true zurück; ansonsten false. Einer der entwickelten Testfälle findet sich in Listing 24; es handelt sich hierbei um einen Auszug aus der im Anahng in Listing 34 zu findenden gesamten Testsuite.

```
it("should return true given an undefined gefahrenbeurteilung", function
    () {
     var $scope = $rootScope.$new();
     var ctrl = $controller("OverviewCtrl", {$scope: $scope, initialData:
    initialData});
     $scope.gefahrenbeurteilung = undefined;

     expect($scope.validatebegehungsdatum()).to.be.true;
});
```

Listing 24: Testfall aus der zum OverviewCtrl gehörenden Testsuite (s. Listing 34)

Zugriff auf den Controller wird hierbei durch Nutzung des standardmäßigen AngularJS-Services `$controller` erlangt [Goo17d], welcher in der `beforeEach`-Funktion durch ngMock in den Test injiziert wird. Ein neuer Scope wird mit der Methode `$new` des Root-Scopes erzeugt und dem Controller zur Nutzung übergeben. Die Variable `initialData` ist ein in der `beforeEach`-Funktion erzeugtes Dummyobjekt und wird vom Controller zwingend benötigt. Nach den Vorbereitungen kann die zu testende Scope-Funktion aufgerufen und der Rückgabewert mit der Erwartung verglichen werden. Bei Ausführung zeigt sich, dass der Test korrekt ausgeführt wird, keine Fehler ausgegeben und alle definierten Testfälle als bestanden markiert werden.

Factory In der Anwendung existieren viele Factories, von denen jedoch die meisten als externe Software zugeliefert wurden. Für den Test wird daher eine Factory gewählt, welche projektintern entwickelt wurde. Diese sind in der `svc_obs.js` zu finden und beinhalten die hauptsächliche Geschäftslogik der Anwendung. Die Factory `cls_betriebsratsbereich` bildet die Datenstruktur eines Betriebsratsbereiches ab und bietet eine Funktion, diese Datenstruktur aus einem XML-Element zu

befüllen. Ein solches Element ist nach folgendem Muster aufgebaut: `<item ows_ID=1 ows_Title='Betriebsrat Bremen' ows_Sortierung=1 />`. Ein Auszug aus der kompletten Testsuite (s. Listing 35 im Anhang) findet sich in Listing 25.

```
beforeEach(inject(function(svc_sp, cls_betriebsratsbereich) {
    betriebsratsbereich = new cls_betriebsratsbereich(svc_sp);
}));

it("should fill the fields with values from the xmlItem", function() {
    var xmlItem = {attr: sinon.stub()};
    xmlItem.attr.withArgs("ows_ID").returns(42);
    xmlItem.attr.withArgs("ows_Title").returns("Test");
    xmlItem.attr.withArgs("ows_Sortierung").returns(1);
    betriebsratsbereich.loadfromxmlitem(xmlItem);

    expect(betriebsratsbereich.sp_id).to.equal(42);
    expect(betriebsratsbereich.title).to.equal("Test");
    expect(betriebsratsbereich.sortierung).to.equal(1);
});
```

Listing 25: Testfall aus der zu svc_obs gehörenden Testsuite (s. Listing 35)

Vor Ausführung jedes Testfalls wird in der `beforeEach`-Funktion die zu testende Factory injiziert und unter Nutzung des Operators `new` instanziiert. Beim hierbei übergebenen `svc_sp` handelt es sich um eine weitere Factory, welche Methoden zur SOAP-Kommunikation mit dem unterliegenden SharePoint bietet. Der oben beschriebene Aufbau des XML-Elementes wird hier als Dummyobjekt umgesetzt, welches die intern zum Abruf von Attributen genutzte Funktion `attr` als Sinon-Stub enthält. Dieses Stub ist so konfiguriert, dass es die jeweiligen Testwerte zurückgibt. Nach Aufruf der zu testenden Funktion `loadfromxmlitem` können die im Objekt enthaltenen Werte mit den Erwarteten verglichen werden. Die Ausführung der implementierten Testsuite ergibt keine Fehler; alle Tests werden korrekt ausgeführt und als bestanden markiert.

Direktive Für den Test einer Direktive wird `dir_filHb` ausgewählt, welche eine mit einem Button kombinierte Textbox zur Eingabe einer Fillialkennnummer realisiert. Für die Textbox ist in der Direktive eine Funktion definiert, welche aufgerufen wird, wenn der Fokus die Textbox verlässt. Die Funktion validiert die Eingabe auf das erlaubte Format von fünf Ziffern; bei fehlerhafter Eingabe wird ein Alert ausgegeben und die CSS-Klasse `ng-invalid-input` gesetzt. Die vollständige Testsuite findet sich in Listing 36 im Anhang, ein Auszug in Form eines einzelnen Testfalls in Listing 26.

```
describe("given a not numerical value", function () {
    it("should set input class to EingabefeldEinzeile,ng-invalid-input
    and alert", function () {
        var el = getCompiledDirective();

        var setAttributeSpy = sinonSandbox.spy();
        var alertSpy = sinonSandbox.spy(window, "alert");
        var eventDummy = { srcElement: { value: "abc", setAttribute:
    setAttributeSpy } };

        el.isolateScope().myBlur(eventDummy);

        expect(setAttributeSpy.calledWith("class", "EingabefeldEinzeile")
    ).to.be.true;
        expect(setAttributeSpy.calledWith("class", "EingabefeldEinzeile
    ng-invalid-input")).to.be.true;
        expect(alertSpy.called).to.be.true;
    });
});

 function getCompiledDirective() {
    var el = $compile('<filhb myfilhb="filHb" my-tooltip-input="TT" my-
    tooltip-button="TT2"></filhb>')(scope);
    return el;
}
```

Listing 26: Testfall aus der zu dir_filHb gehörenden Testsuite (s. Listing 36)

Zugriff auf die zu testende Direktive wird hierbei über den `$compile`-Service von AngularJS erlangt, welcher einen übergebenen HTML-String in ein Template kompiliert [Goo17c]. Dieses Template wird dann zusammen mit einem übergebenen Scope zu einem Element verbunden. Zugriff auf die im Scope der Direktive vorhandenen Funktionen ist über die Funktion `isolateScope` möglich, was die Ausführung der zu testenden Funktion `myBlur` (Fokus verlässt die Textbox) ermöglicht. Die Ausführung der Testsuite zeigt keine Fehler; alle Tests werden als bestanden markiert und korrekt ausgeführt.

Filter In der Anwendung existiert der Filter `unique`, welcher aus einer übergebenen Liste alle gleichen Einträge entfernt und somit eine eindeutige Liste ohne Dopplungen erzeugt. Für die Überprüfung der Gleichheit zweier Eintrage wird jedoch nicht das gesamte Element verglichen, sondern nur ein einzelner Schlüsselindex, so dass beispielsweise die Elemente `[1, 'Bremen']` und `[1, 'Hamburg']` bei Betrachtung des nullten Schlüsselindex als gleich betrachtet werden. Ein Testfall für den `unique`-Filter

findet sich in Listing 27; es handelt sich hierbei um einen Auszug aus der vollständigen Testsuite aus Listing 37 im Anhang.

```
beforeEach(inject(function ($filter) {
    uniqueFilter = $filter("unique");
}));

it("should return an empty array given a null input-array", function () {
    expect(uniqueFilter(null, 0)).to.deep.equal([]);
});
```

Listing 27: Testfall aus der zum unique-Filter gehörenden Testsuite (s. Listing 37)

Die zu testende Filter-Funktion kann mithilfe des injizierten AngularJS-Services `$filter` in einer Variable gespeichert werden [Goo17e]. Die so erlangte Filterfunktion kann dann in den einzelnen Testfällen aufgerufen und das Ergebnis mit der Erwartung verglichen werden. Bei Ausführung zeigt sich, dass der Test korrekt ausgeführt wird, keine Fehler ausgegeben und alle definierten Testfälle als bestanden markiert werden.

4.2.3. End-To-End-Tests

Für die Durchführung von End-To-End-Tests musste das installierte Protractor eingerichtet werden. Hierzu wurde gemäß der Dokumentation [Pro17a] der Befehl `webdriver-manager update` aufgerufen. Auch nach Einstellung des zu verwendenden Proxyservers konnten die zu installierenden Abhängigkeiten nicht aufgelöst werden, da die Verbindung vom Proxyserver zurückgewiesen wird. Ohne die vorhandenen ausführbaren Dateien des Selenium-Servers (`selenium-server-standalone.jar`) und des Internet-Explorer-Treibers (`IEDriverServer.exe`) ist Protractor nicht funktionsfähig. Zur Feststellung der grundsätzlichen Funktionsfähigkeit wurden die benötigten Dateien über verschiedene, vom Commerzbank-Standard abweichende, Wege bezogen und einzeln im Ordner abgelegt. Nach dieser manuellen Installation konnte der Selenium-Server erfolgreich gestartet werden.

Zum Test der Einsatzbereitschaft wurde Protractor konfiguriert: Die Tests sollen im Internet Explorer ausgeführt werden; als Testframework soll Mocha genutzt werden. Die fertiggestellte Konfigurationsdatei ist in Listing 31 im Anhang zu finden. Als auszuführender Test wurde der Beispieltest aus [Pro17a] verwendet, welcher für die Nutzung mit Mocha abgewandelt wurde (s. Listing 38 im Anhang).

Bei Ausführung von Protractor zeigte sich, dass der Start des Internet Explorers nicht möglich ist und einen Fehler erzeugt. Zur Problemlösung wird unter Anderem die Deaktivierung des *Geschützten Modus* in allen Zonen im Internet Explorer empfohlen (vgl. [Sid16; Hen14]). Das Zutreffen dieser Aussage konnte in einem lokalen System außerhalb des Commerzbank-Netzwerks validiert werden. Jedoch ist die Deaktivierung

des Geschützten Modus' im Internet Explorer eines Commerzbank-Arbeitsplatzes nicht möglich (s. Abbildung 3 im Anhang).

Aufgrund der aufgeführten Problematiken ist ein Einsatz von Protractor im Commerzbank-Netz als unmöglich zu bewerten.

4.3. Auswertung

Zusammenfassend zeigt sich bei der Auswertung der Evaluierung der Softwareauswahl ein gemischtes Bild. Die Auswahl von Software für die Durchführung von Komponententests kann als erfolgreich angesehen werden. Sie funktioniert im Commerzbank-Umfeld problemlos.

Anders stellt sich dies bei der Auswahl für End-To-End-Tests dar: Protractor ist für den Einsatz im Commerzbank-Umfeld nicht geeignet, auch wenn in Abschnitt 3.3 alle Anforderungen als erfüllt gewertet sind. Die zur Verfügung stehenden Alternativen Intern und PhantomJS/CasperJS eignen sich auch nicht für einen Einsatz. Intern basiert genau wie Protractor auf Selenium und würde somit die gleichen Probleme mit der Konfiguration des Internet Explorers hervorrufen. Die Kombination aus PhantomJS und CasperJS hingegen benötigt für die Ausführung zwingend Python, was in der Commerzbank nicht zur Verfügung steht.

5. Ausblick

5.1. Mögliche Verbesserungen durch Einsatz von Angular statt AngularJS

Angular bietet im Vergleich zum älteren AngularJS einige Verbesserungen im Hinblick auf automatisierte Testbarkeit, welche in diesem Kapitel oberflächlich betrachtet werden sollen.

Angular wird zusammen mit der Angular CLI ausgeliefert, einem Kommandozeilentool zur Vereinfachung häufig benötigter Abläufe. Der integrierte Assistent zur Generierung der Grundstruktur neuer Angular-Projekte bindet in diese auch eine Unterstützung für automatisierte Tests ein: ein konfiguriertes Karma sowie passende Ordnerstrukturen. Dieses kann umkonfiguriert werden, so dass auch die Nutzung von Mocha und Chai, statt dem standardmäßigen Jasmine, möglich ist. [Sil+17]

Angular bietet die *Angular Testing Utilities*, wobei es sich um Klassen und Funktionen handelt, welche die Testdurchführung erleichtern. Sie ermöglichen es, ein `TestBed` zu erstellen, mit welchem ein Testmodul geschaffen wird, welches die zu testende Komponente enthält. Aus diesem Testmodul kann ein `TestFixture` kreiert werden, welches Zugriff auf die zu testende Komponente sowie auf das, der Komponente zugeordnete, DOM-Element bietet. Dies bietet im Vergleich zu Tests mit ngMock in AngularJS den Vorteil einer besseren Lesbarkeit und dem einfacheren Zugriff auf interne Funktionen und Eigenschaften. Ein beispielhafter Test unter Nutzung der Angular Testing Utilities findet sich in Listing 28. [Goo17m]

Zusammenfassend lässt sich sagen, dass die Realisierung von automatisierten Tests in neueren Versionen von Angular mit weniger Problemen und deutlich einfacher möglich ist, so dass sich mögliche Anfangshindernisse und -schwellen verringern und so den Einstieg eines Entwicklers in automatisierte Tests erleichtern.

```
describe('BannerComponent (inline template)', () => {
  let comp:    BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;
  let de:      DebugElement;
  let el:      HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ], // declare the test component
    });

    fixture = TestBed.createComponent(BannerComponent);

    comp = fixture.componentInstance; // BannerComponent test instance

    // query for the title <h1> by CSS element selector
    de = fixture.debugElement.query(By.css('h1'));
    el = de.nativeElement;
  });

  it('no title in the DOM until manually call `detectChanges`', () => {
    expect(el.textContent).toEqual('');
  });
});
```

Listing 28: Beispielhafter Komponententest einer *Component* in Angular; geschrieben in TypeScript (aus [Goo17m])

5.2. Fazit

Es konnte eine Softwareauswahl gefunden werden, mit welcher sich automatisierte Komponententests für AngularJS-Anwendungen im Commerzbank-Konzern realisieren lassen. Auch konnte durch die prototypische Testentwicklung für das Projekt Gefährdungsbeurteilungen gezeigt werden, dass diese Auswahl problemlos funktioniert, einfach anzuwenden ist und sich auch in bestehende Projekte integrieren lässt. Dadurch sollte die Eintrittsschwelle und der initiale Aufwand einer Einbindung in existierende und neue Projekte gesunken sein. Es ist nun zu hoffen, dass die ausgewählte Software und überhaupt die Idee automatisierter Tests Anklang findet und so sukzessive die Testabdeckung gesteigert und eine höhere Softwarequalität erreicht werden kann. Als fördernde Maßnahme ist hierzu eine Vorstellung der Ergebnisse dieser Bachelorthesis und eine Einführung in die grundlegenden Vorgehensweisen zur Entwicklung automatisierter Tests in AngularJS-Anwendungen für die Kolleginnen und Kollegen in der Commerz Systems GmbH durch den Autor angedacht.

Als letztlich kleiner Misserfolg kann die Untersuchung der Möglichkeiten der Durchführung automatisierter End-To-End-Tests angesehen werden. Sämtliche gefundenen und untersuchten Tools scheitern an den Anforderungen, welche sich durch regulatorisch und historisch bedingte Regelwerke, Prozesse und technische Systeme im Commerzbank-Konzern ergeben. Zukünftig soll versucht werden, diese Hindernisse zu beseitigen, zu ändern oder durch geeignete Maßnahmen zu überwinden, um die Durchführung automatisierter End-To-End-Tests zu ermöglichen und eine weitere Verbesserung der Effektivität und Qualität herbeizuführen.

Alles in Allem wurden die Ziele dieser Arbeit erfüllt und gute Grundlagen für die zukünftige Softwareentwicklung in der Commerz Systems GmbH und Commerzbank AG geschaffen.

Literaturverzeichnis

[Asu14] Asutosh. *An example of use of Protractor mit Mocha and Chai for an Angular applicaion*. 16. Feb. 2014. URL: http://litutech.blogspot.de/2014/02/an-example-of-use-of-protractor-with.html (besucht am 27. 05. 2017).

[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).

[ByV14] Albert ByVerdu. *e2e tests with protractor and mocha.* 16. Nov. 2014. URL: http://byverdu.github.io/e2e-tests-with-protractor-and-mocha/ (besucht am 28. 05. 2017).

[Cha17a] Chai. *Assert - Chai*. 2017. URL: http://chaijs.com/api/assert/ (besucht am 11. 05. 2017).

[Cha17b] Chai. *Assertion Styles - Chai*. 2017. URL: http://chaijs.com/guide/styles/ (besucht am 11. 05. 2017).

[Cha17c] Chai. *Chai Assertion Library*. 2017. URL: http://chaijs.com/ (besucht am 11. 05. 2017).

[Cha17d] Chai. *Installation - Chai*. 2017. URL: http://chaijs.com/guide/installation/ (besucht am 11. 05. 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).

[Com14] Commerzbank AG. »Gefährdungsbeurteilungen - Leistungsbeschreibung«. internes Dokument. 8. Juli 2014.

[Com16a] Commerz Systems GmbH. »MarisCV Entwicklerhandbuch«. internes Dokument. 2016.

[Com16b] 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.

[Com17c] Commerzbank AG. »Software Engineering Framework. Ergebnistyp: Testergebnisse«. internes Dokument. 27. Apr. 2017.

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

[Dur15] Amir Duran. *Why is CasperJS better than PhantomJS*. 28. Juli 2015. URL: http://code-epicenter.com/why-is-casperjs-better-than-phantomjs/ (besucht am 11. 05. 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).

[Git17a] GitHub. *Issues - ariya/phantomjs*. dynamisch generierte Auswertung. 25. Mai 2017. URL: https://github.com/ariya/phantomjs/issues (besucht am 25. 05. 2017).

[Git17b] GitHub. *Pulse - ariya/phantomjs*. dynamisch generierte Auswertung. 25. Mai 2017. URL: https://github.com/ariya/phantomjs/pulse (besucht am 25. 05. 2017).

[Goo17a] Google. *AngularJS: API: angular.mock.inject*. 2017. URL: https://docs.angularjs.org/api/ngMock/function/angular.mock.inject (besucht am 15. 05. 2017).

[Goo17b] Google. *AngularJS: API: angular.mock.module*. 2017. URL: https://docs.angularjs.org/api/ngMock/function/angular.mock.module (besucht am 15. 05. 2017).

[Goo17c] Google. *AngularJS: API: $compile*. 2017. URL: https://docs.angularjs.org/api/ng/service/$compile (besucht am 08. 06. 2017).

[Goo17d] Google. *AngularJS: API: $controller*. 2017. URL: https://docs.angularjs.org/api/ng/service/$controller (besucht am 08. 06. 2017).

[Goo17e] Google. *AngularJS: API: $filter*. 2017. URL: https://docs.angularjs.org/api/ng/service/$filter (besucht am 08. 06. 2017).

[Goo17f] Google. *AngularJS: API: ngMock*. 2017. URL: https://docs.angularjs.org/api/ngMock (besucht am 15. 05. 2017).

[Goo17g] Google. *AngularJS: Developer Guide: Controllers*. 2017. URL: https://docs.angularjs.org/guide/controller (besucht am 24. 04. 2017).

[Goo17h] Google. *AngularJS: Developer Guide: Dependency Injection*. 2017. URL: https://docs.angularjs.org/guide/di (besucht am 21. 04. 2017).

[Goo17i] Google. *AngularJS: Developer Guide: E2E Testing*. 2017. URL: https://docs.angularjs.org/guide/e2e-testing (besucht am 27. 05. 2017).

[Goo17j] Google. *AngularJS: Developer Guide: Scopes*. 2017. URL: https://docs.angularjs.org/guide/scope (besucht am 21. 04. 2017).

[Goo17k] Google. *AngularJS: Developer Guide: Services*. 2017. URL: https://docs.angularjs.org/guide/services (besucht am 20. 04. 2017).

[Goo17l] Google. *AngularJS: Miscellaneous: FAQ*. 2017. URL: https://docs.angularjs.org/misc/faq.

[Goo17m] Google. *Testing*. 2017. URL: https://angular.io/docs/ts/latest/guide/testing.html (besucht am 11. 06. 2017).

[Goo17n] 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).

[Ham+15] Sylvain Hamel u. a. *karma-ie-launcher*. 9. Juni 2015. URL: https://github.com/karma-runner/karma-ie-launcher (besucht am 02. 06. 2017).

[Hen+17] Anna Henningsen u. a. *Istanbul, a JavaScript test coverage tool*. 2017. URL: https://istanbul.js.org/ (besucht am 28. 05. 2017).

[Hen14] Mitch Hentges. *Kommentar in „IE doesn't work as of Protractor 1.1.0 unless settings have been fine-tuned"*. 30. Dez. 2014. URL: https://github.com/angular/protractor/issues/1248#issuecomment-68384845 (besucht am 09. 06. 2017).

[Hid17a] Ariya Hidayat. *FAQ | PhantomJS*. 2017. URL: http://phantomjs.org/faq.html (besucht am 10. 05. 2017).

[Hid17b] Ariya Hidayat. *Headless Testing | PhantomJS*. 2017. URL: http://phantomjs.org/headless-testing.html (besucht am 11. 05. 2017).

[Hid17c] Ariya Hidayat. *PhantomJS*. 2017. URL: http://phantomjs.org/ (besucht am 10. 05. 2017).

[Hid17d] Ariya Hidayat. *Quick Start | PhantomJS*. 2017. URL: http://phantomjs.org/quick-start.html (besucht am 11. 05. 2017).

[Hom16] Ruslan Homyak. *Large websites using AngularJS*. 11. Feb. 2016. URL: https://stfalcon.com/en/blog/post/large-websites-using-AngularJS (besucht am 12. 06. 2017).

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

[Int17] Intern. *Intern: The user guide*. 2017. URL: https://theintern.github.io/intern/ (besucht am 12. 05. 2017).

[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).

[Kro+15] Tanguy Krotoff u. a. *A Karma plugin. Generate code coverage.* 16. Nov. 2015. URL: https://github.com/karma-runner/karma-coverage (besucht am 28. 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).

[Ora13] Oracle. *Introduction to Facelets - The Java EE6 Tutorial*. 2013. URL: http://docs.oracle.com/javaee/6/tutorial/doc/giepx.html (besucht am 12. 06. 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).

[Per+17a] Nicolas Perriault u. a. *CasperJS, a navigation scripting and testing utilty for PhantomJS and SlimerJS*. 2017. URL: http://casperjs.org/ (besucht am 11. 05. 2017).

[Per+17b] Nicolas Perriault u. a. *FAQ - CasperJS 1.1.0-DEV documentation*. 2017. URL: http://docs.casperjs.org/en/latest/faq.html (besucht am 11. 05. 2017).

[Per+17c] Nicolas Perriault u. a. *Installation - CasperJS 1.1.0-DEV documentation*. 2017. URL: http://docs.casperjs.org/en/latest/installation.html (besucht am 11. 05. 2017).

[Per+17d] Nicolas Perriault u. a. *Testing - CasperJS 1.1.0-DEV documentation*. 2017. URL: http://docs.casperjs.org/en/latest/testing.html (besucht am 11. 05. 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).

[Sid16] Sid. *Beitrag in „Protractor test in IE“*. 26. Mai 2016. URL: https://stackoverflow.com/questions/37456099/protractor-test-in-ie (besucht am 09. 06. 2017).

[Sil+17] Filipe Silva u. a. *Angular CLI*. 6. Juni 2017. URL: https://github.com/angular/angular-cli/wiki (besucht am 11. 06. 2017).

[Sin17a] SinonJS. *Mocks - SinonJS*. 2017. URL: http://sinonjs.org/releases/v2.2.0/mocks/ (besucht am 14. 05. 2017).

[Sin17b] SinonJS. *Sandboxes - SinonJS*. 2017. URL: http://sinonjs.org/releases/v2.2.0/sandbox/ (besucht am 14. 05. 2017).

[Sin17c] SinonJS. *SinonJS - Standalone test spies, stubs and mocks for JavaScript. Works with any unit testing framework*. 2017. URL: http://sinonjs.org/ (besucht am 14. 05. 2017).

[Sin17d] SinonJS. *Spies - SinonJS*. 2017. URL: http://sinonjs.org/releases/v2.2.0/spies/ (besucht am 14. 05. 2017).

[Sin17e] SinonJS. *Stubs - SinonJS*. 2017. URL: http://sinonjs.org/releases/v2.2.0/stubs/.

[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).

[Sor+17] Sindre Sorhus u. a. *avajs/ava: Futuristic JavaScript test runner*. 6. Mai 2017. URL: https://github.com/avajs/ava (besucht am 14. 05. 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).

[Wat15] Jason Watmore. *Unit Testing in AngularJS with Mocha, Chai, Sinon & ngMock*. 9. Apr. 2015. URL: http://jasonwatmore.com/post/2015/04/09/unit-testing-in-angularjs-with-mocha-chai-sinon-ngmock (besucht am 15. 05. 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).

Glossar

Abkürzungen

BaFin Bundesanstalt für Finanzdienstleistungsaufsicht
DOM Document Object Model
HTTP Hypertext Transfer Protocol
JSON JavaScript Object Notation
MVVM Model-View-ViewModel, ein Entwurfsmuster
MVC Model-View-Controller, ein Entwurfsmuster
REST Representational State Transfer
SEF Software Engineering Framework (der Commerzbank)

Begriffe

falsy Ein in JavaScript als false angesehener Wert: false, 0, „“(leerer String), null, undefined und NaN.
jQuery jQuery ist eine JavaScript-Bibliothek für DOM-Manipulation, Event-Handling und Animation.
truthy Ein in JavaScript als true angesehener Wert: alle nicht falsy Werte, auch „0“, „false“ oder leere Funktionen, Arrays, Objekte.
WebSocket WebSocket ist eine bidirektionale Erweiterung von HTTP.
Whitebox-Test Ein Whitebox-Test ist ein Test, welcher unter Kenntniss der inneren Funktionsweise entwickelt wird.

Anhang

A. Konfigurationsdateien

```
{
  "name": "gfb",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "karma start"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "angular-mocks": "1.2.28",
    "chai": "^4.0.0",
    "karma": "^1.7.0",
    "karma-chai": "^0.1.0",
    "karma-coverage": "^1.1.1",
    "karma-firefox-launcher": "^1.0.1",
    "karma-ie-launcher": "^1.0.0",
    "karma-mocha": "^1.3.0",
    "karma-mocha-reporter": "^2.2.3",
    "karma-sinon": "^1.0.5",
    "mocha": "^3.4.2",
    "sinon": "^2.3.2",
    "jquery": "1.9.1"
  }
}
```

Listing 29: package.json für das GFB-Testprojekt

```
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['mocha', 'chai', 'sinon'],
    files: [
      'src/ext/jquery/jquery.js',
      'src/ext/jquery/*.js',
      'src/ext/angularjs/angular.min.js',
      'src/ext/angularjs/**/*.js',
      'src/ext/bootstrap/js/ui-bootstrap-tpls-0.12.0.min.js',
      'node_modules/angular-mocks/angular-mocks.js',
      'src/app.js',
      'src/ext/*.js',
      'src/*.js',
      'test/spec/**/*_spec.js'
```

```
    ],
    exclude: ['src/ext/angularjs/angular-sanitize.min.js', 'src/
    gruppenlogik*.js'],
    preprocessors: { 'src/ext/*.js': ['coverage'], 'src/*.js': ['coverage
    '] },
    reporters: ['mocha', 'coverage'],
    browsers: ['IE_no_addons'],
    concurrency: Infinity,
    autoWatch: true,
    singleRun: false,
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,

    customLaunchers: {
      IE_no_addons: {
        base: 'IE',
        flags: ['-extoff']
      }
    },
    coverageReporter: {
      dir: 'coverage/',
      reporters: [
        { type: 'html', subdir: 'html' },
        { type: 'lcov', subdir: 'lcov' }
      ]
    }
  })
}
```

Listing 30: karma.conf.js für das GFB-Testprojekt

```
exports.config = {
  framework: 'mocha',
  capabilities: {
    'browserName': 'internet explorer'
  },
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['test/e2e/**/*_spec.js'],
  mochaOpts: {
    enableTimeouts: false
  },
};
```

Listing 31: protractor.conf.js für das GFB-Testprojekt

B. Testsuites

```
describe("A test suite", function () {
    beforeEach(function () { });
    afterEach(function () { });
    it('should succeed', function () { expect(true).to.be.true; });
    it('should verify Sinon is working', function () {
        callback = sinon.spy();
        callCallback(callback);
        expect(callback.calledOnce).to.be.true;
    })
});

function callCallback(callback) {
    callback();
}
```

Listing 32: test/spec/test_spec.js

```
describe("A ngMock test suite", function () {
    var $filter;

    beforeEach(module("CBGFD"));
    beforeEach(inject(function (_$filter_) {
        $filter = _$filter_;
    }));

    it("should have a defined $filter factory", function () {
        expect($filter).to.exist;
    });

    it("should resolve a defined filter", function () {
        var unique = $filter("unique");
        expect(unique).to.exist;
    });

    it("should inject a service", inject(function (svc_ai) {
        expect(svc_ai).to.exist;
    }));
});
```

Listing 33: test/spec/ngTest_spec.js

```
describe("OverviewCtrl", function() {
    var $controller, $rootScope;
    var sandbox;

    beforeEach(module("CBGFD"));
    beforeEach(inject(function(_$controller_, _$rootScope_) {
        $controller = _$controller_;
```

```
        $rootScope = _$rootScope_;
    }));
    beforeEach(function() {
        sandbox = sinon.sandbox.create();
    });
    afterEach(function() {
        sandbox.restore();
    });


    describe("$scope.validatebegehungsdatum", function() {
        var initialData;
        beforeEach(inject(function(cls_gefahrenbeurteilung) {
            initialData = {initialGefaehrdung: new
cls_gefahrenbeurteilung()};
        }));

        it("should exist", inject(function() {
            var $scope = $rootScope.$new();
            var ctrl = $controller("OverviewCtrl", {$scope: $scope,
initialData: initialData});

            expect($scope.validatebegehungsdatum).to.be.defined;
        }));

        it("should return true given an undefined gefahrenbeurteilung",
function() {
            var $scope = $rootScope.$new();
            var ctrl = $controller("OverviewCtrl", {$scope: $scope,
initialData: initialData});
            $scope.gefahrenbeurteilung = undefined;

            expect($scope.validatebegehungsdatum()).to.be.true;
        });

        it("should return false given gfbTyp 'Begehung' and
begehungsdatum null", function() {
            var $scope = $rootScope.$new();
            var ctrl = $controller("OverviewCtrl", {$scope: $scope,
initialData: initialData});
            $scope.gefahrenbeurteilung = {mitarbeiterrolle:{gfbTyp:'
Begehung'}, begehungsdatum:null};

            expect($scope.validatebegehungsdatum()).to.be.false;
        });

        it("should return false given gfbTyp 'Begehung' and undefined
begehungsdatum", function() {
```

```
            var $scope = $rootScope.$new();
            var ctrl = $controller("OverviewCtrl", {$scope: $scope,
    initialData: initialData});
            $scope.gefahrenbeurteilung = {mitarbeiterrolle:{gfbTyp:'
    Begehung'}, begehungsdatum:undefined};

            expect($scope.validatebegehungsdatum()).to.be.false;
        });

        it("should return true given a gfbTyp other than 'Begehung'",
    function() {
            var $scope = $rootScope.$new();
            var ctrl = $controller("OverviewCtrl", {$scope: $scope,
    initialData: initialData});
            $scope.gefahrenbeurteilung = {mitarbeiterrolle:{gfbTyp:''}};

            expect($scope.validatebegehungsdatum()).to.be.true;
        });

        it("should return true given a not-undefend and not-null
    begehungsdatum", function() {
            var $scope = $rootScope.$new();
            var ctrl = $controller("OverviewCtrl", {$scope: $scope,
    initialData: initialData});
            $scope.gefahrenbeurteilung = {mitarbeiterrolle:{},
    begehungsdatum:''};

            expect($scope.validatebegehungsdatum()).to.be.true;
        });
    })
});
```

Listing 34: test/spec/overviewCtrl_spec.js

```
describe("cls_betriebsratsbereich", function() {
    var betriebsratsbereich;
    var sandbox;

    beforeEach(module("CBGFD"));
    beforeEach(inject(function(svc_sp, cls_betriebsratsbereich) {
        betriebsratsbereich = new cls_betriebsratsbereich(svc_sp);
    }));
    beforeEach(function() {
        sandbox = sinon.sandbox.create();
    });
    afterEach(function() {
        sandbox.restore();
    })

```

```
    it("should be initialised with correct values", inject(function(
svc_sp, cls_betriebsratsbereich) {
        expect(betriebsratsbereich.sp_id).to.equal(-1);
        expect(betriebsratsbereich.title).to.equal("");
        expect(betriebsratsbereich.sortierung).to.equal(0);
    }));

    describe("loadfromxmlitem", function() {
        it("should fill the fields with values from the xmlItem",
function() {
            var xmlItem = {attr: sinon.stub()};
            xmlItem.attr.withArgs("ows_ID").returns(42);
            xmlItem.attr.withArgs("ows_Title").returns("Test");
            xmlItem.attr.withArgs("ows_Sortierung").returns(1);
            betriebsratsbereich.loadfromxmlitem(xmlItem);

            expect(betriebsratsbereich.sp_id).to.equal(42);
            expect(betriebsratsbereich.title).to.equal("Test");
            expect(betriebsratsbereich.sortierung).to.equal(1);
        });

        it("should set field id to undefined given a xmlItem with missing
 attribute ows_ID", function() {
            var xmlItem = {attr: sinon.stub()};
            //xmlItem.attr.withArgs("ows_ID").returns(42);
            xmlItem.attr.withArgs("ows_Title").returns("Test");
            xmlItem.attr.withArgs("ows_Sortierung").returns(1);
            betriebsratsbereich.loadfromxmlitem(xmlItem);

            expect(betriebsratsbereich.sp_id).to.equal(undefined);
            expect(betriebsratsbereich.title).to.equal("Test");
            expect(betriebsratsbereich.sortierung).to.equal(1);
        });

        it("should leave field title blank given a xmlItem with missing
attribute ows_Title", function() {
            var xmlItem = {attr: sinon.stub()};
            xmlItem.attr.withArgs("ows_ID").returns(42);
            //xmlItem.attr.withArgs("ows_Title").returns("Test");
            xmlItem.attr.withArgs("ows_Sortierung").returns(1);
            betriebsratsbereich.loadfromxmlitem(xmlItem);

            expect(betriebsratsbereich.sp_id).to.equal(42);
            expect(betriebsratsbereich.title).to.equal("");
            expect(betriebsratsbereich.sortierung).to.equal(1);
        });
```

```
59          it("should set field sortierung to -1 blank given a xmlItem with
        missing attribute ows_Sortierung", function() {
                var xmlItem = {attr: sinon.stub()};
61              xmlItem.attr.withArgs("ows_ID").returns(42);
                xmlItem.attr.withArgs("ows_Title").returns("Test");
63              //xmlItem.attr.withArgs("ows_Sortierung").returns(1);
                betriebsratsbereich.loadfromxmlitem(xmlItem);
65
                expect(betriebsratsbereich.sp_id).to.equal(42);
67              expect(betriebsratsbereich.title).to.equal("Test");
                expect(betriebsratsbereich.sortierung).to.equal(-1);
69          });
        })
71  })
```

Listing 35: test/spec/svc_obs_spec.js

```
1   describe("dir_filHb", function () {
        var $compile, $rootScope;
3       var scope;
        var sinonSandbox;
5
        beforeEach(module("CBGFD"));
7       beforeEach(inject(function (_$compile_, _$rootScope_) {
            $compile = _$compile_;
9           $rootScope = _$rootScope_;
            scope = $rootScope.$new();
11          scope.filhb = '';
        }));
13      beforeEach(function () {
            sinonSandbox = sinon.sandbox.create();
15      });

17      afterEach(function () {
            sinonSandbox.restore();
19      })

21      it("should replace the element with an input and a button", function
        () {
            var el = getCompiledDirective();
23          scope.$digest();
            expect(el.html()).to.contain("input");
25          expect(el.html()).to.contain("button");
        });
27
        describe("given an empty value", function () {
29          it("should set input class to EingabefeldEinzeile and not alert",
         function () {
                var el = getCompiledDirective();
```

```
31
            var setAttributeSpy = sinonSandbox.spy();
33          var alertSpy = sinonSandbox.spy(window, "alert");
            var eventDummy = { srcElement: { value: "", setAttribute:
    setAttributeSpy } };
35
            el.isolateScope().myBlur(eventDummy);
37
            expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile")).to.be.true;
39          expect(setAttributeSpy.calledOnce).to.be.true;
            expect(alertSpy.called).to.be.false;
41      });
    });
43
    describe("given an valid value", function () {
45      it("should set input class to EingabefeldEinzeile and not alert",
     function () {
            var el = getCompiledDirective();
47
            var setAttributeSpy = sinonSandbox.spy();
49          var alertSpy = sinonSandbox.spy(window, "alert");
            var eventDummy = { srcElement: { value: "11111", setAttribute
    : setAttributeSpy } };
51
            el.isolateScope().myBlur(eventDummy);
53
            expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile")).to.be.true;
55          expect(setAttributeSpy.calledOnce).to.be.true;
            expect(alertSpy.called).to.be.false;
57      });

59      it("should set input class to EingabefeldEinzeile and not alert",
     function () {
            var el = getCompiledDirective();
61
            var setAttributeSpy = sinonSandbox.spy();
63          var alertSpy = sinonSandbox.spy(window, "alert");
            var eventDummy = { srcElement: { value: "99999", setAttribute
    : setAttributeSpy } };
65
            el.isolateScope().myBlur(eventDummy);
67
            expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile")).to.be.true;
69          expect(setAttributeSpy.calledOnce).to.be.true;
            expect(alertSpy.called).to.be.false;
```

```
71          });
73          it("should set input class to EingabefeldEinzeile and not alert",
     function () {
                var el = getCompiledDirective();
75
                var setAttributeSpy = sinonSandbox.spy();
77              var alertSpy = sinonSandbox.spy(window, "alert");
                var eventDummy = { srcElement: { value: "00000", setAttribute
    : setAttributeSpy } };
79
                el.isolateScope().myBlur(eventDummy);
81
                expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile")).to.be.true;
83              expect(setAttributeSpy.calledOnce).to.be.true;
                expect(alertSpy.called).to.be.false;
85          });
     });
87
     describe("given a not numerical value", function () {
89          it("should set input class to EingabefeldEinzeile,ng-invalid-
    input and alert", function () {
                var el = getCompiledDirective();
91
                var setAttributeSpy = sinonSandbox.spy();
93              var alertSpy = sinonSandbox.spy(window, "alert");
                var eventDummy = { srcElement: { value: "abc", setAttribute:
    setAttributeSpy } };
95
                el.isolateScope().myBlur(eventDummy);
97
                expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile")).to.be.true;
99              expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile ng-invalid-input")).to.be.true;
                expect(alertSpy.called).to.be.true;
101         });
     });
103
     describe("given a wrong formatted numerical value", function () {
105         it("should set input class to EingabefeldEinzeile,ng-invalid-
    input and alert (value too long)", function () {
                var el = getCompiledDirective();
107
                var setAttributeSpy = sinonSandbox.spy();
109             var alertSpy = sinonSandbox.spy(window, "alert");
```

```
            var eventDummy = { srcElement: { value: "123456",
    setAttribute: setAttributeSpy } };
111
            el.isolateScope().myBlur(eventDummy);
113
            expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile")).to.be.true;
115         expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile ng-invalid-input")).to.be.true;
            expect(alertSpy.called).to.be.true;
117     });

119     it("should set input class to EingabefeldEinzeile,ng-invalid-
    input and alert (value too short)", function () {
            var el = getCompiledDirective();
121
            var setAttributeSpy = sinonSandbox.spy();
123         var alertSpy = sinonSandbox.spy(window, "alert");
            var eventDummy = { srcElement: { value: "1234", setAttribute:
     setAttributeSpy } };
125
            el.isolateScope().myBlur(eventDummy);
127
            expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile")).to.be.true;
129         expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile ng-invalid-input")).to.be.true;
            expect(alertSpy.called).to.be.true;
131     });

133     it("should set input class to EingabefeldEinzeile,ng-invalid-
    input and alert (value negative)", function () {
            var el = getCompiledDirective();
135
            var setAttributeSpy = sinonSandbox.spy();
137         var alertSpy = sinonSandbox.spy(window, "alert");
            var eventDummy = { srcElement: { value: "-12345",
    setAttribute: setAttributeSpy } };
139
            el.isolateScope().myBlur(eventDummy);
141
            expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile")).to.be.true;
143         expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile ng-invalid-input")).to.be.true;
            expect(alertSpy.called).to.be.true;
145     });
```

```
147         it("should set input class to EingabefeldEinzeile,ng-invalid-
    input and alert (value is decimal)", function () {
                var el = getCompiledDirective();
149
                var setAttributeSpy = sinonSandbox.spy();
151             var alertSpy = sinonSandbox.spy(window, "alert");
                var eventDummy = { srcElement: { value: "123.4", setAttribute
    : setAttributeSpy } };
153
                el.isolateScope().myBlur(eventDummy);
155
                expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile")).to.be.true;
157             expect(setAttributeSpy.calledWith("class", "
    EingabefeldEinzeile ng-invalid-input")).to.be.true;
                expect(alertSpy.called).to.be.true;
159         });
    });
161


163 function getCompiledDirective() {
        var el = $compile('<filhb myfilhb="filHb" my-tooltip-input="TT"
    my-tooltip-button="TT2"></filhb>')(scope);
165     return el;
    }
167 });
```

Listing 36: test/spec/dir_filHb_spec.js

```
1 describe("uniqueFilter", function () {
    var uniqueFilter;
3
    beforeEach(module("CBGFD"));
5   beforeEach(inject(function ($filter) {
        uniqueFilter = $filter("unique");
7   }));

9   it("should return an empty array given a null input-array", function
    () {
        expect(uniqueFilter(null, 0)).to.deep.equal([]);
11  });

13  it("should return an empty array given an empty input-array",
    function () {
        expect(uniqueFilter([], 0)).to.deep.equal([]);
15  });

17  it("should return the same array given an array with only 1 entry",
    function () {
```

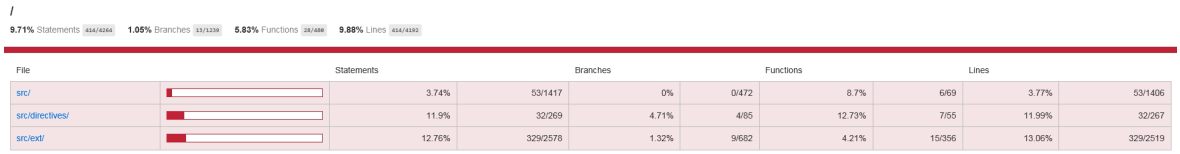
```
        expect(uniqueFilter([[42]], 0)).to.deep.equal([[42]]);
    });

    it("should return the same array given an array with 2 different
    entries", function () {
        expect(uniqueFilter([[42], [1]], 0)).to.deep.equal([[42], [1]]);
    });

    it("should remove the doubled value given an array with 2 matching
    entries", function () {
        expect(uniqueFilter([[42], [1], [42]], 0)).to.deep.equal([[42],
    [1]]);
    });

    it("should remove the doubled value given an array with 2 different
    entries but the key is the same", function () {
        expect(uniqueFilter([[42, 1], [1, 1], [42, 2]], 0)).to.deep.equal
    ([[42, 1], [1, 1]]);
    })
});
```

Listing 37: test/spec/uniqueFilter_spec.js

```
var chai = require('chai');
var chaiAsPromised = require('chai-as-promised');
chai.use(chaiAsPromised);
var expect = chai.expect;

describe('angularjs homepage todo list', function() {
  it('should add a todo', function() {
    browser.get('https://angularjs.org');

    element(by.model('todoList.todoText')).sendKeys('write first
    protractor test');
    element(by.css('[value="add"]')).click();

    var todoList = element.all(by.repeater('todo in todoList.todos'));
    expect(todoList.count()).to.eventually.equal(3);
    expect(todoList.get(2).getText()).to.eventually.equal('write first
    protractor test');

    // You wrote your first test, cross it off the list
    todoList.get(2).element(by.css('input')).click();
    var completedAmount = element.all(by.css('.done-true'));
    expect(completedAmount.count()).to.eventually.equal(2);
  });
});
```

Listing 38: test/e2e/test_spec.js

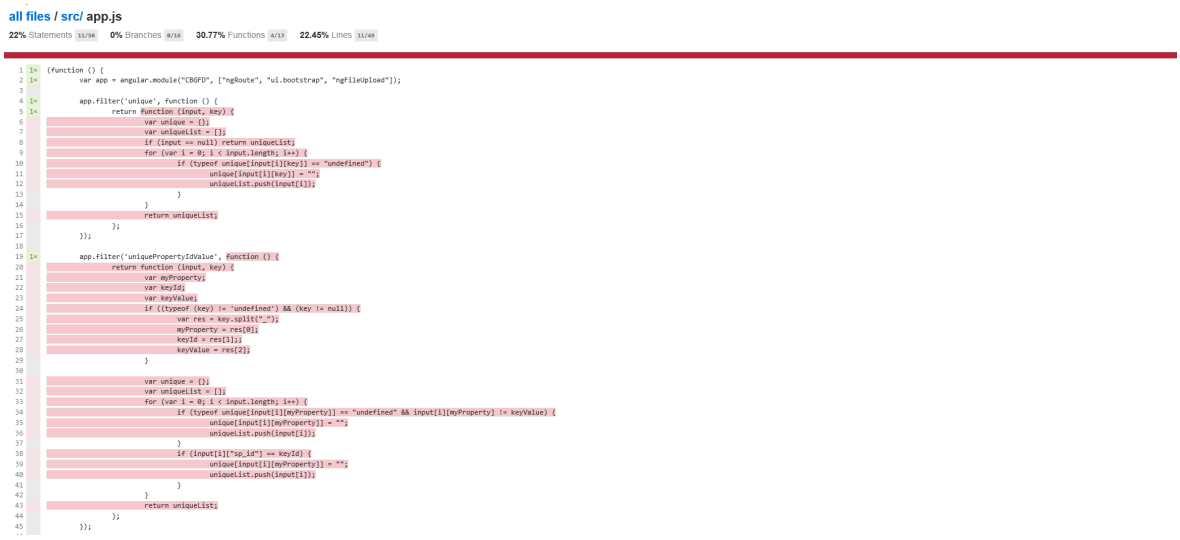
C. Sonstiges


/

9.71% Statements 414/4264 **1.05%** Branches 13/1230 **5.83%** Functions 28/480 **9.88%** Lines 414/4192

File	Statements		Branches		Functions		Lines	
src/	3.74%	53/1417	0%	0/472	8.7%	6/69	3.77%	53/1406
src/directives/	11.9%	32/269	4.71%	4/85	12.73%	7/55	11.99%	32/267
src/ext/	12.76%	329/2578	1.32%	9/682	4.21%	15/356	13.06%	329/2519

Abbildung 1: Übersichtsseite des Code-Coverage-Berichts im Browser


all files / src/ app.js

22% Statements 11/50 **0%** Branches 0/14 **30.77%** Functions 4/13 **22.45%** Lines 11/49

```
1  1x  (function () {
2  1x          var app = angular.module("CBGFD", ["ngRoute", "ui.bootstrap", "ngFileUpload"]);
3
4  1x          app.filter('unique', function () {
5  1x                  return function (input, key) {
6                              var unique = {};
7                              var uniqueList = [];
8                              if (input == null) return uniqueList;
9                              for (var i = 0; i < input.length; i++) {
10                                     if (typeof unique[input[i][key]] == "undefined") {
11                                             unique[input[i][key]] = "";
12                                             uniqueList.push(input[i]);
13                                     }
14                             }
15                             return uniqueList;
16                 };
17         });
18
19 1x         app.filter('uniquePropertyIdValue', function () {
20                 return function (input, key) {
21                             var myProperty;
22                             var keyId;
23                             var keyValue;
24                             if ((typeof (key) != 'undefined') && (key != null)) {
25                                     var res = key.split("_");
26                                     myProperty = res[0];
27                                     keyId = res[1];
28                                     keyValue = res[2];
29                             }
30
31                             var unique = {};
32                             var uniqueList = [];
33                             for (var i = 0; i < input.length; i++) {
34                                     if (typeof unique[input[i][myProperty]] == "undefined" && input[i][myProperty] != keyValue) {
35                                             unique[input[i][myProperty]] = "";
36                                             uniqueList.push(input[i]);
37                                     }
38                                     if (input[i]["sp_id"] == keyId) {
39                                             unique[input[i][myProperty]] = "";
40                                             uniqueList.push(input[i]);
41                                     }
42                             }
43                             return uniqueList;
44                 };
45         });
```

Abbildung 2: Detailseite des Code-Coverage-Berichts im Browser (Der abgebildete Quelltext ist Teil des GFB-Projekts und damit Eigentum der Commerz Systems GmbH)

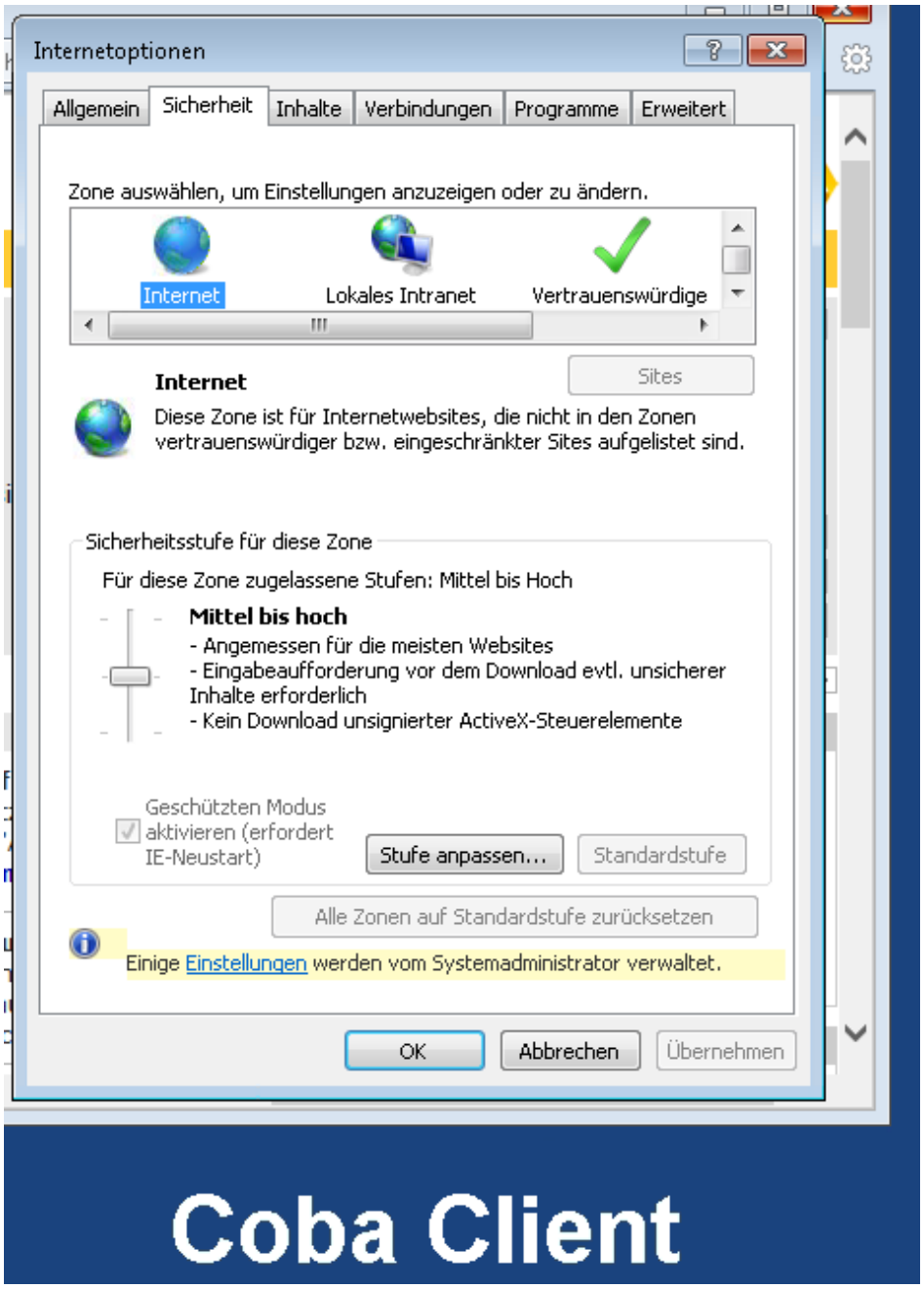


Abbildung 3: Einstellungsdialog zur Deakitiverung des Geschützten Modus' an einem Commerzbank-Arbeitsplatz

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.

$Unterschrift$: $Ort, Datum$: