



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:	21. April 2017
Erstprüfer	Prof. Dr.-Ing. Heide-Rose Vatterrott
Zweitprüfer	Dipl.-Inf. Jochen Schwitzing

Sperrvermerk

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

Lorem ipsum dolorem sit amet.

NIKOLAS SCHRECK

Abstract

In dieser Bachelor-Thesi

Inhaltsverzeichnis

Abbildungsverzeichnis	7
Tabellenverzeichnis	7
Listingverzeichnis	8
1 Motivation	9
2 Grundlagen	10
2.1 Test	10
2.1.1 Komponententest	10
2.1.2 Integrationstest	11
2.1.3 Systemtest	12
2.2 AngularJS	13
2.2.1 Architektur	13
2.2.2 Two-Way-Databinding	14
2.2.3 Scopes	15
2.2.4 Dependency Injection	15
2.3 Node.js	17
2.3.1 Laufzeitumgebung	17
2.3.2 npm	17
3 Systeminformationen	20
4 Rahmenbedingungen	21
5 Durchführung	22
6 Ergebnisse	23
7 Abschluss	24
Literaturverzeichnis	25
Stichwortverzeichnis	27
Anhang	28
Eidesstattliche Erklärung	28

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1	Beispiel eines AngularJS-Templates, adaptiert nach [BT14]	14
2	Beispielhafter AngularJS-Controller, adaptiert nach []	14
3	Beispielhafter AngularJS-Controller mit Dependency Injection, adaptiert nach [Goo17a]	16
4	Beispiel einer package.json	18
5	Beispiel von Skripten in einer package.json (aus [Cir14])	19
6	Beispiel der Watch-Funktionalität in einer package.json (aus [Cir14], angepasst durch den Autor)	19

1 Motivation

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

Quelle!

Viele Unternehmen nutzen bereits seit einigen Jahren Angular als Technologiebasis für clientseitige Webanwendungen, beispielsweise *Gmail*, *PayPal* oder *Youtube*. In 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.

quellen

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 Bachelorthesisrelevant sind. Der Abnahmetest wird 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 sollten nur die komponenteninterne Funktionsweise getestet werden[SL12, S. 45]. Beim Test von AngularJS-Anwendungen sind die Komponenten beispielsweise Controller, Services .

Ein Komponententest hat spezifische Testziele. Das Wichtigste ist die Sicherstellung, dass die Funktion der Forderung 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,

welche
noch,
Ref,
und
Cite

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

do the
ref

2.1.2 Integrationstest

Der Integrationstest folgt nach den Komponententests und setzt getestete Komponenten voraus. Diese Komponenten werden zu größeren Komponenten zusammengesetzt. Der Integrationstest dient dann dazu, zu überprüfen ob alle Einzelteile korrekt zusammenarbeiten und soll Fehler in Schnittstellen und im Zusammenspiel aufdecken. [SL12, S. 52 f.] Beispielsweise kann in einem Integrationstest auch die Anbindung an externe Komponenten, wie Datenbanken oder REST-APIs überprüft werden. Diese werden im Komponententest durch sogenannte *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 hiermit Nachteile verbunden sind. Es ist schwer bis unmöglich, die tatsächliche Fehlerursache herauszufinden, da oft nicht klar ist in welcher Teilkomponente der Fehler aufgetreten ist, sondern dieser sich nur in einem abweichenden Gesamtverhalten äußert. Manche Fehler werden möglicherweise gar nicht gefunden, da regelmäßig kein vollumfänglicher Zugriff auf Einzelkomponenten besteht.[SL12, S. 57]

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

- Bei der Top-Down-Integration beginnt der Test mit der obersten Systemkomponente, von der alle Anderen aufgerufen werden. Sukzessive werden die weiteren Komponenten von oben nach unten integriert und getestet, wobei die untergeordneten Komponenten zunächst durch Platzhalter ersetzt werden. Vorteilhaft ist, dass keine aufwändigen Testtreiber zum Aufruf benötigt werden. Jedoch müssen die 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 übergeordneten, aufrufenden Komponenten emulieren.
- Bei der Ad-hoc-Integration werden Komponenten integriert, sobald sie fertiggestellt sind. Nachteilig hierbei ist, dass sowohl Platzhalter als auch

Testtreiber implementiert werden müssen. Allerdings bietet sich ein Zeitgewinn, da jede Komponente so früh wie möglich integriert wird.

- Bei der wenig empfehlenswerten Big-Bang-Integration werden alle Komponenten auf einmal integriert. Sie bietet ausschließlich Nachteile: Es wird Zeit verschwendet, da bis zur Fertigstellung der letzten Komponente gewartet wird. Auch treten 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 sollten keine Testtreiber oder Platzhalter mehr vorhanden sein, sondern überall die finale Hard- und Software genutzt werden. Das Testziel ist die Validierung, ob funktionale und nicht-funktionale Anforderungen erfüllt werden.[SL12, S. 60 ff.]

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

über
Back-
bone
schrei-
ben?

2.2 AngularJS

AngularJS ist ein von Google Inc. ins Leben gerufenes JavaScript-Framework zur Entwicklung von clientseitigen Webanwendungen[Goo17d]. Der Quellcode von AngularJS steht auf Github zur Verfügung und wird dort auch von einer großen Entwicklergemeinschaft weiterentwickelt[Ler13, S. 9]. Da er unter der MIT-Lizenz veröffentlicht ist eignet sich AngularJS auch für den kommerziellen Einsatz[Ler13, S. 9; Pat16].

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

2.2.1 Architektur

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

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

Es ist erforderlich, die über die Model-Schicht ermittelten Daten zu verwalten und gegebenenfalls zu transformieren um sie der Anzeige zur Verfügung zu stellen. Hierfür wird die ViewModel-Schicht genutzt. Außerdem wird in dieser Schicht die Funktionalität definiert, welche die View-Schicht steuert und dieser die 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 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.]Diese Direktiven ermöglichen es wiederverwendbare Komponenten zu erschaffen, indem Template und Quelltext in einem neuen Tag oder Attribut gekapselt werden[BT14, S. 49 f.].

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

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

9  <script src="js/angular.js" />
</body>
11 </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 [ref](#)) 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).

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

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

5  $scope.setName = function() {
7    $scope.someModelName = "Neue" + $scope.someModelName;
  }
9 });
```

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

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 diesem 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.; Goo17b]

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

2.2.4 Dependency Injection

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

```
1 someModule.controller("MyController", ["$scope", "$http", "dep", function  
    ($scope, $http, dep) {  
2     $scope.aMethod = function() {  
3         dep.someFunction();  
4         // ...  
5     }  
    });
```

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

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

2.3 Node.js

2.3.1 Laufzeitumgebung

Node.js ist eine Laufzeitumgebung, mit der JavaScript ohne Webbrowser ausgeführt werden kann[HWD12, S. 1]. Somit ist es möglich, JavaScript nicht nur für die Darstellung von Benutzeroberflächen im Webbrowser zur 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[Goo17e]. ECMA-262 ist der im Juni 2016 veröffentlichte und zurzeit aktuellste JavaScript-Standard[Ecm16]. Somit bietet Node.js alle spezifizierten und von Google Chrome unterstützten Sprachfunktionalitäten. Es eignet sich daher auch für den Test von für Webbrowser entwickelte Webanwendungen.

Unterschied
Node.js
<-> JS

2.3.2 npm

Der Node Package Manager (npm) ist der zusammen mit Node.js verteilte Paketmanager für JavaScript. Unter npm wird außerdem die *npm Registry*, also die zentrale Ablage für JavaScript-Pakete, verstanden, auf welche der Node Package Manager zugreift. [npm17a] Die npm Registry enthält über 180.000 Pakete und ist damit das größte 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. [npm17b] 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.

Quelle

```
{
  "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) 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).

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

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

Wo
kommt
die
Quelle
hin?

Referen
Quelle

```
2  "scripts": {  
4    "lint": "jshint *.js",  
    "build": "browserify index.js > myproject.min.js",  
    "test": "mocha test/",  
6    "prepublish": "npm run build # also runs npm run prebuild",  
    "prebuild": "npm run test # also runs npm run pretest",  
8    "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 Pakets. 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]

```
1  "scripts": {  
    "lint": "jshint *.js",  
3    "build:js": "browserify assets/scripts/main.js > dist/main.js",  
    "build": "npm run build:js",  
5    "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 Systeminformationen

4 Rahmenbedingungen

5 Durchführung

6 Ergebnisse

7 Abschluss

Literaturverzeichnis

- [BT14] Robin Böhm und Philipp Tarasiewicz. *AngularJS: Eine praktische Einführung in das JavaScript-Framework*. 27. Mai 2014.
- [Bun16] Bundesanstalt für Finanzdienstleistungsaufsicht. *BaFin - Banken & Finanzdienstleister*. 22. März 2016. URL: https://www.bafin.de/DE/Aufsicht/BankenFinanzdienstleister/bankenfinanzdienstleister_node.html (besucht am 02.04.2017).
- [Cir14] Keith Cirkel. *How to Use npm as a Build Tool*. 9. Dez. 2014. URL: <https://www.keithcirkel.co.uk/how-to-use-npm-as-a-build-tool/> (besucht am 10.04.2017).
- [Coma] Commerzbank AG. »Allgemeine Programmierrichtlinien«. internes Dokument.
- [Comb] Commerzbank AG. »Book of Standards«. internes Dokument.
- [Comc] Commerzbank AG. »Programmierrichtlinien JavaScript«. internes Dokument.
- [DeB17] Erik DeBill. *Modulecounts*. 9. Apr. 2017. URL: <http://www.modulecounts.com> (besucht am 09.04.2017).
- [Ecm16] Ecma International. *Standard ECMA-262. ECMAScript 2016 Language Specification*. Version 7. Juni 2016. URL: <https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf> (besucht am 06.04.2017).
- [Goo17a] Google. *AngularJS: Developer Guide: Dependency Injection*. 2017. URL: <https://docs.angularjs.org/guide/di> (besucht am 21.04.2017).
- [Goo17b] Google. *AngularJS: Developer Guide: Scopes*. 2017. URL: <https://docs.angularjs.org/guide/scope> (besucht am 21.04.2017).
- [Goo17c] Google. *AngularJS: Developer Guide: Services*. 2017. URL: <https://docs.angularjs.org/guide/services> (besucht am 20.04.2017).
- [Goo17d] Google. *AngularJS: Miscellaneous: FAQ*. 2017. URL: <https://docs.angularjs.org/misc/faq>.
- [Goo17e] Google Developers. *Chrome V8*. 2017. URL: <https://developers.google.com/v8/> (besucht am 06.04.2017).
- [Gor17] Konstantin Gorodinskii. *concat*. 2017. URL: <https://www.npmjs.com/package/concat> (besucht am 21.04.2017).
- [Hal+17] James Halliday u. a. *browserify*. 5. Apr. 2017. URL: <https://github.com/substack/node-browserify/blob/master/readme.markdown> (besucht am 10.04.2017).
- [HWD12] T Hughes-Croucher, M Wilson und T Demmig. *Einführung in Node.js*. O'Reilly, 2012. ISBN: 9783868997972.
- [Ler13] Ari Lerner. *ng-book. The Complete Book on AngularJS*. 2013. ISBN: 978-0-9913446-0-4.
- [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, Inc. *npm*. 2017. URL: <https://www.npmjs.com/about> (besucht am 08.04.2017).
- [npm17b] npm, Inc. *Using a package.json*. 9. März 2017. URL: <https://docs.npmjs.com/getting-started/using-a-package.json> (besucht am 09.04.2017).
- [oos06] oose Innovative Informatik eG. *A-256 Testkonzept erstellen*. 6. Nov. 2006. URL: https://www.oose.de/oep/desc/a_824d.htm?tid=256 (besucht am 17.04.2017).
- [Pat16] Bill Patrianakos. *Why I Use the MIT License*. 28. Juli 2016. URL: <http://billpatrianakos.me/blog/2016/07/28/why-i-use-the-mit-license/> (besucht am 19.04.2017).
- [Pre16] Pascal Precht. *Angular 2 Is Out - Get Started Here*. 18. Dez. 2016. URL: <https://blog.thoughttram.io/angular/2016/09/15/angular-2-final-is-out.html> (besucht am 19.04.2017).
- [Roi05] Erich H. Peter Roitzsch. *Analytische Softwarequalitätssicherung in Theorie und Praxis*. MV-Verlag, 2005. ISBN: 9783865822024.
- [SL12] Andreas Spillner und Tilo Linz. *Basiswissen Softwaretest*. Dpunkt.Verlag GmbH, 11. Sep. 2012. ISBN: 3864900247. URL: http://www.ebook.de/de/product/19361935/andreas_spillner_tilo_linz_basiswissen_softwaretest.html.
- [Sof10] Software-Sanierung. *Warum End-To-End-Tests alleine mehr schaden als nützen*. 28. Feb. 2010. URL: <https://softwaresanierung.wordpress.com/2010/02/28/warum-end-to-end-tests-alleine-mehr-schaden-als-nutzen/> (besucht am 17.04.2017).
- [Sta14] Ilias Stampoulis. »Börse Frankfurt: Commerzbank schießt in die Höhe«. In: *Handelsblatt* (6. Juni 2014).
- [Sym] Symetics GmbH. *AngularJS.DE -> Dirty-Checking / Updatezyklus*. URL: <https://angularjs.de/buecher/angularjs-buch/dirty-checking> (besucht am 21.04.2017).
- [Wal+17] Rick Waldron u. a. *About JSHint*. 28. Jan. 2017. URL: <http://jshint.com/about/> (besucht am 10.04.2017).
- [Zae12] Jörn Zaefferer. *Introduction To JavaScript Unit Testing. How To Build A Testing Framework*. 27. Juni 2012. URL: <https://www.smashingmagazine.com/2012/06/introduction-to-javascript-unit-testing/> (besucht am 15.04.2017).

Abkürzungsverzeichnis

JSON JavaScript Object Notation

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 :

