Fullstack JavaScript Testing mit Jasmine, Karma & Co.

Christopher Konze
Universität Kassel
c.konze@uni.de

ABSTRACT

JavaScript ist eine Programmiersprache die sich an einigen Stellen nicht intuitive verhält [22]. Daher ist es essenziell JavaScript-Anwendungen auf korrektes Verhalten zu prüfen. Diese Seminararbeit gibt einen Einblick in die Frameworks, die benötigt werden, um solide JavaScript-Anwendung entwickeln zu können. Dabei wird zunächst das Code-Quality-Tool JSHint[19] vorgestellt. Anschließend wird gezeigt, wie mit Hilfe von Jasmine[1] und Karma[2] sowohl Unit-Tests als auch Integrationstests erstellt und in den verschrienen Browsern ausgeführt werden können. Für End-to-end bzw. Selenium-Tests wird WebDriver[3] vorgestellt. Zudem wird veranschaulicht, wie einige der gezeigten Frameworks, mittels Grunt[4] automatisiert in den Entwicklungsprozess eingebettet werden können.

1. EINLEITUNG

Jahrelang wurde JavaScript ehr *stiefmütterlich* behandelt. Doch mit dem Aufkommen von professionellen JavaScript-Frameworks wie Cordova [5], AngularJs [6] und NodeJs [7] hat sich auch die Entwickelung von JavaScript-Anwendungen professionalisiert. Ein essenzieller Bestandteil der professionellen Entwickelung von Software, sind automatisierte Tests [20, 21].

Diese Seminararbeit zeigt eine Möglichkeit zum umfassenden und automatisierten Testen von JavaScript Programmen. Der Fokus der Arbeit wird dabei auf eine einfache und kompakte Darstellung der Testverfahren gelegt. Daher wird davon ausgegangen, dass Interessenten am das Testen von JavaScript-Anwendungen bereits über grundlegende JavaScript Kenntnisse vorfügen.

Im nächsten Abschnitt 2 wird verdeutlicht, warum Tests in JavaScript besonders wichtig sind. Im Anschluss wird in Abschnitt 3 das Automatisierungstool Grunt vorgestellt. Mit Hilfe dieses Tools werden andere, in dieser Seminararbeit vorgestellte, Frameworks in den Entwicklungsprozess eingebunden werden. Abschnitt 4 beschäftigt sich mit der statischen Code Analyse von JavaScript Programmen. Hierzu wird das Framework JSHint verwendet. Danach wird im 5. Abschnitt das Jasmine Framework zum Erstellen von Tests vorgestellt und in Abschnitt 6 ein Tool zum Ausführen der Tests. In Abschnitt 7 wird eine Möglichkeit zur Erstellung von *End-to-end*-Tests vorgestellt. Abschließend wird in Abschnitt 8 ein kurzes Fazit gezogen.

2. MOTIVATION

Innerhalb dieses Abschnittes wird die Notwendigkeit von Tests, im Speziellen für JavaScript, hervorgehoben. Dazu werden einige Beispiele gezeigt, die verdeutlichen sollen, dass sich JavaScript an vielen Stellen anders verhält, als ein Entwickler dies aus anderen Programmiersprachen gewöhnt ist.

Ein Beispiel für ungewohntes Verhalten, zeigt sich beim Vergleich mit dem *NaN*-Wert (Not a Number). Dieses ist im Codebeispiel 1 zu sehen. In Zeile 1 und 2 wird ein String mit der Funktion *parseInt* in eine Zahl umgewandelt. Wenn der String keine Zahl gibt die Funktion *NaN* zurück. Intuitive sollte also Zeile 1 wahr sein. Allerdings wird – wie Zeile 3 zeigt – den *NaN*-Wert von dem Vergleichsoperator nicht korrekt behandelt. Um zu testen ob ein Wert *NaN* ist, muss die *isNaN*-Funktion verwendet werden. Zeile 4 zeigt zudem eine falsche Verwendung des *Negation*-Operators, daher ist der Vergleich hier wieder *false*.

Textauszug 1 Vergleich mit NaN

```
1 parseInt("no numbers here")==NaN //false
2 parseInt("1") == NaN //false
3 NaN == NaN //false
4 NaN == !NaN //false
```

Ein weiter häufiger Fehler tritt beim Iterieren über Array Objekte auf. Der naive Ansatz ist in Textauszug 2 gezeigt. Wie zu sehen ist, werden die Wert 0, 1, 2 und *sum* ausgegeben. Dies ist darin begründet, dass der *for-in-loop* nicht über die Arrayelemente selbst, sondern über die Indizes iteriert. Weiterhin wird auch über alle Funktionen des Objektes iteriert.

Textauszug 2 Falscher ForIn-Loop

```
1 var someNumbers = [1,2,3];
2 someNumbers.sum = function(){ return 6;};
3 for(var item in someNumbers) {
4   console.log(item);
5 }
6 // Prints: 0 1 2 sum
```

Eine Korrekte Lösung muss daher in jeder Iteration testen, ob es sich um eine Eigenschaft oder um eine Funktion handelt. Dies ist in Textauszug 3 verdeutlicht.

Textauszug 3 Korrekter ForIn-Loop

```
1 for(var index in someNumbers) {
2   if(someNumbers.hasOwnProperty(index)){
3     var item = someNumbers[index];
4     console.log(item);
5   }
6 }
7 //Prints: 1 2 3
```

Das Codebeispiel 4 zeigt, weitere Sprachkonstrukte die unerwartet Ergebnisse liefern. In Zeile 1 wird getestet, ob -1 kleiner als der minimale *Number* Wert. Da *Number* nur positive Zahlen enthält ($Number.Min_Value = 5e^{-324}$), wird dies als *true* angesehen. In der zweiten Zeile wird versucht den Typen eines Strings festzustellen. Allerdings schlägt dies fehl, da er falsche Operator verwendet wird. In JavaScript gibt der *instanceof*-Operator nur den Type von Objekten, die mittels *new* erzeugt wurden. (z.B.: new String(„i'm a String“)). Die 3. Zeile zeigt, den Versuch eine Array von Zahlen zu sortieren. Hierfür ist – wider Erwartung – die Sort-funktion nicht geeignet, da sie dafür ausgelegt ist nur Strings zu sortieren. Somit werden alle Einträge als Strings sortiert.

Textauszug 4 Weitere Beispiele

```
1 -1 < Number.MIN_VALUE //true
2 "I am a String" instanceof String //false
3  [5, 12, 9, 2, 18, 1, 25].sort();
4   //[1, 12, 18, 2, 25, 5, 9]
```

Die Liste von JavaScript Konstrukten, die sich wider der Erwartung verhalten, lässt sich belibig fortsetzen. Weitere Beispiel sind in [22] und im Anhang von [18] gegeben. Aus den o.g Beispielen sollte hervorgehen, dass das Verhalten von JavaScript von den Erwartungen abweichen kann. Daher sind Tests die einzige Möglichkeit, eine Anwendung gegen solche Fehler abzusichern.

3. GRUNT

Das Einbinden der verschiedene Testsmethodiken wird innerhalb dieser Seminararbeit mittels Grunt[4] durchgeführt. Im Folgenden wird daher, eine kurze Einleitung in das Tool gegeben und essenzielle Plugins vorgestellt. Grunt ist ein JavaScript *Task Runner*, der häufige und wiederkehrende Aufgaben und Prozesse (Task) automatisieren kann. Mit Hilfe von Grunt wird das Erstellen einer JavaScript-Anwendung deutlich vereinfacht. Grunt selbst dient nur zum Konfigurieren der Tasks. Die eigentliche Logik zur Abarbeitung eines Tasks wird durch Plugins realisiert. Diese Plugins werden großteils von der Community selbst erstellt. Aus diesem Grund existieren mittlerweile eine vielzahl von Plugins.

3.1 Installation und Konfiguration

Da Grunt eine Node.js Anwendung ist, muss zunächst NodeJs installiert werden [7]. Anschließend kann über den Node Package Manager Grunt global installiert werden. Dies geschieht mit dem Konsolenkommando in Testauszug 5.

Textauszug 5 Grunt installieren

```
npm install -g grunt-cli
```

Um nun Grunt in einem Projekt nutzen zu können, müssen zwei Dateien erstellet werden. Zum einen eine *Package.Json* (siehe Textauszug 6), in der Metainformationen zum Projekt und benötigte Grunt Plugins bzw. Abhängigkeiten angegeben werden.

Textauszug 6 package.json

```
1 {
2   "name": "my-project-name",
3   "version": "0.1.0",
4   "devDependencies": {
5     "grunt": "~0.4.5",
6     "grunt-contrib-uglify": "~0.5.0"
7   }
8 }
```

Zum anderen eine Datei mit dem Namen *Gruntfile.js* (Gruntfile). Diese Datei enthält die Konfigurationen der einzelnen Plugins und spezifiziert die ausführbaren Grunttasks. Ein Beispiel einer Gruntfile ist in Textauszug 7 gegeben.

Textauszug 7 Gruntfile

```
1 module.exports = function(grunt) {
2   //load plugins
3   grunt.loadNpmTasks("grunt-contrib-watch
        ");
4
5   //config
6   var config = {};
7   config["watch"] = {
8     scripts: {
9       files: "**/*.js",
10      tasks: ["dev_tests"]
11    }
12   };
13   grunt.initConfig(config);
14
15   //register tasks
16   grunt.registerTask("dev_tests", ["
        watch"]);
17 };
```

Um den Grunt Build Prozess zu starten muss lediglich, das in Textauszug 8 gezeigte Konsolenkomando ausgeführt werden.

Textauszug 8 Ausführen von Grunt

```
grunt dev_tests
```

3.2 Watch-Task

Ein für diese Seminararbeit wichtiger Task ist der Watch-Task[8]. Dieser erlaubt es, automatisch andere Tasks auszuführen, sobald sich der Inhalt einer definierten Datei geändert hat. Da Fehler im Programm möglichst schnell entdeckt werden sollen, wird nahegelegt, mit Hilfe des Watch-Tasks die Unit-Tests und das statisches Testen bei jedem Speichern einer JavaScript Datei ausführen zu lassen.

Textauszug 9 Watch-Task installieren

```
npm install grunt-contrib-watch --save-dev
```

Die Installation des Plugins erfolgt mittels des in Textabschnitt 9 angegebenen Konsolenkommandos. Der Zusatz *–save-dev* bewirkt dabei, dass die Abhängigkeit automatisch in der *Package.Json* hinterlegt wird. Damit das Plugin von Grunt benutz werden kann, muss es noch innerhalb der *Gruntfile* geladen werden (Siehe Textauszug 10).

Textauszug 10 Plugin laden

```
grunt.loadNpmTasks("grunt-contrib-watch")
```

Danach wird durch die folgende minimal Konfiguration in die *Gruntfile* eingefügt. Hierdurch wird auf Änderungen aller JavaScript Dateien gehorcht und nach einer Änderung der Grunttask *dev_tests*ausgeführt.

Textauszug 11 Plugin konfigurieren

```
1 config["watch"] = {
2   scripts: {
3     files: "**/*.js",
4     tasks: ["dev_tests"]
5   },
6 }
```

Der Task *dev_tests* wird im Zuge der Seminararbeit fortlaufend angepasst. Die *Gruntfile* am Ende diese Kapitels ist in Abschnitt ist in Textauszug 7 gegeben. Die finale Gruntfile befined sich im Textauszug 42 im Anhang dieser Seminararbeit.

4. JSHINT

Bevor ein ordentlicher Test für eine Anwendung geschrieben werden kann, muss zunächst die Syntax der verwendeten Programmiersprache beherrscht werden. Dies ist in JavaScript, wie in Abschnitt 2 dargelegt, nicht trivial. Daher wird empfohlen, die gängigen *pitfalls* durch statische Tests (bzw. statische Code Analyse) zu verhindern.

Für eine statische Code Analyse in JavaScript gibt es viele Frameworks, die alle relative ähnlich arbeiten. In dieser Seminararbeit wird JSHint[19] verwendet. Dies ist darin Begründet, dass JSHint sich besonders einfach in einen Grunt Build integrieren lässt.

4.1 Installation und Konfiguration

Die Installation von JSHint erfolgt über den Node Package Manager mittels des in Textauszug 12 dargestellten Kommandos.

Textauszug 12 JSHint installieren

```
npm install jshint
```

Nach dem die Installation abgeschlossen wurde, können beliebige JavaScript-Dateien, statisch getestet werden (Siehe Textauszug 13).

Textauszug 13 Anwendung von JSHint

```
jshint myJSFile.js
```

JSHint biete eine Vielzahl an Konfigurationsoptionen an [9]. So können Beispielsweise unbenutzte Variablen durch das *unused*-Flag oder undefinierte Variablen durch Verwendung des *undef*-Flags gestattet werden. Es können aber auch bestimmte Code Metriken aktiviert werden. Die unterstützen Metriken umfassen dabei von er maximale Verschachtelungstiefe über die maximale Anzahl von Parametern und Zeilen pro Funktion bis zur zyklometrischen Komplexität.

Die Konfiguration selbst kann auf unterschiedliche Arten durchgeführt werden. Zum einen kann eine Konfigurationsdatei angelegt werden. Standardmäßig wird beim Aufruf von JSHint nach einer *.jshintrc* Datei im gleichen Verzeichnis gesucht und wenn diese vorhanden ist benutzt. Ein Beispiel einer solchen Datei ist in Textauszug 14 gegeben.

Textauszug 14 .jshintrc

```
1 {
2   "undef": true,
3   "unused": true,
4 }
```

Alternativ dazu kann die Konfiguration auch inline innerhalb des Quellcodes erfolgen (siehe Textauszug 15).

Textauszug 15 Inline Konfiguration

```
1 /* jshint undef: true, unused: true */
2 // code
```

4.2 Beispiele

Im Folgenden werden Beispielausgaben von JSHint gezeigt. Als Beispielcode dienen dabei, einige der *Bad Practice* Beispiele aus Abschnitt 2.

Für das Codebeispiel 2 wird die Wahrung ausgegeben, dass möglicherweise ungewollte Eigenschaften innerhalb des For-Loops herausgefiltert werden sollten (siehe Textauszug 16). Diese gibt einen direkten Hinweis auf die korrekte Implementierung, die im Codebeispiel 3 angegeben wurde.

Textauszug 16 JSHint Warnung

```
The body of a for in should be wrapped in
    an if statement to filter unwanted
    properties from the prototype.
```

Der Quellcode in Textauszug 1 wird von JSHint mit zwei Hinweisen beanstandet (siehe Textauszug 17). Einerseits wird darauf hingewiesen, dass für Vergleiche mit *NaN* die *isNaN*-Funktion verwendet werden soll. Andererseits wird auf die unübliche und ggf. verwirrende Verwendung der Negation *!* hingewiesen.

Textauszug 17 JSHint Warnung

```
Use the isNaN function to compare with NaN
Confusing use of "!".
```

4.3 Integration in Grunt

JSHint kann mit Hilfe des *grunt-contrib-jshint* Plugins[10] in den Grunt Build eingebunden werden. Die Installation des Plugins erfolgt über das Kommando in Textauszug 18:

Textauszug 18 JSHint-Grunt-Plugin installieren

```
npm install grunt-contrib-jshint --save-dev
```

Anschließend kann das Plugin konfiguriert werden. Hierbei müssen die JavaScript Dateien angegeben werden, die getestet werden sollen. Die Konfiguration von JSHint selbst, kann entweder wieder in einer *.jshintrc* Datei erfolgen. Dazu kann der Pfad zu der Datei kann in der Pluginkonfiguration angegeben werden. Standardmäßig wird diese im gleichen Verzeichnis wie die Gruntfile erwartet. Die Konfiguration kann aber auch in einem Optionsobjekt, direkt in der Gruntfile angegeben werden. Hierbei werden die Optionen einer ggf. vorhandenen *.jshintrc*-Datei überschrieben.

Textauszug 19 Gruntfile JsHint konfiguration

```
1 config["jshint"] = {
2   options: { "undef": true} //overrides
        options in jshintrc
3   jshintrc: "config/.jshintrc" //path to
        jshintrc
4   app: ["app/**/*.js"] // include all js
        files in folder app
5 }
```

Die Gruntfile am Ende diese Abschnittes ist in Textauszug 20 gegeben. Nun wird sobald sich eine JavaScript Datei geändert hat, die Statische Code Analyse mittels JSHint auf allen JavaScript Dateien ausgeführt. Das Erstellen der Anwendung schlägt fehl, sobald JSHint eine Beanstandung am Quellcode hat.

Textauszug 20 Gruntfile

```
1 module.exports = function(grunt) {
2   //load plugins
3   grunt.loadNpmTasks("grunt-contrib-watch
        ");
4   grunt.loadNpmTasks("grunt-contrib-
        jshint");
5   //config
6   var config = {};
7   config["watch"] = {
8     scripts: {
9     files: "**/*.js",
10    tasks: ["dev_tests"]
11    }
12   };
13   config["jshint"] = {
14       all: ["**/*.js"] // include all js
            files
15    }
16    grunt.initConfig(config);
17    //register tasks
18    grunt.registerTask("dev_tests", ["
          jshint" , "watch"]);
19 };
```

5. JASMINE

Jasmine ist ein *Behavior-Driven-Devolpment* (BDD) Framework zum Erstellen vom Unit- und Integrationstests. Es bietet eine einfache und verständliche Syntax – die sich fast wie ein englischer Satz liest – um Programmspezifikationen anzugeben. Das Framework selbst ist unabhängig von dem verwendeten Browser und ist somit bestens für browserübergreifende Tests geeignet. Innerhalb dieser Seminararbeit wird ein spezieller *Testrunner* verwendet um Jasmine-Tests auszuführen. Dieser *Testrunner* wird in Abschnitt 6 vorgestellt und bringt eine Jasmine Installation mit sich. Als Standalone Variante lässt sich Jasmine mit dem Kommando in Textabschnitt 21 installieren. Danach können Testdateien mit dem Kommando *jasmine-node testfile.js* gestartet werden.

Textauszug 21 Jasmine installieren

```
npm install jasmine-node
```

5.1 Aufbau und Funktion

Ein Beispiel für einen Jasmine-Test ist in Textauszug 22 gegeben. Eine Programmspezifikation bzw. ein Test wird durch die *it*-Funktion repräsentiert. Der erste Parameter der Funktion ist der Name des Tests und der zweite Parameter der Test selbst.

Textauszug 22 Beispiel für Jasminetest

```
1 describe("a awesome module", function() {
2   it("should be awesome", function() {
3     expect(true).toBe(true);
4   });
5 });
```

Die einzelnen Tests können durch mit der *describe*-Funktion zu einer Test-Suite zusammengefasst werde. Die *describe*-Funktion hat wieder als ersten Parameter einen Namen und als zweiten Parameter eine Funktion die die *Test-Suite* beinhaltet. Innerhalb einer *Test-Suite* sind Setup bzw. Teardown Methoden mögliche. Dazu werden die Funktionen *beforeEach*, *AfterEach*, *beforeAll* und *afterAll* zur Verfügung gestellt.

Das Überprüfen des Ergebnisses bzw. – im BDD Kontext – das Angeben des zu erwartenden Wertes ist wieder bewusst leserlich gehalten. Grundsätzlich wird die Variable, die geprüft werden soll der *expect*-Funktion übergeben. Anschließend wird die Art der Überprüfung durch Aufrufen weiter Funktionen (Matcher-Funktionen) realisiert.

In Textauszug 22 werden einige der Matcher-Funktionen exemplarisch vorgestellt. In Zeile 2 wird ein einfacher Vergleich auf Gleichheit durchgeführt.

Textauszug 23 Überprüfungen

```
1 it("can do that simple stuff",function(){
2   expect(true).toBe(true);
3   expect(false).not.toBe(true);
4 });
5
6 it("can check for existance",function(){
7   var a = ["foo", "bar", "baz"];
8   expect(a).toContain("bar");
9 });
10
11 it("can do relative compairs",function(){
12   expect(2).toBeLessThan(1);
13   expect(2).not.toBeGreaterThan(1);
14 });
```

Um das Ergebnis einer Matcher-Funktion zu invertieren, kann vor der Funktion selbst ein *.not* geschrieben werde. Dies wird in Zeile 3 dazu verwendet, um auf Ungleichheit hin zu überprüfen.

Oftmals ist es notwendig, zu Testen, ob ein bestimmtes Element in einem Array vorkommt. Dies kann mit dem *toContain*-Matcher durchgeführt werden (Siehe Zeile 8). In manchen Fällen spielt der konkrete Wert einer Variablen ehr eine untergeordnete Rolle, solange er einen bestimmten Schwellwert nicht über- oder unterschreitet. Hierzu können die in Zeile 12 und 13 verwendeten *Matcher*-Funktionen verwendet werden.

5.2 Spies

Jasmine bietet ebenfalls ein einfaches, aber mächtiges, System zur Erstellung von Mocks. Diese werden von Jasmine als *Spies* (Spione) bezeichnet. Der Verdeutlichung des Mocking-Systems wird zunächst etwas *Business Logic* benötigt. Hierzu wird das in Textauszug 24 gezeigte Datenbank Modul verwendet. Das Modul hat eine Funktion, die eine ID übergeben bekommt und einen Usernamen zurückgibt.

Textauszug 24 Business Logic für Spies

```
1 var dataBase = {
2   getUserById: function(id) {
3     return "testUser";
4   }
5 };
```

Um nun zu überprüfen, ob die Funktion des Datenbank Modules aufgerufen wurde, kann ein einfacher *Spy* erstellt werden. (Siehe Textauszug 25 Zeile 2). Nun kann mit Hilfe des *toHaveBeenCalled*-Matchers der Aufruf der *getUserById*-Funktion getestet werden.

Textauszug 25 Einfacher Spies in Jasmine

```
1 it("tracks calles to getUser",function(){
2   spyOn(dataBase, "getUserById");
3   var user = dataBase.getUserById(0); //
        undefined
4
5   expect(dataBase.getUserById).
        toHaveBeenCalled();
6
7   expect(dataBase.getUserById).
        toHaveBeenCalledWith(0);
8 });
```

Bei einem einfachen *Spy* wird die *Business Logic* selbst nicht ausgeführt. Der Aufruf der *getUserById*-Funktion gibt daher den Wert *undefined* zurück. Falls es gewünscht ist, dass das ursprüngliche Modul aufgerufen wird, muss der *Spy* wie in Zeile 2 des Textauszuges 26 erstellt werden.

Textauszug 26 Spies in Jasmine

```
1 it("calls real module", function() {
2    spyOn(dataBase, "getUserById").and.
         callThrough();
3
4    var user = dataBase.getUserById(0);
5    expect(user).toBe("testUser");
6 });
```

Alternativ dazu kann auch der Spy auch angewiesen werden immer einen bestimmten Wert zurückzugeben. Um dies zu erreichen muss beim Erstellen *.and.returnValue* mit dem entsprechendem Wert aufgerufen werden. (Siehe Textauszuge 27)

Textauszug 27 Spies in Jasmine

```
1 it("mocks return value", function() {
2    spyOn(dataBase, "getUserById").and.
         returnValue("mockedTestUser");
3
4    var user = dataBase.getUserById(0);
5    expect(user).toBe("mockedTestUser");
6 });
```

5.3 Testen von asynchronen Aktivitäten

Jeder JavaScript Anwendung steht nur ein Thread zur Verfügung. Da eine Anwendung keine blockierenden Abschnitte haben soll, stellt das asynchrone Bearbeiten von Aktivitäten ein zentrales Element in JavaScript Anwendungen dar. Der Beispielcode für eine asynchrone Datenbank wird in Textauszug 28 gezeigt. Die Funktion *fetchUserById* bekommt eine ID und eine *callback*-Funktion übergeben. Zur Emulation der Asynchronität wird die *setTimeout*-Funktion verwendet. Diese führt in 500ms, die übergebene *action*-Funktion aus. Innerhalb der *action*-Funktion wird die Callback Funktion mit einem Benutzernamen aufgerufen.

Textauszug 28 Business Logik für asynchronen Datenbank

```
1 var dataBase = {
2   fetchUserById: function(id, callback) {
3     var action = function(){callback("
          testUser");};
4     setTimeout(action, 500);
5     }
6 };
```

Für das Testen solcher Szenarien biete Jasmine ein *done*-Callback an. Diese wird der Testfunktion als erster Parameter übergeben. Jasmine sieht einen Test als asynchronen Test an, wenn der erste Parameter der Testfunktion den Namen *done* besitzt.

Ein Beispiel für einen asynchronen Teste ist in Textauszug 29 gegeben. Das überprüfen der Daten wird in der Callback Funktion der *fetchUserById*-Funktion durchgeführt. Nach dem alle Überprüfungen abgeschlossen sind, wird die *done*-Funktion aufgerufen. Jasmin wertet einen asynchronen Test als fehlgeschlagen, wenn entweder eine der Überprüfungen fehlschlägt oder die *done*-Funktion nicht, innerhalb eines definierten Timeouts, aufgerufen wird.

Textauszug 29 Asynchronität in Jasmine

```
1   it("can handle asynchron suff",
        function(done) {
2    dataBase.fetchUserById(0, function(
         user){
3       expect(user).toBe("testUser");
4   done();
5     });
6  });
```

Für das Ausführen, der mit Jasmine geschriebenen Tests, wird eine Testumgebung benötigt. Da Jasmin selbst dazu nur eine sehr unflexiblen Mechanismus bereitstellt, wird empfohlen einen externen *Testrunner* zu verwenden. Innerhalb dieser Seminararbeit wird der in Abschnitt 6 vorgestellte *Testrunner* Karma verwendet.

6. KARMA

Karma [2] ist ein *Testrunner*, der es ermöglicht verschiedene Browser als Testumgebung zu verwenden. Neben den gängigen Browsern (z.B. Google Chrome, Mozilla Firefox [11, 12]) kann ebenfalls ein schneller *Headless*-Browser (PhantomJS [13]) verwendet werden. Karma ist auch in der Lage, Tests auf externen Geräten (z.B. Smartphone) auszuführen.

6.1 Installation und Konfiguration

Die Installation von Karma erfolgt abermals über das *Node Package Management* (npm) mittels des in Textauszug 30 gezeigtem Kommandos.

Textauszug 30 Karma installieren

```
npm install karma --save-dev
```

Neben Karma selbst, werden auch noch einige Plugins benötigt. Zum einen wird ein Jasmine Plugin gebraucht, um Jasmine-Tests ausführen zu können. Zum anderen werden für die Unterschiedlichen Browser sog. *launcher* benötigt. Diese Karma-Plugins können ebenfalls über npm bezogen werden (siehe Textauszug 31).

Textauszug 31 Karma Plugins installieren

```
npm install karma-jasmine
    karma-chrome-launcher
    karma-firefox-launcher
    karma-phantomjs-launcher --save-dev
```

Damit Karma über die Konsole einfacher gesteuert werden kann, ist es zu empfehlen das *Command Line Interface* von Karma zu installieren (Siehe Textauszug 32).

Textauszug 32 Karma installieren

```
npm install karma-cli -g
```

Damit Karma benutzt werden kann, muss es zunächst Konfiguriert werden. Dieses wird durch eine Konfigurationsdatei realisiert. Standardmäßig wird eine Datei mit dem Namen *karma.config.js* verwendet. Grundsätzlich lässt sich Karma vielseitig Konfigurieren [14]. Für eine Benutzung sind aber nur wenige Einstellungen nötig. Ein Beispiel für eine minimal Konfiguration ist in Textauszug 33 gegeben. In Zeile 3 wird das Testframework festgelegt. Alternative zu Jasmine könnte auch *mocha* oder *qunit* verwendet werden. Damit Karma die Tests und die *Business Logic* finden kann müssen die entsprechenden JavaScript Dateien innerhalb der Konfiguration geladen werden. Dies wird in den Zeilen 4 bis 7 gemacht. Zur Vereinfachung unterstützt Karma an dieser Stelle auch reguläre Ausdrücke. Schlussendlich muss noch die Testumgebung definiert werden. Im Minimalbeispiel wird in Zeile 8 der PhantomJS Browser als Testumgebung gewählt.

Textauszug 33 karma.config.js Datei

```
1 module.exports = function(config) {
2   config.set({
3     frameworks: ["jasmine"],
4     files: ["scripts/**/*.js",
5             "specs/**/*.js"],
6     browsers: ["PhantomJS"]
7   });
8 };
```

Nach der Konfiguration kann Karma durch Kommando *karma run* gestartet werden

6.2 Einbinden von Karma in Grunt

Bevor ein Grunttask angelegt werden kann muss zunächst das entsprechende Grunt Plugin [15] installiert werden. Neben diesem wird zusätzlich noch das *karma-coverage*-Plugin installert [16]. Dieses wird benötigt um Berichte bezüglich der Testabdeckung erstellen zu können. Abbildung 1 zeigt die Ausgabe des Plugins anhand einer produktive eingesetzen JavaScript Anwendung.

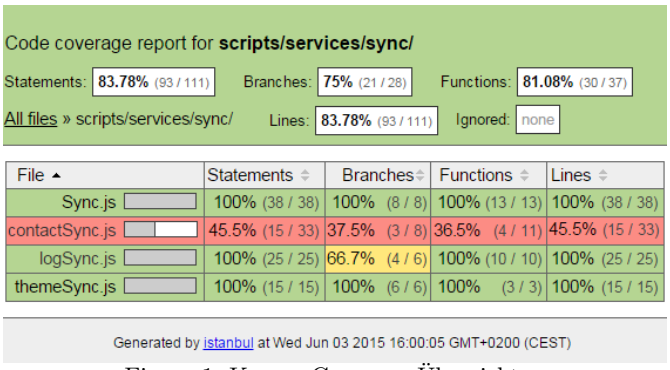

Figure 1: Karma-Coverage Übersicht

Das Kommando für die Installation beider Plugins ist in Textausschnitt 34 gegeben. Innerhalb der *Gruntfile* muss anschließend, durch die Anweisung *grunt.loadNpmTasks("grunt-karma");* das Karma-Plugin geladen werden.

Textauszug 34 Karma-Grunt-Plugin installieren

```
npm install grunt-karma
    karma-coverage --save-dev
```

Für die Integration von Karma in den Gruntbuild werden zwei Tasks angelegt. Der erste Task (*karma:dev*) soll kontinuierlich, bei jedem Speichern einer Quellcode Datei die Tests ausführen. Hierzu ist es erforderlich, möglichst schnell die Tests ausführen zu können. Aus diesem Grund wird als Testumgebung lediglich der schnelle PhantomJS Browser benutzt. Während der erste Task dazu dient die Programmlogik zu testen, wird im zweiten Task (*karma:prod*) getestet, ob sich das Programm unter realen Bedingungen gleich verhält. Dieser Task sollte im besten Fall bei jedem Commit von einem Buildserver ausgeführt werden. Daher ist es akzeptabel wenn er mehr Zeit beansprucht, als der erste Task.

Zur Erstellung dieser Tasks, wird Karma ausschließlich innerhalb der Gruntfile konfiguriert. Prinzipiell ist es auch möglich eine Datei zur Konfiguration anzulegen. Da zur Erreichung der o.g. Ziele nur wenige Konfigurationen nötig sind, wird aber auf das Erstellen von Konfigurationsdateien verzichtet.

Textauszug 35 Gruntfile Karma konfiguration

```
1 config["karma"] = {
2   options: {
3     frameworks: ["jasmine"],
4     files: ["scripts/**/*.js", "specs
          /**/*.js"]
5
6     //config coverage
7     reporters: ["progress", "coverage"],
8     preprocessors: {
9       "specs/**/*.js": ["coverage"]
10    },
11    coverageReporter: { type : "html",
          dir : "coverage/"  }
12  },
13  prod:{ browsers:["Chrome","Firefox"] },
14  dev:{ browsers: ["PhantomJS"] }
15 };
```

Die Grunt Konfiguration für beide Karma-Tasks ist in Textauszug 35 gegeben. Innerhalb der Zeilen 2 bis 13 werden Tasks übergreifende Einstellungen definiert. In Zeile 3 wird Jasmine als Testframework festgelegt und in Zeile 4 die Source- bzw. Test-Dateien. Die Zeilen 6-11 sind für die Konfigurierung der Testabdeckungsberichte zuständig. Anschließend werden in Zeile 13 die Browser *Chrome* und *firefox* als Testumgebung für den *karma:prod*-Task festgelegt. Für den *karma:dev*-Task wird in Zeile 14 *PhantomJS* als Testumgebung gewählt.

Im Anschluss an die Konfiguration muss noch der Metatask *dev_tests* angepasst werden. Zudem wird noch ein weiterer Metatask mit dem Namen *prod_tests* angelegt. Beides ist in Textauszug 36 gezeigt.

Textauszug 36 Metatasks mit Karma

```
1 //register tasks
2 grunt.registerTask("dev_tests",
3   ["jshint", "karma:dev", "watch"]);
4
5 grunt.registerTask("prod_tests",
6   ["jshint", "karma:prod"]);
```

7. END-TO-END-TEST

Die letzte Art von Tests, mit denen sich diese Seminararbeit beschäftigt, sind die *end-to-end*-Tests oder auch *Selenium*-Tests. Diese simulieren Benutzerverhalten und testen somit Anwendung als Ganzes. Hierzu wird einen Selenium Server [17], Webdrive.io [3] und Jasmine als Technologien eingesetzt.

Ein Selenium Server ist in der Lage automatisierbare Instanzen der gängigen Browser zu erzeugen. Zudem bietet er eine einheitliche API (in mehreren Programmiersprachen) zur Steuerung und zum Auslesen des Browsers zur Verfügung. Webdriver.io ist eine JavaScript Implementierung der Selenium API.

7.1 Installation

Damit die Installation gelingt muss Java und Node.js auf dem Rechner installiert sein. Zunächst wird ein Standalone Selenium Server benötigt. Dieser kann von direkt von der Selenium Webseite heruntergeladen werden. Anschließend kann der Server durch den, in Textauszug 37 gezeigten, Befehl gestartet werden.

Textauszug 37 Download Selenium Server

```
java -jar selenium-server-standalone.jar
```

Als nächstes muss Webdriverio selbst installiert werden. Dies ist dank des Node Package Manager sehr einfach und kann durch das Kommando in Textauszug 38 realisiert werden

Textauszug 38 Download Selenium Server

```
npm install webdriverio --save-dev
```

7.2 WebdriveIO

Nachdem die Installation abschlossen ist und der Selenium Server gestartet wurde, kann der Selenium Server mittels WebdriveIO gesteuert werden. Der Textauszug 39 zeigt ein einfaches Beispiel zum Auslesen des Titels einer Webseite. In Zeile 1 wird das WebdriverIO-Modul geladen und in Zeile 2 eine Variable mit Optionen angelegt. Innerhalb der Optionen (Zeile 4) wird der Firefox-Browser als Ziel Browser festgelegt. Die Steuerung des Selenium Servers beginnt in Zeile 7. Zunächst werden dazu in Zeile 8 die Optionen übergeben und anschließend der Server initialisiert. Zeile 10 bringt den Server dazu die Webseite *http://seblog.cs.uni-kassel.de* aufzurufen. Mit der 11. Zeile wird der Title der aktuellen Seite abgefragt und dieser in Zeile 12 auf der Konsole ausgeben.

Textauszug 39 WebdriveIO Beispiel

```
1 var webdriverio = require("webdriverio");
2 var options = {
3   desiredCapabilities: {
4     browserName: "firefox"
5   }
6 };
7 webdriverio
8   .remote(options)
9   .init()
10  .url("http://seblog.cs.uni-kassel.de")
11  .title(function(err, res) {
12    console.log("Title was:"+res.value);
13  })
14  .end();
```

Die API von WebdriverIO ist vielseitig um ermöglicht es einen Benutzer vollständig zu simulieren. Zudem werden auch Umfangreichende Befehle zum Abfragen der gängigen Seiteneigenschaften (z.B.: Title, Seitentext, CSS-Eigenschaft). Die Vollständige API ist in [3] beschrieben.

7.3 Integration mit Jasmine

Für eine bessere Strukturierung der Testfälle werden die Tests abermals mit Jasmine umgesetzt. Dabei werden unterschiedliche Bestandteile von Jasmine eingesetzt um das Testen zu vereinfachen.

Ein Beispiel für einen Selenium Test, der den Title einer Webseite testet ist in Textauszug 40 gegeben. Auch hier wird zunächst in Zeile 1 das WebdriverIO Modul geladen. Anschließend wird eine Jasmine Test-suite deklariert (Zeile 2). Innerhalb des *beforeEach*-Block (Zeile 4-7) wird vor dem Ausführen jedes Testes eine neue Verbindung zum Selenium Server erstellt und Initialisiert. Hierbei wird *options* aus Textauszug 39 übernommen. In Zeile 9 beginnt, mit einem *it*-Block, der eigentliche Test. Dabei wird ähnliche des Beispiels in Testabschnitt 39 der Selenium Server beauftragt, eine Webseite aufzurufen und den Titel der Webseite zurückzugeben. Die Überprüfung des Titels wird innerhalb der *getTitle*-Funktion ausgeführt. Hierzu wird Jasmine-Syntax verwendet. In Zeile 19 wird der Befehl abgeschickt. Sobald dieser komplett vom Selenium-Server ausgeführt wurde, wird die *done*-Funktion aufgerufen. Hierdurch wird Jasmine von der Beendigung des Asynchronen Testes benachrichtigt. In den Zeilen 22-24 wird mit Hilfe des *afterEach*-Blockes, nach jedem Test die Verbindung zum Selenium Server ordnungsgemäß beendet.

Textauszug 40 WebdriveIO mit Jasmine

```
1 var webdriverio = require("webdriverio");
2 describe("Test for SE-Blog", function(){
3   var client = {};
4   beforeEach(function(){
5     client = webdriverio.remote(options);
6     client.init();
7   });
8
9   it("should have the correct title",
10    function(done) {
11    client
12      .url("http://seblog.cs.uni-kassel.de")
13      .getTitle(function(err, title) {
14        expect(err).toBeFalsy();
15        expect(title).toBe("Web Engineering"
16          + " | Software Engineering"
17          + " Research Group Kassel");
18      })
19      .call(done);
20  });
21
22  afterEach(function(done) {
23    client.end(done);
24  });
25 });
```

Die Selenium Tests können manuell, oder von einem Buildserver, direkt mit Jasmine ausgeführt werden (Siehe Textauszug 41).

Textauszug 41 Ausführen mit Jasmine

```
jasmine-node testfile.js
```

8. FAZIT

Innerhalb dieser Seminararbeit wurden die grundlegenden Technologien und Methodiken vorgestellt, die Benötigt werden um JavaScript-Anwendungen ganzheitlich zu Testen. Somit wird es einem Entwickler ermöglicht professionell Anwendungen in JavaScript zu Erstellen.

9. REFERENCES

[1] Jasmine - Behavior-Driven JavaScript
`https://www.jasmine.github.io/`.
[Online; zugegriffen am 7. Mai 2015].

[2] Karma - Spectacular Test Runner for Javascript
`https://www.karma-runner.github.io/`.
[Online; zugegriffen am 7. Mai 2015].

[3] WebdriverIO - Selenium 2.0 javascript bindings for NodeJS
`https://www.webdriver.io/`.
[Online; zugegriffen am 7. Mai 2015].

[4] Grunt - The JavaScript Task Runner
`https://www.gruntjs.com/`.
[Online; zugegriffen am 7. Mai 2015].

[5] Apache Cordova
`https://cordova.apache.org/`.
[Online; zugegriffen am 4. Juni 2015].

[6] AngularJS - Superheroic JavaScript MVW Framework
`https://angularjs.org/`.
[Online; zugegriffen am 4. Juni 2015].

[7] Node.Js - JavaScript runtime
`https://nodejs.org/`.
[Online; zugegriffen am 20. Mai 2015].

[8] Watch Task, grunt-contrib-watch
`https://github.com/gruntjs/grunt-contrib-watch`.
[Online; zugegriffen am 20. Mai 2015].

[9] JSHint - Configuration
`http://jshint.com/docs/options/`.
[Online; zugegriffen am 20. Mai 2015].

[10] JSHint Grunt Task, grunt-contrib-jshint
`https://github.com/gruntjs/grunt-contrib-jshint`.
[Online; zugegriffen am 20. Mai 2015].

[11] Google Chrome
`https://www.google.de/chrome/browser/desktop/`.
[Online; zugegriffen am 1. Juni 2015].

[12] Mozilla Firefox
`https://www.mozilla.org/de/firefox/new/`.
[Online; zugegriffen am 1. Juni 2015].

[13] PhantomJS - Full web stack No browser required
`http://phantomjs.org/`.
[Online; zugegriffen am 1. Juni 2015].

[14] Karma - configuration options
`http://karma-runner.github.io/0.10/config/configuration-file.html`.
[Online; zugegriffen am 1. Juni 2015].

[15] Grunt-Karma - grunt plugin
`https://github.com/karma-runner/grunt-karma`.
[Online; zugegriffen am 1. Juni 2015].

[16] Karma-coverage - Code Coverage Plugin
`https://github.com/karma-runner/karma-coverage`.
[Online; zugegriffen am 1. Juni 2015].

[17] SeleniumHQ - Selenium automates browsers
`http://docs.seleniumhq.org/d`.
[Online; zugegriffen am 20. Mai 2015].

[18] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[19] A. Kovalyov. JSHint, a JavaScript Code Quality Tool
`https://www.jshint.com/`.
[Online; zugegriffen am 7. Mai 2015].

[20] R. C. Martin. *Clean Code: A Handbook of Agile*

Software Craftsmanship. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[21] R. C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2011.

[22] R.B. Why does javascript suck? `http://whydoesitsuck.com/why-does-javascript-suck/`. [Online; zugegriffen am 7. Mai 2015].

APPENDIX

Textauszug 42 Finale Gruntfile

```
module.exports = function(grunt) {
  //load plugins
  grunt.loadNpmTasks("grunt-contrib-watch");
  grunt.loadNpmTasks("grunt-contrib-jshint");
  grunt.loadNpmTasks("grunt-karma");

  //config
  var config = {};
  config["watch"] = {
    scripts: {
    files: "**/*.js",
    tasks: ["dev_tests"]
    }
   };

  config["jshint"] = {
    all: ["**/*.js"] // include all js files
    }

  config["karma"] = {
    options: {
      frameworks: ["jasmine"],
      files: ["scripts/**/*.js","specs/**/*.js"]

      //config coverage
      reporters: ["progress","coverage"],
      preprocessors: {
        "specs/**/*.js": ["coverage"]
      },
      coverageReporter: { type : "html", dir : "coverage/"  }
    },
    prod:{browsers:["Chrome","Firefox"]},
    dev:{browsers: ["PhantomJS"]}
  };
  grunt.initConfig(config);

  //register tasks
  grunt.registerTask("dev_tests",
    ["jshint", "karma:dev", "watch"]);

  grunt.registerTask("prod_tests",
    ["jshint", "karma:prod"]);
};
```