Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 291

Implementierung und Test einer mobilen Hybrid-Anwendung für Fußgängernavigation

Oliver Röhrdanz

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Dipl.-Inf. Sebastian Wagner, Dr. rer. nat. Daniel Schleicher
Beginn am:	8. September 2015
Beendet am:	9. März 2016
CR-Nummer:	H.2.1, D.2.5, D.2.11, E.2, J.4

Kurzfassung

Der Trend zu mobiler Navigation und standortbezogenen Diensten nimmt in den letzten Jahren stetig zu. Die meisten Systeme eignen sich jedoch nicht immer für die Fußgängernavigation. Zu diesem Zweck wurde die Anwendung Nifino entwickelt. Neben der Lokalisierung von Freunden soll Nifino Personen auch zu statischen oder sich bewegenden Entitäten, den sogenannten Beacons, führen können. Nifino wird als hybride Mobilanwendung auf Basis von Ionic, AngularJS und Cordova geschrieben.

Im Rahmen dieser Bachelorarbeit wurde ein Beaconkonzept in der hybriden Mobilanwendung implementiert. Die hierfür erstellte Datenstruktur erlaubt es, geräteunabhängig verwendet zu werden. Da sich Nifino noch im Aufbau befindet, kann die Datenstruktur ohne großen Aufwand erweitert werden, um in Zukunft neue Features zu ermöglichen. Zudem können in der entwickelten Version von Nifino Beacons in der Umgebung gefiltert und gefunden, oder bei Bedarf neue Beacons erstellt und bearbeitet werden. Um Nifino besser in das existierende Android Ökosystem zu integrieren wurden native Plugins entwickelt. Diese native Plugins erlauben es zum Beispiel Benachrichtungen anzuzeigen sofern eine Navigation aktiv ist.

Neben den Beacons werden in dieser Arbeit verschiedene Testverfahren und Tools vorgestellt, die in mobilen Umgebungen Verwendung finden können und zur Implementierung der Testmethoden angewandt werden. Diese Tools werden gegeneinander evaluiert. Unter Verwendung der jeweiligen Tools wurden in dieser Arbeit Unit- und End-to-End Testfälle geschrieben, um die Anwendung systematisch und nach den Kriterien der Softwaretechnik zu testen.

Inhaltsverzeichnis

1. Einleitung **13**

2. Grundlagen **17**

3. Testverfahren **21**
3.1. Teststufen . . . 22
3.1.1. Unittest . . . 22
3.1.2. Integrationstest . . . 23
3.1.3. Systemtest . . . 23
3.2. Testmethoden . . . 24
3.2.1. Regressionstest . . . 24
3.2.2. White Box Test . . . 25
3.2.3. Gray Box Test . . . 25
3.2.4. Black Box Test . . . 25
3.2.5. End-to-End Testing . . . 25
3.2.6. Testgetriebene Entwicklung . . . 26

4. Test Tools **27**
4.1. Anforderungen an die Test Tools . . . 28
4.2. Browser Treiber . . . 31
4.2.1. Selenium . . . 31
4.3. Testrunner . . . 32
4.3.1. Karma . . . 33
4.3.2. Testem . . . 34
4.3.3. Protractor . . . 34
4.3.4. Nightwatch . . . 35
4.4. Testframeworks . . . 36
4.4.1. Jasmine . . . 36
4.4.2. Mocha . . . 37
4.4.3. QUnit . . . 38
4.5. Laufzeitumgebungen . . . 38
4.5.1. PhantomJS . . . 39
4.5.2. Appium . . . 40
4.5.3. Ripple . . . 40
4.6. Zusammenfassung . . . 42

5. Implementierung **45**
5.1. Beacons . . . 45
5.1.1. Datenstruktur . . . 45
5.1.2. Erstellen und Bearbeiten von Beacons . . . 49
5.1.3. Beaconsuche . . . 52
5.1.4. Benutzerprofil . . . 54
5.1.5. Notifications . . . 55
5.2. Tests . . . 56
5.2.1. Unit Tests . . . 56
5.2.2. End to End Tests . . . 65

6. Zusammenfassung und Ausblick **75**

A. Anhang **77**
A.1. Use Case Beschreibungen . . . 77

Literaturverzeichnis **83**

Abbildungsverzeichnis

2.1. Nifino Aufbau 17

3.1. Teststufen 22

4.1. Nifino Tracking [Nif] 29
4.2. Selenium 32
4.3. Jasmine Test Szenario 36
4.4. Ripple Emulator Screenshot 41

5.1. Beacon Use Case 46
5.2. Beacon Datenstruktur 47
5.3. Beacon erstellen 49
5.4. Google Maps Direktive 51
5.5. Beacons finden 53
5.6. Benutzer Profil 54
5.7. Benachrichtigungs Plugin 55
5.8. Model-View-ViewModel 58
5.9. Karma Ausgabe 62
5.10. Nifino Login Dialog 72

Tabellenverzeichnis

4.1. Test Tools im Überblick 27
4.2. Überblick der genannten Testframeworks 42
4.3. Überblick der End-to-End Laufzeitumgebungen 43

5.1. Karma Konfigurationen 57
5.2. Vergleich zwischen Controller und Services [SG14] 64
5.3. Protractor Konfigurationen [Anga] 69

A.1. Use Case: Beacon anlegen 78
A.2. Use Case: Beacon bearbeiten 79
A.3. Use Case: Beacons finden 80
A.4. Use Case: Benutzerprofil bearbeiten 81

Verzeichnis der Listings

5.1. Map Direktive 50
5.2. Konfiguration für Karma Testrunner 57
5.3. Factory Mocking 60
5.4. Asynchroner Testfall - erfolgreiches Einloggen 61
5.5. Controller Injection 61
5.6. Directive Injection 63
5.7. Pretty Filter Test Suite 66
5.8. Protractor Konfiguration 68
5.9. Ausschnitt aus dem Login Testfall 71
5.10. Protractor als Gulp Task 72

1. Einleitung

Die Entwicklungen der vergangenen Jahre haben gezeigt, dass mobile Helfer wie Smartphones oder Smartwatches in unserem Leben eine immer wichtigere Rolle spielen. Für die meisten ist das Smartphone aus dem täglichen Leben nicht mehr wegzudenken, dies zeigt allein schon die Tatsache, dass „Smombie“ zum Jugendwort des Jahres 2015 gewählt wurde.

Allerdings eröffnet uns unser täglicher Helfer auch viele neue Möglichkeiten und kann auch im Alltag helfen Stress zu vermeiden und auftretende Probleme leichter zu bewältigen. Die Kommunikation vieler Kinder, Jugendlicher und inzwischen auch Personen aus der nicht „Digital Native“ Generation läuft inzwischen fast ausschließlich über unser Smartphone. WhatsApp, Telegram und Co. erfreuen sich enormer Beliebtheit. Auch um sich dann zu verabreden werden die digitalen Helfer verwendet. Verabredet man sich mit Jemandem, helfen die Messenger Apps aber nur bedingt weiter. In einer überfüllten Metro, Innenstadt oder auf einem Festival fällt es schwer sich direkt zu sehen, da die Menschenmasse zu groß scheint. Google Maps und andere Navigationsdienste können helfen den Weg zum Treffpunkt zu finden, doch auf den letzten Meter versagen auch diese Dienste, da die Lokalisierung zu ungenau ist.

Eine App, die einem dabei hilft, die sich möglicherweise umherbewegende Zielperson, schließlich auch ausfindig zu machen, würde uns hier enorm helfen und uns unseren Alltag noch stressfreier durchleben lassen. Diese Möglichkeit sich auf den letzten Meter navigieren zu lassen erlaubt uns die mobile Applikation Nifino.
Ebenso helfen einem Dienste wie Foursquare oder Google Places die „Points of Interests“ der Umgebung zu entdecken, versagen jedoch Personen bis auf den letzten Meter zu eben diesen Punkten zu führen.

Nifino will auch diesen Punkt angehen und führt daher das Konzept der Beacons ein. Beacon ist hierbei ein relativ abstrakter Begriff, der für eine Person, ein Event, eine Sehenswürdigkeit oder eine Location stehen kann. Es soll für den Nutzer auch möglich sein solche Beacons nach Belieben zu erstellen. Diese Idee wurde in der bereits existenten iOS App umgesetzt. Um allerdings ein größeres Publikum zu haben, soll diese Idee auch für andere Plattformen, vor allem Android, umgesetzt werden. Dazu soll Nifino als mobile HybridApp implementiert werden, die auf einer Vielzahl verschiedener Plattformen und Geräte laufen soll.
Als Basis hierfür dient Cordova und Ionic, das auf AngularJS basiert. Die Anwendung wird als Hybridapp in Javascript, HTML5 und CSS 3 implementiert und läuft nicht nativ auf einem

Gerät, kann jedoch plattformunabhängig in native Webcontainer eingebettet werden und über die lokalen Appstores deployed werden.

Ein Ziel dieser Arbeit ist es das Beaconkonzept zum einen auf die Hybride Applikation zu übertragen und zum anderen auch zu erweitern. Doch mit steigendem funktionalen Umfang einer App steigt zwangsläufig auch die Komplexität dieser. Jeder Smartphonebesizter und -nutzer hat sich schon ein mal eine Anwendung aus dem Appstore installiert und musste dann feststellen, dass diese die beschriebene Funktionalität nicht erfüllt oder direkt beim Starten wieder beendet wird. Nifino soll, wie andere Smarthpone Applikationen oder allgemein Programme auch, auf die Funktionalität hin getestet werden umd damit die Verwendbarkeit zu gewährleisten.

Die App basiert auf dem Ionic Framework und damit auf AngularJS. Zum Testen von AngularJS Applikation gibt es bereits einige Dokumentation und Bücher, die einem das ein oder andere Testfahren und Testframework empfehlen. In Kombination mit Ionic oder Cordova existiert bisher kaum Dokumentation. Die bekannten AnguarJS Testweisen lassen sich auch nur bedingt auf die mobile App übertragen, da eine Smartphone Applikation mehr Möglichkeiten bereitstellt als dies nur im Web der Fall wäre. Smartphones unterstützen (Push-)Benachrichtigungen und Sensoren die in einer Webumgebung nicht vorzufinden sind.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: In diesem Kapitel werden grundlegende und häufiger verwendete Begriffe kurz erläutert um die weiteren Ausführungen der Arbeit besser nachvollziehen zu können.

Kapitel 3 – Testverfahren: Die Softwaretechnik bietet eine Menge Möglichkeiten Software auf verschiedenste Art und Weisen zu testen. Zu diesen Möglichkeiten und deren Herausforderungen gibt es bereis eine Menge Dokumentation. Dieses Kapitel soll die wichtigstens Ansätze zusammenfassen um einen Einblick in das Testen von Software zu geben.

Kapitel 4 – Test Tools: Nachdem grundlegene Begriffe und Möglichkeiten des Testens im vorherigen Kapitel erläutert wurden, beschäftigt sich dieses Kapitel konkret mit dem Thema Testen für Hybrid Apps. Hierbei werden verschiedene Frameworks vorgestellt und deren Funktionalität erläutert und ggf. Probleme oder fehlende Features aufgezeigt.

Kapitel 5 – Implementierung: Dieses Kapitel beschreibt die Implementierungen die im Laufe dieser Arbeit erfolgt sind. Aufgeteilt wird dieses Kapitel in zwei Sektionen. Zuerst wird auf das Beaconkonzept eingegangen, wie dieses erweitert und implementiert wurde und wie dieses in Zukunft noch erweitert werden kann. Anschließend wird die Testumgebung für Nifino betrachtet, wie diese aufgebaut ist, welche Probleme aufgetreten sind und wie mit diesen umgegangen wurde.

Kapitel 6 – Zusammenfassung und Ausblick: In diesem Kapitel werden die wesentlichen Punkte der Arbeit zusammengefasst und erneut kurz beleuchtet. Zudem wird in diesem Kapitel ein Ausblick auf die Zukunft gegeben und wie das Beschriebene umgesetzt und weiter ausgebaut werden kann um auch zukünftigen Anforderungen zu entsprechen.

2. Grundlagen

In diesem Kapitel werden häufig verwendete Begriffe kurz beschrieben, um die weiteren Ausführungen dieser Arbeit besser nachvollziehen zu können. Sofern es für das Verständnis erforderlich ist diese genauer zu erklären, wird dies zum jeweiligen Zeitpunkt im Laufe dieser Arbeit gemacht.

Abbildung 2.1 zeigt den Aufbau und die Zusammenhänige zwischen den einzelnen, auch hier beschriebenen Frameworks.

Node.js

Node.js ist eine Javascript Runtime und basiert auf der V8 Javascript Runtime die auch in Google Chrome Verwendung findet. Mit dem NPM (Node Package Manager) bietet diese ein enorm großes Ökosystem. Auf der Homepage wird vom größten Ökosystem der Welt gesprochen[Nod]. Node.js ist zwar kein Javascript Framework, sondern wurde v.a. in C und C++ geschrieben, dennoch lassen sich diese Erweiterungen in Javascript implementieren. Mit Hilfe dieser Runtime kann ein großes und skalierbares Netzwerk von verschiedenen Anwendungen geschaffen werden, die alle serverseitig ausgeführt werden. Diese Möglichkeit der Skalierung wird v.a. durch zwei Faktoren ermöglicht. Zum einen durch die ereignisgesteuerte Architektur, hierbei werden Events durch Zustandsänderungen ausgelöst[06], und zum anderen durch die asynchrone Ausführung von I/O Operationen.

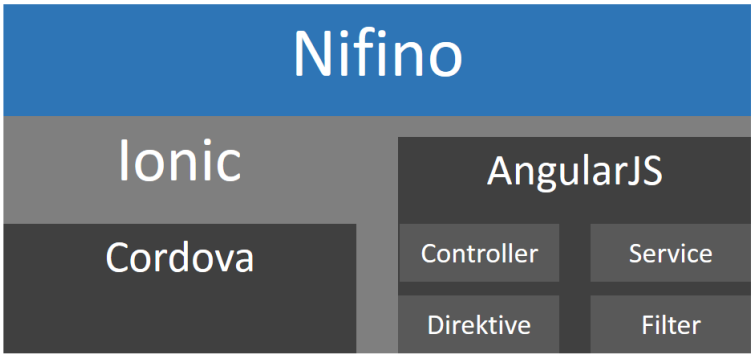


Abbildung 2.1.: Nifino Aufbau

AngularJS

AngularJS ist ein Javascript Framework, das im Gegensatz zu Node.js clientseitig ausgeführt wird. Entwickelt wird dieses Projekt von Google und eignet sich um Single Page Applications nach dem Model-View-Controller Pattern zu erstellen. AngularJS setzt hierbei auf die Erweiterung des HTML Standards und führt eigene Typen und Modifizierer ein. Dies hat den Vorteil, dass die Seite dynamisch angepasst werden kann ohne auf DOM Elemente zugreifen zu müssen. AngularJS setzt für die Realisierung Controller, Direktiven, Services und Filter ein[Bra14]. Diese Elemente sind in Abschnitt 5.2.1 genauer beschrieben.

Controller Controller entsprechen hierbei einem kombinierten View und Controller im MVC Pattern. Controller lassen sich anschließend zu Modulen zusammenfassen und als benötigte Abhängigkeiten in andere Module und die Applikation einbinden.

Service Services lassen sich auf gewisse Art und Weise mit Controllern vergleichen. In Services wird Geschäftslogik implementiert, diese werden als Singleton Objekt instanziiert. Damit besitzen diese einen Zustand, der in verschiedenen Komponenten verwendet werden kann.

Direktive Direktiven sind wiederverwendbare Komponenten. Mit Hilfe dieser lässt sich das bekannte HTML Vokabular über den Standard hinaus erweitern. Sofern trotz Verwendung von AngularJS ein Zugriff auf DOM Elemente nötig ist, sollte dies in Direktiven geschehen.

Filter Filter lassen sich einsetzen um die Ausgabe zu formatieren und den eigenen Wünschen anzupassen.

Cordova

Cordova (ehemals Phonegap) ist ein Framework, das zur Erstellung hybrider Applikation auf mobilen Endgeräten verwendet werden kann. Das Framework erlaubt es hierbei Anwendungen mit den Webtechnologien HTML 5, Javascript und CSS 3 zu entwickeln. Cordova Apps sind weder native Apps, der Code wird von nativen Webviews ausgeführt, noch Web basiert, da die Applikationen aus den gerätespezifischen Appstores bezogen werden können. Cordova unterstützt inzwischen auch native Plugins, mit denen gerätespezifischer Code aus der Webview Umgebung aufgerufen werden kann. Diese Plugins erlauben es, Benachrichtigungen in der App zu verwenden, jedoch müssen diese für jede Plattform separat entsprechend implementiert werden[Cor].

Ionic

Ionic ist ein komplettes Entwicklungs Framework für die Erstellung von hybrider Applikationen. Dabei baut Ionic auf AngularJS und Cordova auf. AngularJS wird verwendet um die Appstruktur aufzubauen und die Logik zu implementieren. Ionic selbst kümmert sich vor allem um das Look and Feel und die Performance der Anwendung. Cordova findet beim Deployen des Projekts Verwendung, um die Appdatei anschließend in den entsprechenden Appstores bereitstellen zu können[Ion].

Nifino

Nifino ist unsere Anwendung, die Fußgängern Navigation, im Gegensatz zu bisher bekannten Navgationsanwendungen, auch auf den letzten Meter ermöglichen soll. Nifino soll hierbei als hybride Applikation entwickelt werden, um eine Verwendung auf Android und IOS zu ermöglichen. Hierbei soll eine Beaconstruktur entwickelt werden. Diese Beacons stellen bspw. Orte oder Events dar zu denen navigiert werden kann. Das Beaconkonzept wurde auch von Muhammad Bilal Chughtai in seiner Arbeit „Concept and Implementation of Digital Beacons“[Chu15] vorgestellt.

3. Testverfahren

> Tests sind nicht die einzige Maßnahme im Qualitätsmanagement der Softwareentwicklung, aber oft die letztmögliche. [...]
>
> *(Pol, Koomen und Spillner)*

Ein Test der Software prüft diese bezüglich der Spezifikation, also ob die Software, den definierten Anforderungen entspricht oder nicht. Ein solcher Test dient auch zum Messen der Qualität von Software. Die Ergebnisse der Tests können anschließend verwendet werden um eventuell entdeckte Fehler in der Software auszubessern und diese dann möglichst bald und fehlerfrei in Betrieb nehmen zu können.

Das Softwaretesten im Allgemeinen bezieht sich nicht auf einen solchen Test, sondern auf die Gesamtheit aller Maßnahmen zur Überprüfung von Software. Hierzu gehört u.a. das Überprüfen der Anforderungen, das Verhalten gegenüber beliebigen, auch fehlerhaften Eingaben, die Geschwindigkeit der Anwendung, die Nutzbarkeit und das Gesamtverhalten der Software genauer unter die Lupe zu nehmen.

Beim Testen ist es nicht möglich zu zeigen, dass die Software fehlerfrei ist, es kann nur gezeigt werden, dass Fehler vorhanden sind. Die Software ist in der Regel nur so gut, wie auch ihre Tests, werden seltene Extremfälle nicht getestet, kann dies im Betrieb zu Problemen führen. Deshalb ist es ratsam verschiedene Testverfahren und -metriken miteinander zu kombinieren, um ein möglichst breites Spektrum an Fehlerquellen zu identifizieren, zu testen und ggf. die gefundenen Fehler zu beheben.

Pol, Koomen und Spillner [Mar02] bezeichnen Tests nicht als einzige Maßnahme zur Qualitätssicherung, aber als letztmögliche. Je später ein Fehler entdeckt wird, umso aufwendiger ist die Behebung, daher muss Qualität die gesamte Projektzeit implementiert werden und kann nicht hineingetestet werden.

Je nachdem, wie viel Aufwand und Vorbereitung für diese Tests aufgewendet wird, lassen sich diese unterschiedlich klassifizieren. Ein **Laufversuch** bezeichnet das geläufige Testen, was Entwickler mehrfach am Tag machen, indem sie die Applikation kompilieren und starten. Der Entwickler versucht dann neue Features zu testen. Stürzt die Software dazu ab, wird etwas am Code geändert und derselbe Test beginnt von neuem. Es ist schnell ersichtlich, dass dies für kleine und offensichtliche Fehler funktioniert, aber in größeren und komplexen Systemen unmöglich ist, denn es kann nie jeder Fall nach jeder kleinen Änderung vom Entwickler neu

Abbildung 3.1.: Teststufen

getestet werden. Ähnlich dazu verhalten sich die **Wegwerftests**. Hierbei wird versucht mit verschiedenen Daten das System kurz zu testen um dann zu entscheiden, ob Fehler vorliegen. Am besten und sinnvollsten ist es allerdings **systematische Tests** einzusetzen. Bei diesen werden Testfälle aus der Spezifikation abgeleitet. Diese Testfälle werden ausgeführt und die Ergebnisse und Rahmenbedinungen gut dokumentiert, sodass die Ursache der Fehler auch genau nachvollzogen werden kann.

3.1. Teststufen

In der Softwaretechnik und der Entwicklung in einem Projekt können verschiedene Phasen des Testens unterschieden werden. Abbildung 3.1 zeigt drei Stufen. Die Unit- oder Modultests (Abschnitt 3.1.1), die Integrationstests (Abschnitt 3.1.2) und den Systemtest (Abschnitt 3.1.3). Als vierte Phase könnte man noch den Akzeptanztest erwähnen, der so allerdings nur in Auftragsprojekten zu finden ist. Dieser wird in aller Regel vom Kunden durchgeführt, nachdem das Produkt in der Produktionsumgebung installiert worden ist. Die Richtung des Pfeils beschreibt die Testrichtung. In den meisten Fällen wird bottom-up getestet, also bei den kleinsten Einheiten angefangen, und sofern diese Tests bestanden sind, zur nächsthöheren Ebene gewechselt.

3.1.1. Unittest

Unit Tests (auch Modultests genannt) sind eine Möglichkeit individuelle Module (Units) der Software zu testen. Diese einzelnen Module sind die kleinste testbare Einheit der Applikation. In der objektorientierten Programmierung sind Module meist einzelne Interfaces oder Klassen, in der prozeduralen Programmierung meist einzelne Funktionen. Das Ziel der Unit Tests ist es, Fehler in diesen kleinsten Programmeinheiten ausfindig zu machen. Daher werden Module separiert oder nur mit abhängigen Modulen auf ihre Funktionalität hin getestet um festzustellen, ob sie der Spezifikation genügen. Um Seiteneffekte zu vermeiden wird versucht die einzelnen Testfälle unabhängig voneinander zu gestalten. Diese Tests sind Teil der White Box Tests (Abschnitt 3.2.2) Fehler können in einer frühen Projekt oder Testphase des Projekts aufgedeckt, und nicht erst wenn einzelne Module mit anderen Modulen zur finalen

Applikation zusammengesetzt und verwendet werden. Desweiteren ist es möglich diese als Teil des Regressionstests (Abschnitt 3.2.1) zu verwenden, um zu zeigen, dass sich die Funktionalität durch das Refactoring nicht geändert hat.

3.1.2. Integrationstest

In der Softwareentwicklung beschreibt der Integrationstest die Phase, in der einzelne Module zu der kompletten Software zusammengesetzt werden und so eine größere Codemenge auf Fehler untersucht. Mit Hilfe des Integrationstests wird versucht Fehler ausfinding zu machen, die erst durch das Zusammenspiel verschiedener Module auftreten. Module werden hier als Black Boxes betrachtet, unabhängig vom internen Aufbau und der genauen internen Funktionsweise. Stattdessen werden nur nach außen sichtbare Schnittstellen verwendet um das Zusammenspiel der einzelnen Komponenten zu untersuchen. Zeitlich gesehen tritt diese Phase häufig nach den Modultests ein.

Bei Integrationstests kann man verschiedene Herangehensweisen unterscheiden.

Big Bang Verfährt man nach dem Big Bang Prinzip, so fügt man alle Komponenten auf einmal zusammen und führt anschließend auf dem gesamten System einen Integrationstest durch. Sofern dieser Prozess richtig durchgeführt und die Ergebnisse entsprechend gut dokumentiert werden, kann dieses Verfahren Zeit und Geld sparen.

Bottom-Up Hierbei werden zuerst die Komponenten integriert, die nicht von anderen Komponenten abhängen. Als nächstes werden Komponenten integriert, die auf diesen Basiskomponenten aufbauen. Das System wird also von Grund an aufgebaut. Die Integration der Schichten wird einzeln getestet, so kann direkt festgestellt werden auf welcher Ebene Fehler aufgetreten sind.

Top-Down Analog zum Bottom-Up Verfahren werden bei diesem Verfahren zuerst die Hauptkomponenten integriert und getestet. Allerdings mit dem Nachteil, dass benötigte Komponenten gemockt[1] werden müssen, also noch nicht integrierte aber referenzierte Module mit Beispieldaten, implementiert werden müssen.

3.1.3. Systemtest

Der Systemtest kann als Integrationstest auf höchster Ebene betrachtet werden. Hier werden Kommunikationsprobleme zwischen einzelnen Teilsystemen aufgedeckt. Dies ist zudem

[1] Gemockte Komponenten oder Objekte dienen in der Softwareentwicklung als Platzhalter für echte Objekte innerhalb eines Modultests. In Software existieren Abhängigkeiten zwischen Modulen, um diese dennoch isoliert testen zu können wird mit Hilfe von gemockten Objekten die Umgebung simuliert [Fre09].

der einzige Test der zeigen kann, dass Funktionalitäten fehlen oder nicht der Spezifikation entsprechen [LL10].

3.2. Testmethoden

Neben den in Abschnitt 3.1 vorgestellten Teststufen gibt es weitere Testmethoden, die in der modernen Softwareentwicklung Verwendung finden. Diese können in den beschriebenen Testphasen verwendet werden um die Software auf verschiedene Aspekte hin zu testen. Verschiedene Testmethoden werden im Folgenden erläutert.

3.2.1. Regressionstest

Selten wird ein Projekt von Neuem begonnen, stattdessen wird häufig eine bereits vorhandene Software weiterentwickelt. Die Weiterentwicklung kann bedeuten, dass der Software neue Features hinzugefügt wurden oder vorhandene Fehler in der Sofware beseitigt wurden. Es ist jedoch relativ wahrscheinlich, dass sich Fehler bei der Implementierung neuer Features oder bei der Beseitigung vorhandener Bugs in die Software einschleichen. Der Regressionstest zielt im speziellen darauf ab, Fehler zu finden, die durch solche Modifikationen unabsichtlich hinzugefügt wurden.
Um einen Regressionstest durchführen zu können muss jedoch schon eine lauffähige Version der Software existieren und Soll-Daten müssen entsprechend vorhanden sein. Ein Vorteil dieses Testverfahrens ist es, dass nicht nur initiale Soll-Daten für die Tests verwendet werden können. Stattdessen ist es möglich, sofern das System bereits im Einsatz war, auf echte, realitätsnahe Daten zurückzugreifen und auch diese für den Regressiontest zu verwenden. Hierbei werden die Daten dann sowohl in das neue, modifizierte System als auch in das bestehende eingespielt. Die Ergebnisse werden dann miteinander verglichen. Unterscheiden sich diese, fand eine semantische Änderung statt und die entstandenen Fehler müssen beseitigt werden.
Der Regressionstest kann Fehler in der Spezifikation jedoch nicht aufdecken und ist ebenso nicht in der Lage, bereits in einer vorherigen Iteration der Software vorhande Fehler zu finden. Die Ausführung dieses Tests gilt als positiv, sofern im Ergebnis eine Änderung gegenüber einer vorherigen Version stattgefunden hat.
Regressionstests lassen sich grundsätzlich in allen Teststufen verwenden. Sofern Änderungen an einzelnen Modulen vorgenommen werden, ist es sinnvoll sowohl Unit Tests (Abschnitt 3.1.1), als auch Integrations- (Abschnitt 3.1.2) und Systemtests (Abschnitt 3.1.3) auszuführen. Dadurch wird sichergestellt, dass die Modifkation die Funktionsweise der Software nicht negativ beeinflusst hat.

3.2.2. White Box Test

Bei einem White Box Test wird die innere Struktur des Prüflings genauer unter die Lupe genommen und Teil für Teil getestet. Hierfür muss der Quellcode des Programms ersichtlich sein, andernfalls kann dieser Test nicht durchgeführt werden. Das Ziel eines White Box Tests ist es, eine gewisse, möglichst maximale Codeabdeckung zu erreichen und dabei jede Funktion und möglichst jede Codezeile mindestens einmal ausgeführt zu haben. White Box Tests werden am häufigsten in der Phase der Unit Tests (Abschnitt 3.1.1) verwendet, da hier auf die internte Struktur der einzelnen Module eingegangen werden kann.

3.2.3. Gray Box Test

Gray Box Tests [KGTB07] sollen im Rahmen der testgetriebenen Entwicklung die Vor- und Nachteile von White- (Abschnitt 3.2.2) und Blackbox-Tests (Abschnitt 3.2.4) miteinander vereinen. Diese Gray Box Tests werden, analog zu den White Box Tests, noch von den selben Entwicklern geschrieben, die auch den zu testenden Quellcode verfasst haben. Da der Test allerdings vor der eigentlichen Implementierung geschrieben wird, ist die genaue Funktionsweise des zu testenden Codes noch nicht bekannt, sodass zwangsläufig gegen die Spezifikation getestet wird, analog zu einem Black Box Test. Wie auch bei den White Box Tests, werden diese v.a. in der Phase der Unit Tests (Abschnitt 3.1.1) eingesetzt.

3.2.4. Black Box Test

Bei einem Black Box Test betrachtet man, im Gegensatz zum White Box Test, nicht die genaue Beschaffenheit, sondern nur was nach außen sichtbar ist. Man prüft, ob das Programm der Spezifikation genügt. J. Ludewig und H. Lichter bezeichnen dies als wichtigste Form des Testens, die durch die anderen Tests ergänzt wird [LL10]. Auch dieses Verfahren findet meist auf Unit Tests (Abschnitt 3.1.1) Ebene Verwendung.

3.2.5. End-to-End Testing

> Focus on the user and all else will follow. [Gooa]

Diese Aussage: „Konzentriere dich auf den Nutzer, der Rest wird folgen“ stammt aus der Philosophie von Google und beschreibt auch die Philosophie hinter End-to-End Tests. Zwar sind Unit Tests zum Testen deutlich schneller und verlässlicher als End-to-End Tests, jedoch haben diese den entscheidenden Vorteil einen realen Nutzer zu simulieren. Dies ist die Aufgabe eines End-to-End Tests, die Anwendung systematisiert möglichst so zu benutzen, wie es ein Endanwender am Schluss benutzen wird. End-to-End Tests sind damit in einer späten Phase des Testens anzusehen und eignen sich zum Durchführen von Systemtests (Abschnitt 3.1.3).

Die komplette Applikation wird ausgeführt und fehlende Features können dem Tester hier auffallen.

3.2.6. Testgetriebene Entwicklung

Die Testgetriebene Entwicklung (oder Verhaltensgetriebene Entwicklung) bezeichnet eine Technik in der Softwareentwicklung, bei der das Testen vor der Implementierung der Funktionalität stattfindet [Amr]. Die Testfälle geben in diesem Falle die zu implementierende Funktionalität an, wodurch die gesamte Qualität der Applikation gesteigert werden soll, da es keine Module ohne jeweiligen Testfall gibt.

4. Test Tools

Es existieren bereits einige Tools, die für das Testen von AngularJS Applikationen oder auch zum Testen von Cordova bzw. Ionic Apps verwendet werden können. Einige Tools werden in diesem Kapitel beschrieben und erläutert. Viele dieser Tools sind bereits durch ihren Einsatz in anderen großen Projekten bekannt und haben durch jahrelange Verwendung ihre Tauglichkeit unter Beweis gestellt. Andere Tools sind noch in der Beta, wurden aber speziell für diesen Einsatzzweck entwickelt.

Die noch im Laufe dieses Kapitels beschriebenen Tools lassen sich in verschiedene Aufgabenbereiche gliedern. Tabelle 4.1 zeigt diese Gliederung. Unterscheiden lassen sich hier vor allem Testrunner (Abschnitt 4.3) und Testframeworks (Abschnitt 4.4).

Testrunner sind für die korrekte Ausführung von Tests verantwortlich und bereiten den benötigten Kontext vor. Hierfür laden sie, die für die Testausführung nötigen Module der Applikation. In unserem Fall wird also Nifino und die jeweils zu testenden Module geladen. Zusätzlich werden die Testdateien geladen und das Ergebnis der Testausführung am Ende wieder ausgegben. Diese Tools erlauben es dem Benutzer, die Ausführung der Tests an seine Bedürfnisse anzupassen, bspw. welcher Browser oder welches Framework für die Tests oder die Generierung der Ergebnisreports verwendet werden soll. Allerdings stellen Testrunner keine Assertion[1] Bibliothek zur Verfügung und führen die Applikation nicht selbst aus, sondern starten für diesen Zweck einen Browser.

Typ	Tool
Testrunner	Karma, Testem, Protractor, Nightwatch
Testframework	Jasmine, Mocha, QUnit
Browser Treiber	Selenium
Laufzeitumgebung	PhantomJS, Appium, Ripple

Tabelle 4.1.: Test Tools im Überblick

[1] Assertions lassen sich als Aussagen über den Zustand eines Programms beschreiben. Mit Hilfe dieser kann festgestellt werden ob das Programm fehlerhafte Funktionen besitzt. Sofern diese Assertions in der Testausführung als wahr evaluiert werden, wird kein Fehler entdeckt und der Test ist negativ.

Testframeworks hingegen bieten meist eine Assertion Bibliothek, die verwendet wird um die Testfälle zu implementieren. Ein solches Framework kann dann von einem Testrunner eingebunden werden, der zur Testausführung benötigte Dateien lädt und mit der Ausführung beginnt.

Oft ist es nötig auf DOM Elemente der zu testenden Applikation zugreifen zu können. Vor allem im Falle der End-to-End Tests (Abschnitt 3.2.5), um verschiedene Aktionen auszulösen oder sich durch die Applikation navigieren zu lassen. Da diese jedoch nach außen meist durch einen Browser abstrahiert werden, ist hier ein Browser Treiber nötig der den Zugriff auf die DOM Elemente ermöglicht.

Führt man End-to-End Tests aus, wird die komplette Applikation ausgeführt, daher ist auch die Laufzeitumgebung (Abschnitt 4.5), in welcher diese ausgeführt werden, entscheidend. Diese muss gewisse Features bereitstellen um die Applikation korrekt ausführen zu können. Für diesen Fall gibt es verschiedene vorhandene Tools und Möglichkeiten.

Zunächst werden aber die Anforderungen definiert, die Nifino an ein solches Tool stellt um einen reibungslosen Testablauf zu garantieren.

4.1. Anforderungen an die Test Tools

Nifino unterstützt mehrere Features, die alle auf verschiedenen Ebenen zu testen sind. In Kapitel 3 [Testverfahren] wurden bereits verschiedene Testmöglichkeiten und Teststufen beschrieben. Wir wollen ein Framework für jede dieser Teststufen finden, um so eine möglichst fehlerfreie Benutzung von Nifino garantieren zu können.
AngularJS Applikation lassen sich vor allem mit Unit Tests (Abschnitt 3.1.1) und End-to-End Tests (Abschnitt 3.2.5) testen. Erstere eignen sich um die Modultests der in Abschnitt 3.1 vorgestellten Testphasen abzudecken. Die Ausführung des End-to-End Tests testet neben der kompletten Applikation aus Sicht des Benutzers auch das Zusammenspiel der einzelnen Komponenten in der finalen Anwendung. Damit eignen sich diese zur Verwendung als Integrations- und Systemtests, sodass alle Testphasen mit Hilfe der beiden Testmethoden abgedeckt sind.
Eine einzige Testsuite, die verwendet werden kann um sowohl Unit- als auch End-to-End Tests auszuführen existiert nicht. Wir suchen Tools, die uns die Ausführung von Tests für diese beiden Möglichkeiten erlauben.

Das Testen nativer Android und IOS Anwendungen ist inzwischen gut dokumentiert und es gibt eine Vielzahl entsprechend geeigneter Tools, um diese auf ihre Funktionalität hin zu testen. Allerdings handelt es sich bei Nifino um eine HybridApp, also eine in Javascript und HTML5 geschriebene App, die in einen nativen Webcontainer eingebettet als Applikation

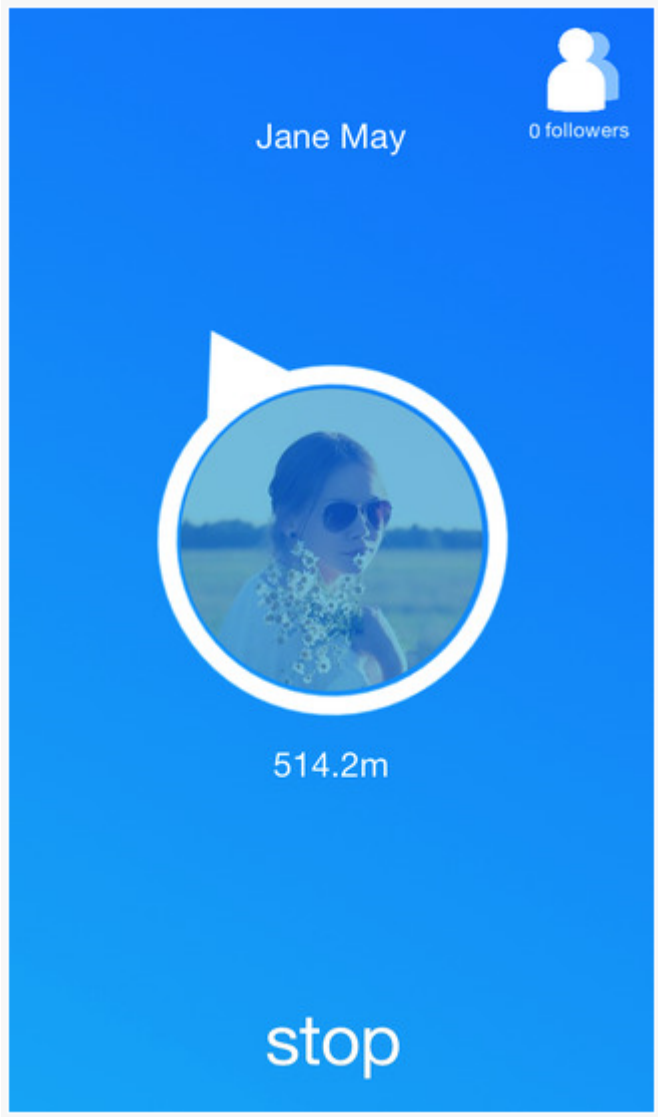


Abbildung 4.1.: Nifino Tracking [Nif]

deployed wird. Da der UI- und Logikcode der Anwendung in HTML und Javascript geschrieben ist, und dieser nicht entsprechend auf jede Plattform angepasst werden muss, helfen uns die nativen Android und IOS Testmethoden nicht weiter. Daher suchen wir ein Framework, das das Testen von Hybrid Anwendungen ermöglicht.

Da Nifino auf Basis von Ionic und damit Cordova und AngularJS entwickelt wurde, besteht dieses aus verschiedenen Controller, Direktiven, Services und Filter, die alle auf Unittest Ebene getestet werden können. Unittest Ebene heißt in diesem Fall, eben diese Elemente (Controller, Services, Filter und Direktiven) gesondert, unabhängig externer Abhängigkeiten auf Fehler zu untersuchen. Das verwendete Testframework muss zum einen AngularJS Komponenten, zum

anderen asynchrone Funktionen testen können und sollte im Idealfall auch das Testen von textitPromises[2] ohne großen Aufwand ermöglichen.

Bisher existiert eine Menge an Dokumentationen, Büchern und Anleitungen um AngularJS Applikationen auf Unitbasis Ebene zu testen. In Unit Tests ist es wichtig externe Abhängigkeiten zu mocken und die Tests unabhängig von diesen zu gestalten. Es wäre erstrebenswert ein Framework zu haben, dass das Mocken von Funktionen erlaubt und vereinfacht. Das Mocken externer Komponenten ist notwendig, sodass der Login Prozess bspw. ohne Verbindung zum Backend ausgeführt werden kann. Allerdings ist diese Funktioalität nur bedingt von Nöten, da AngularJS das `ngMock` Modul zur Installation bereitstellt. Dieses Modul erlaubt es Abhängigkeiten für die Testausführung zu mocken und einige von AngularJS bereitgestellte Komponenten zu injecten[3] um diese während der Testausführung besser überwachen und die Testfälle überhaupt ausführen zu können. AngularJS setzt unter anderem auf Direktiven. Diese werden in Nifino bspw. zur Auswahl des Ortes für Beacons verwendet und bestehen aus Templates die zur Laufzeit kompiliert werden. Um einen vergleichbaren Sollwert zu erzeugen sollten Hooks oder Präprozessoren unterstützt werden. Diese Hooks oder Präprozessoren erlauben es eigenen Code in die Ausführung zu integrieren um bspw. für das Testen notwendige Vorarbeit leisten zu können. Dies ist bspw. nötig um Templates vor der Ausführung der Testfälle zu kompilieren. Dieser kompilierte Code kann dann in den Testfällen verwendet werden.

Die AngularJS Komponenten sind zwar ähnlich, jedoch nicht identisch zu testen. Die genauen Unterschiede der einzelnen Elemente und inwiefern dies beim Testen berücksichtigt werden muss, wird in Abschnitt 5.2.1 genauer beschrieben. In diesem Kapitel werden nur Frameworks als AngularJS kompatibel beschrieben, die es erlauben sowohl Controller, Direktiven, Services und auch Filter zu testen.

Neben Unit Tests wird Nifino auch End-to-End Tests unterzogen, im Idealfall sollte das verwendete Testframework sich für diese beiden Einsatzzwecke eignen, um den Organisationsaufwand bzw. den Aufwand Testfälle zu entwicklen zu minimieren.
Continuous Integration erfreut sich immer größerer Beliebtheit. In Zukunft soll es möglich sein, den Testprozess in den Buildprozess zu integrieren, sodass die gewählten Tools nach Möglichkeit auch für den Einsatzzweck der Continuous Integration eingesetzt werden können. Insgesamt sollte der Testprozess möglichst eigenständig ablaufen, sodass dem Entwickler mehr Zeit für die eigentliche Arbeit, der Implementierung der Funktionalität und Testfälle bleibt.
Da Nifino hauptsächlich auf Mobilgeräten läuft, stehen dort weitere Sensoren und Features zur

[2] Promises ersetzen in Javascript asynchrone Callbacks. Jedoch lassen sich Promises leichter lesen und debuggen und bieten daher einige Vorteile gegenüber traditionellen Callbacks.
[3] In Testfällen ist es nötig auf externe Abhängigkeiten zugreifen zu können. Diese Möglichkeit wird von der `angular.mock.inject` Funktion geboten.

Verfügung die sich nicht ohne weiteres testen lassen. Nifino erlaubt es Nutzern sich gegenseitig zu tracken. Hierfür werden bspw. Lagesensor und GPS Sensoren verwendet. Letztere lassen sich dank aktueller HTML Standards zwar im Browser abfragen, jedoch stellt dieser keinen Lagesensor zur Verfügung. Um dennoch diese Funktionalität zu testen wäre eine Laufzeitumgebung von Vorteil, aus welcher Nifino diese Sensordaten auslesen kann. Abbildung 4.1 zeigt einen Anwendungsfall von Nifino, indem gerade eine Person getrackt wird. Sofern sich diese Person oder man sich selbst bewegt, soll die Richtung, in die der Pfeil zeigt, sowie die Distanz zur Zielperson entsprechend aktualisiert dargestellt werden. Dies kann ohne Lage- oder Positionsdaten nicht funktionieren.
Nifino verwendet, wie die meisten anderen mobilen Anwendungen Benachrichtigungen. Diese lassen sich nicht ohne weiteres Testen, da Browser keine Unterstützung für diese Features bieten. Ebenso wenig wie für die Android und Apple spezifischen Pushsysteme GCM (Google Cloud Messaging) und APN (Apple Push Notification Service). Für erstere sind Google Play Services nötig[Goob], welche in Webumgebungen nicht vorhanden sind. Allerdings sollen auch diese Features von Nifino getestet werden.

4.2. Browser Treiber

Auf Testrunner oder Testframework Ebene ist es oft nicht möglich auf den Inhalt einer im Browser geöffneten Webseite zuzugreifen oder die DOM Elemente zu untersuchen. Da jeder Browser die Darstellung anders handhabt ist es auch nicht ohne weiteres möglich jeden Browser zu unterstützen. Um diesem Problem entgegen zu wirken gibt es Browser Treiber. Diese bieten eine Browser unabhängige API um diesen zu steuern und um auf die Inhalte zugreifen zu können. Daher verwenden viele Testrunner solche Browser Treiber, auch um eine Vielzahl an verschiedenen Browsern zu unterstützen.

4.2.1. Selenium

> Selenium automates browsers. That's it! [Sel]

Mit diesen Worten wird Selenium auf der eigenen Entwickler Homepage beschrieben. Und genau das tut Selenium auch, es automatisiert die Steuerung von Browsern, sodass Entwickler einfacher mit diversen Browsern arbeiten und ihr Produkt in diesen Browsern testen können. Selenium erlaubt es den Entwicklern die verschiedenen Elemente einer App oder einer beliebigen Webseite, abstrahiert auf DOM Basis zu manipulieren, Daten einzugeben oder Schaltfelder anzuklicken.
Wichtig für uns ist allerdings die Möglichkeit die der Selenium WebTreiber bietet. Dieser nimmt Befehle, bspw. zum Testen von Webseiten entgegen und leitet diese an den Browser weiter[Sel]. Dies geschieht durch einen spezifischen Browser Treiber, der dann in den meisten Fällen auch den jeweiligen Browser dazu startet (bspw. Chrome oder Firefox). Testrunner

Abbildung 4.2.: Selenium

wie bspw. Karma (Abschnitt 4.3.1), Testem (Abschnitt 4.3.2) oder Protractor (Abschnitt 4.3.3) können diese API verwenden um mit dem Browser kommunizieren zu können, sodass sich Selenium in viele Testprozesse ohne großen Aufwand integrieren lässt. Da bei der Ausführung von Unit Tests in aller Regel kein Zugriff auf DOM Elemente einer Seite stattfindet, wird dort meist auf Selenium verzichtet. Diese Möglichkeit wird bei End-to-End Tests benötigt. Daher setzen die meisten End-to-End Testing Tools wie Protractor, aber auch Laufzeitumgebungen wie Appium (Abschnitt 4.5.2), auf Selenium. Durch die Unterstützung der Community gibt es einen WebTreiber für die meisten Browser, unter anderem Crosswalk oder Microsofts Edge. Diese sind jedoch nicht offiziell und daher nicht von Selenium selbst enwickelt, diese werden als Third Party Treiber bezeichnet.
Abbildung 4.2 zeigt hierbei die Funktionsweise von Selenium. Es steht zwischen dem eigentlichen Browser bzw. dessen Treiber und dem eigentlichen Testscript und leitet die Informationen beider Parteien an die jeweils andere weiter.
Tatsächlich bietet Selenium aber noch deutlich mehr als nur einen Webtreiber, der zum Ansteuern verschiedener Seiten in diversen Browsern verwendet werden kann. Es bietet unter anderem eine Entwicklungsumgebung zum Aufnehmen von Testsequenzen, Gesten und Eingaben für UI Tests, die dann automatisiert wiederholt werden können, ohne dabei eine Scriptsprache zum Schreiben von echten Tests lernen zu müssen. Desweiteren bietet Selenium neben der Unterstützung von Javascript auch noch weitere populäre Programmiersprachen an, in denen Tests implementiert werden können. Hierzu zählen bspw. Java, Perl oder Python. Weitere Features sind Selenium Grid, welcher es erlaubt Tests auf Remote Browsern auszuführen. Das bedeutet die Teststeuerung, bspw. durch einen Testrunner, wird auf einem anderen Gerät ausgeführt als der eigentliche Test selbst.

4.3. Testrunner

Um in der Vergangenheit Webseiten oder Web-Applikationen auf Fehler zu untersuchen war es die Aufgabe des Entwicklers manuell den Browser zu öffnen, die URL anzusteuern und anschließend die Tests manuell auszuführen. Nach der Durchführung musste dieser entscheiden, ob während des Testlaufs Fehler aufgetreten sind oder nicht. Entsprechend mussten Änderungen vorgenommen und die Prozedur von Neuem begonnen werden. Da die manuelle Ausführung langwierig und v.a. für große Projekte fehleranfällig war, wurden Testrunner entwickelt. Diese haben die Aufgabe dem Entwickler genau diese Schritte abzunehmen und die Testausführung automatisiert anzustoßen, sobald sich bspw. der Quellcode einer zu testenden

Datei manipuliert wurde. Diese Testrunner bieten allerdings kein eigentliches Testframework, aber bieten Möglichkeit solche einzubinden.

4.3.1. Karma

Einer dieser Testrunner ist Karma (ehemals Testacular) [Kara], der hauptsächlich verwendet wird, um Tests auf Unit Ebene auszuführen. Dennoch ist es grundsätzlich möglich auch End-to-End Tests für AngularJS Applikationen damit ausführen zu lassen. Von diesem Einsatzzweck raten die Entwickler von Karma jedoch ab[Karb]. Zum einen da Karma für Unit Tests entwickelt wurde und für die Verwendung in End-to-End Tests ein Plugin nachinstalliert werden muss. Zum anderen aber auch weil es Testrunner gibt, die sich für diese Einsatzzwecke besser eignen und die von den Entwicklern dafür empfohlen werden. Karma kann bspw. für die testgetriebene Entwicklung (Abschnitt 3.2.6) eingesetzt werden. Ein weiteres Einsatzgebiet von Karma ist in der Continuous Integration und beschränkt sich hierbei nicht auf die Test von AngularJS Applikationen, sondern kann auch als Testrunner in bspw. jQuery Projekten Verwendung finden.
Zusätzlich bietet Karma dem Nutzer die Möglichkeit, die Tests in verschieden Internetbrowsern, wie bspw. Google Chrome, Mozilla Firefox, dem Internet Explorer oder auch in PhantomJS (Abschnitt 4.5.1) ausführen zu lassen. Dies ist vor allem für Webseiten von großem Interesse, da sie von allen möglichen Browsern aufgerufen werden können.
Um Tests ausführen zu können wird zusätzlich ein Framework benötigt. Standardmäßig sind keine Plugins für diese installiert, lassen sich jedoch als NPM[4] Module nachrüsten [Karc]. Sofern vorhandene Plugins installiert sind bietet Karma Unterstützung für Jasmine (Abschnitt 4.4.1), Mocha (Abschnitt 4.4.2) und auch QUnit (Abschnitt 4.4.3). Neben Testframeworks können, ebenfalls als NPM Modul, auch Reporter installiert werden, die den aktuellen Fortschritt der Testausführung darstellen oder das Ergebnis für spätere Betrachtung serialisieren können.
Vergleichbar mit der Unterstützung verschiedener Testframeworks ist auch die Unterstützung für Präprozessoren, die als NPM Module nachinstalliert werden können [Kard]. Präprozessoren werden auf Dateien ausgeführt bevor diese an den Testfall weitergegeben werden. Mit Hilfe dieser lassen sich bspw. Dateien für Coverage Tests vorbereiten oder HTML in Javascript Strings konvertieren, um diese an ein Script weitergeben zu können. Zusätzlich bietet Karma die Möglichkeit Dateien nach Manipulation zu beobachten. Stellt Karma eine Modifikation fest, kann die Ausführung der Testfälle erneut automatisiert angestoßen werden. Hooks werden zwar nicht direkt von Karma unterstützt, lassen sich jedoch bedingt in Karma einfügen, indem die Funktionen beinhaltenden Dateien an den richtigen Stellen in der Konfigurationsdatei eingetragen und damit vor oder nach den Tests geladen und ausgeführt werden.
Karma wurde von den Entwicklern hinter AngularJS implementiert. Um möglichst vielen Entwicklern die Verwendung zu ermöglichen existiert Karma für Windows, Linux und Apple

[4] NPM steht für Node Package Manager und ist ein Paketmanager für Node.js. Zu Anfang des Jahres 2016 gab es dort über 200.000 Pakete.

OS X Systeme und kann daher von nahezu jedem Entwickler verwendet werden.
Karma ist letztlich ein nützlicher Testrunner der unabhängig von der eigentlich Zielplattform für Services, Controller, Direktiven und Filter im AngularJS Stil eingesetzt werden kann und außerdem von seiner Erweiterbarkeit durch die Unterstützung von Plugins profitiert. Aktuell existieren über 450 verschiedene NPM Module um die Funktionalität von Karma zu erweitern [Kare].

4.3.2. Testem

> Got Scripts? Test'em! [Tesa]

Testem ist, wie auch Karma (Abschnitt 4.3.1) ein Testrunner und kann als Alternative zu Karma angesehen werden. Tatsächlich unterscheiden sich die beiden Tools kaum. Testem kann ebenfalls für die testgetriebene Entwicklung und die Continuous Integration eingesetzt werden. Zudem bietet Testem Unterstützung für die meisten Browser, wie bspw. Chrome, Firefox, Internet Explorer oder PhantomJS. Bei der Wahl der Testframeworks gibt es hier eine große Auswahl, alle hier vorgestellten Testframeworks von Jasmine (Abschnitt 4.4.1), über Mocha (Abschnitt 4.4.2) bis hin zu QUnit (Abschnitt 4.4.3) werden nativ von Testem unterstützt, ohne dafür Plugins installieren zu müssen.
Dennoch kann auch bei Testem die Funktionalität erweitert werden, indem Adapter für andere Frameworks installiert und verwendet werden. Neben Filewatchern, die Dateien auf Modifkationen überwachen und ggf. den Test erneut ausführen, sofern der zu testende Code manipuliert wurde, bietet Testem Unterstützung für Präprozessoren. Hooks können, analog zu Karma, bedingt eingefügt werden, werden jedoch nicht nativ unterstützt. Testem lässt dem Entwickler die freie Wahl des Betriebssystems, unterstützt werden Windows, Linux und OS X.

4.3.3. Protractor

> Protractor is an end-to-end test framework for AngularJS applications. Protractor runs tests against your application running in a real browser, interacting with it as a user would [Proa]

Mit diesen Worten beschreiben die Entwickler selbst Protractor. Das Tool selbst wird zwar von den Entwicklern als Testframework betitelt, entspricht jedoch eher einem Testrunner, da Protractor, wie auch Karma (Abschnitt 4.3.1) oder Testem (Abschnitt 4.3.2) keine Assertion Bibliothek für das eigentliche Testen bereitstellt. Allerdings liefert Protractor bei der Installation Jasmine (Abschnitt 4.4.1) als Testframework mit, weshalb die Bezeichnung der Entwickler vertreten werden kann.
Protractor eignet sich für Verwendung mit End-to-End Tests (Abschnitt 3.2.5). Als Einsatzzweck von Protractor lässt sich, wie auch bei Karma und Testem, die testgetriebene Entwicklung

nennen. Im Gegensatz zu diesen beiden bietet Protractor jedoch keine native Unterstützung für die Continuous Integration, lässt sich jedoch mit Aufwand, durch den Einsatz von Taskrunnern oder Scripten, auch für diese nutzen.
Um eine Applikation oder Webseite einem End-to-End Test zu unterziehen, muss diese vollständig in einem Browser ausgeführt werden. Außerdem ist es zum Teil notwendig, auf die DOM Elemente der Webseite zugreifen zu können, um die Navigation durch die Applikation zu ermöglichen. Damit auf diese Elemente wieder zugegriffen werden kann benutzt Protractor Selenium (Abschnitt 4.2.1) als Browser Treiber. Daher ist die Unterstützung für verschiedene Browser von Selenium und nicht von Protractor selbst abhängig.
Wie bereits erwähnt installiert Protractor Jasmine standardmäßig als Testframework und bietet nur bedingt Unterstützung für andere Testframeworks wie bspw. Mocha. Allerdings ist die Unterstützung seit 2013 eingeschränkt [Prob] und die Verwendung daher nicht mehr empfehlenswert.
Plugins werden auch von Protractor unterstützt, und im Gegensatz zu Testem und Karma, wird native Unterstützung für Hooks geboten, sodass Code vor, nach oder zwischen einzelnen Testfällen oder dem kompletten Testlauf ausgeführt werden kann [Proc]. Bspw. existieren Plugins die es erlauben, Screenshots der Applikation nach einzelnen Testfällen aufzunehmen.
Wie auch die vorherigen beiden Runner kann Protractor auf Windows, Linux und OS X verwendet werden.

4.3.4. Nightwatch

> Browser automated testing done easy. [Nig]

Nightwatch ist ein komplettes Framework bestehend aus einem Testframework und einem Testrunner und benötigt daher keine weiteren Tools oder Frameworks um Tests ausführen zu können. Das Framework kann als Alternative zu Protractor (Abschnitt 4.3.3) und Jasmine (Abschnitt 4.4.1) angesehen werden, da sich Nightwatch auch in erster Linie für den Einsatz von End-to-End Tests eignet. Nightwatch erlaubt, dank eigenem Testframework, den Einsatz ohne externes Testframework.
Im Gegensatz zu Protractor lässt sich Nightwatch direkt sowohl für die testgetriebene Entwicklung, als auch für die Continuous Integration einsetzen. JUnit XML Reporter sind standardmäßig integriert, sodass sich der Test Prozess in Buildprozesse gängiger Continuous Integration Systeme wie Teamcity oder Jenkins integrieren lassen [Nig].
Vergleichbar mit Protractor ist der Einsatz von Selenium (Abschnitt 4.2.1) zur Browserautomatisierung. Der Selenium Server kann, wie auch bei Nightwatch, lokal oder remote auf einem Server ausgeführt werden. Durch den Einsatz von Selenium ist ein direkter Zugriff auf die DOM Elemente möglich, sodass nicht auf AngularJS basierende Applikationen oder Webseiten ebenfalls getestet werden können. Damit hängt die Unterstützung für verschiedene Browser weniger von Nightwatch als von Selenium ab.
Da Nightwatch im Vergleich zu den anderen vorgestellten Testrunnern ein Testframework beinhaltet, bietet Nightwatch eine integrierte Assertion Bibliothek. Die verwendete Bibliothek

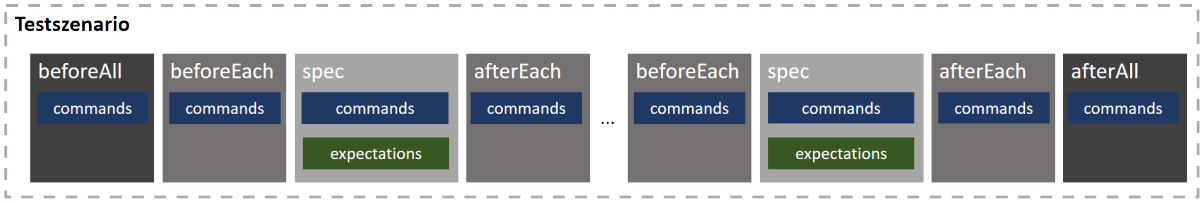


Abbildung 4.3.: Jasmine Test Szenario

lässt sich erweitern oder überschreiben. Auch wenn hier nicht die Auswahl an existierenden Plugins, wie bspw. bei Karma (Abschnitt 4.3.1) zur Verfügung steht, bietet Nightwatch Unterstützung für Erweiterungen.
Seit Version 0.6 [Tesb] ist es möglich Nightwatch auch für Unit Tests (Abschnitt 3.1.1) einzusetzen. Da dieses Feature jedoch noch ziemlich neu ist, können alternative Unit Testrunner und dedizierte Frameworks hier deutlich mehr Funktionen bieten. Die Verbindung und damit Verwendung von Selenium ist in diesen Fällen weder nötig noch möglich, was explizit in den Einstellungen zum jeweiligen Unit Test vorgenommen werden muss.
Da NightwatchJS auf Node.js basiert, kann dieses Framework auf allen gängingen Betriebssystemen eingesetzt werden.

4.4. Testframeworks

Die Verwendung von Testframeworks ist unerlässlich. Einige Testsuites bringen bereits ihre eigenen Frameworks mit. Testframeworks stellen eine Assertion Bibliothek und eine API bereit mit der Testfälle implementiert werden können. Um die Testausführung möglichst zu automatisieren, können die Testframeworks von den Testrunnern (Abschnitt 4.3) eingebunden werden. Im Folgenden werden die in Tabelle 4.1 genannten Testframeworks beschrieben und auf ihre Tauglichkeit bezüglich der Anforderungen untersucht.

4.4.1. Jasmine

Jasmine ist ein Testframework für die Test- oder Verhaltensgetriebene (Abschnitt 3.2.6) Entwicklung für Javascript. Es basiert weder auf einem Browser, noch auf einem anderen Framework oder den DOM Elementen einer Webseite und eignet sich daher für Tests an normalen Webseiten, AngularJS oder sonstigen Projekten, in denen Javascript ausgeführt wird [Jas]. Desweiteren zeichnet sich Jasmine durch die einfache Verwendung und die gute Dokumentation aus.
Jasmine kann in Kombination mit den vorgestellten Testrunnern Karma (Abschnitt 4.3.1), Testem (Abschnitt 4.3.2) oder Protractor (Abschnitt 4.3.3), eingesetzt werden um die Testausführung weiter zu automatisieren. Durch den Einsatz mit Karma und Protractor ist Jasmine außerdem nicht auf einen Testtyp festgelegt, sondern kann sowohl für Unit Tests (Abschnitt 3.1.1),

als auch für End-to-End Tests (Abschnitt 3.2.5) eingesetzt werden. Mit Hilfe von Jasmine lassen sich sowohl synchrone, als auch asynchrone Funktionen testen und damit die häufig in Javascript verwendeten Promises.
Abbildung 4.3 zeigt den Aufbau eines Jasmine Testszenarios. Ein solches Szenario besteht hierbei aus einem `describe` Block, der das Testszenario beschreibt. Solch ein Szenario kann optional eine `beforeAll` Funktion und ebenfalls optional eine `afterAll` Funktion besitzen. Diese werden jeweils nur einmal ausgeführt und zwar vor und nach den einzelnen Testfällen (specs). Ebenfalls optional werden vor den jeweiligen Testfällen (specs) die `beforeEach` und `afterEach` Funktionen ausgeführt.
Da in Unit Tests externe Abhängigkeiten gemockt werden und nicht den vollen Funktionsumfang bieten, ist es in manchen Testfällen hilfreich zu wissen, mit welchen Parametern oder wie oft eine Funktion aufgerufen wurde. Um dies zu ermöglichen führt Jasmine das Prinzip der Spies ein. Diese Spione können auf eine Funktion angesetzt werden, sodass der Aufruf abgefangen oder weitergeleitet, aber überprüft werden kann. In den Testfällen kann dementsprechend auf diesen Aufruf reagiert werden.
Jasmine bietet zudem weitere Funktionalitäten um den Startprozess der Jasmine Umgebung den eigenen Bedürfnissen anzupassen oder um eigene Gleichheit von Objekten zu definieren. Zudem können in Jasmine eigene Reporter verwendet werden und das Framework seinen Bedürfnissen, bspw. in einer Continuous Integration Umgebung, einzurichten.
Jasmine ist Open Source und steht damit allen Interessenten zur Verfügung.

4.4.2. Mocha

> Mocha is a feature-rich JavaScript test framework running on Node.js and the browser, making asynchronous testing simple and fun. [Moca]

Mocha ist ein auf Node.js basierendes Testframework. Die Entwickler selbst bezeichnen es als „einfache und spaßige“ Methode asynchrone Tests auszuführen [Moca]. Insgesamt unterscheidet sich Mocha nicht allzu sehr von Jasmine (Abschnitt 4.4.1). Mocha kann für das Testen von AngularJS basierenden und AngularJS fremden Webseiten verwendet werden. Hierbei lässt sich Mocha mit den meisten vorgestellten Testrunnern verwenden, jedoch ist die Unterstützung von Protractor (Abschnitt 4.3.3) für Mocha eingeschränkt.
Wie Jasmine eignet sich Mocha um Testfälle sowohl synchron als auch asynchron ausführen zu lassen. Ebenso wird die Unterstützung Promises zu testen explizit angegeben [Moca], auch Unit Tests und End-to-End Tests lassen sich ebenfalls mit Hilfe dieses Testframeworks untersuchen. Der Aufbau der Testszenarien unterscheidet sich nicht wesentlich von dem in Abbildung 4.3 Beschriebenem, weshalb hier nicht näher darauf eingegangen wird.
Im Vergleich zu Jasmine fehlt Mocha aber der durchaus nützliche Support für Spies, dieser lässt sich zwar mit Hilfe von Third-Party Bibliotheken nachinstallieren, die Funktionalität wird aber nicht standardmäßig mitgeliefert [Mocb]. Mocha selbst bietet keine integrierte Assertion Bibliothek, stattdessen muss eine externe Bibliothek eingebunden werden. Hierbei werden Bibliotheken wie bspw. `should.js`, `expect.js` oder `chai`[Mocc] unterstützt.

Mocha bietet darüberhinaus einige weitere Features, um Testern ihre Arbeit zu erleichtern. Langsame oder lang andauernde Tests können hervorgehoben werden und wie bei Jasmine lassen sich Timeouts für die asynchronen Testfälle angeben. Zudem werden von Mocha Filewachter bereitgestellt, welche Test erneut ausführen, sofern die zu überwachenden Dateien manipuliert werden. Zusätzlich können Hooks definiert werden, die zum Vorbereiten oder Aufräumen eines Testfalls verwendet werden können.
Wie viele weitere Testframeworks ist Mocha ein Open-Source Projekt und steht damit allen Interessenten zur Verfügung.

4.4.3. QUnit

> QUnit is a powerful, easy-to-use JavaScript unit testing framework. [QUn]

QUnit ist ein weiteres Javascript Test Framework, dass von vielen JQuery Projekten wie jQuery selbst, jQuery UI und jQuery Mobile im Testverfahren zum Einsatz kommt. Mit QUnit lässt sich jeder Javascript Code testen, weshalb QUnit selbst verwendet werden kann um die eigene Funktionalität in Tests zu validieren [QUn]. Damit kann das Framework im Testprozess von AngularJS basierten Projekten Verwendung finden. Jedoch werden Abhängigkeiten, nicht wie bei Jasmine automatisch aufgelöst, stattdessen müssen diese explizit mit Hilfe der `angular.injector` Funktion aufgelöst werden, wodurch das Testen von AngularJS Projekten aufwendiger ausfällt.
QUnit lässt sich mit den vorgestellten Testrunnern Karma (Abschnitt 4.3.1) und Testem (Abschnitt 4.3.2) kombinieren um eine möglichst automatisierte Testumgebung einsetzen zu können. Ergebnisse können direkt an Continuous Integration System, wie bspw. Jenkins, weitergegeben werden.
Wie bei Jasmine (Abschnitt 4.4.1) und Mocha (Abschnitt 4.4.2) lassen sich mit QUnit asynchrone Testfälle ausführen. Promises sind mit Jasmine oder Mocha besser zu testen als mit QUnit. Diese Funktionalität lässt sich in QUnit allerdings durch Plugins erweitern. Damit unterstützt QUnit neben Erweiterungen die Möglichkeit Hooks zu definieren, die nach, vor oder zwischen Modultests ausgeführt werden können, um die Testergebnisse in einer Continuous Integration zu behandeln. Testergebnisse können erzeugt und als HTML Seite ausgegeben werden.
Wie Jasmine und Mocha steht QUnit allen Interessenten unter einer Open Source Lizenz frei zur Verfügung.

4.5. Laufzeitumgebungen

Beim Testen auf Unit- oder Modulebene ist die Laufzeitumgebung nur indirekt ausschlaggebend, da diese Tests möglichst unabhängig von der Umgebung und externen Abhängigkeiten ausgeführt werden sollen. Beim End-to-End Testen ist es jedoch wichtig, dass die Applikation

als Ganzes mit allen externen Komponenten und Abhängigkeiten funktioniert. Die Testumgebung soll in der Lage sein die Applikation so auszuführen, wie es später beim Benutzer der Fall sein wird, im Idealfall also wie ein Smartphone. Nifino setzt auf verschiedene Sensoren, die in mobilen Endgeräten verbaut sind, so wird der GPS Sensor verwendet um die grobe Lokalisierung der Person durchzuführen. Durch Lage- und Rotationssensoren kann zudem die aktuelle Ausrichtung des Geräts festgestellt und die Navigation entsprechend angepasst werden. Zudem werden Benutzer Nachrichten empfangen, auch diese Funktionalität der Benachrichtigungen und Push-Notifications soll getestet werden. Hierbei gibt es erneut verschiedene Möglichkeiten und Frameworks auf die gesetzt werden kann, die diese Anforderungen ganz oder in Teilen erfüllen.

4.5.1. PhantomJS

PhantomJS ist ein *Headless*[5] Browser, der sich sowohl für Unit- (Abschnitt 3.1.1) als auch für End-to-End (Abschnitt 3.2.5) Tests einsetzen lässt und eine scriptbare Javascript API mit sich bringt. Der Begriff PhantomJS tritt häufig im Zusammenhang mit dem Testen von mobilen Applikationen auf, vor allem aber im Bereich der Unit Tests (Abschnitt 3.1.1). Firmen und Entwickler von größeren Projekten, wie bspw. Twitter [Lan], setzen PhantomJS in ihrer Testumgebung ein. Der große Vorteil ist, dass PhantomJS sich nicht um die Darstellung kümmern muss und daher deutlich schneller agiert als bekannte Browser wie Firefox oder Chrome.
PhantomJS kann jedoch, wie viele andere Tools, über die Kommandozeile angesprochen werden, zeichnet sich aber vor allem dadurch aus, dass er headless agiert und sich damit ideal für die kontinuierliche Integration eignet. Eine Einrichtung mit CI Systemen wie Jenkins oder TeamCity ist gut dokumentiert. Der Browser lässt sich mit einer Vielzahl an Testrunnern und Testframeworks, im Fall von Twitter mit QUnit (Abschnitt 4.4.3), aber auch mit Karma (Abschnitt 4.3.1), Testem (Abschnitt 4.3.2), Mocha (Abschnitt 4.4.2), Jasmine (Abschnitt 4.4.1) oder vielen weiteren verwenden.
Zudem bietet PhantomJS noch einige Besonderheiten, so bietet er die Unterstützung um Aufzeichnungen der Seite als Bild abzuspeichern, welche später betrachtet werden können. Ebenso eignet sich PhantomJS um die Netzwerklast zu überwachen oder Seitenautomaisierung zu verwenden. PhantomJS ermöglicht es jedoch nicht benötigte Sensoren zu emulieren, die für das End-to-End Testen notwendig sind.
Allerdings eignet sich PhantomJS nicht um Sensoren oder virtuelle Umgebungen, wie ein mobiles Endgerät, zu simulieren. Dank Unterstützung neuester Webstandards kann die Position abgefragt werden. Rotations- oder Lagesensordaten sind nicht virtuell vorhanden.
PhantomJS lässt sich auf den meist genutzten Betriebssystemen installieren und verwenden,

[5] Headless Software kann auf Geräten ohne grafische Ausgabe eingesetzt werden. Die Steuerung kann über andere Schnittstellen, bspw. das Netzwerk erfolgen.

auf der Downloadseite stehen Binaries für Windows, Linux, sowie Mac OS X und FreeBSD bereit [Pha].

4.5.2. Appium

Appium bietet eine Laufzeitumgebung für die Testumgebung, welche speziell, sowohl für native und hybride, mobile Web Anwendungen ausgelegt ist. Die Entwickler hinter Appium verfolgen hierbei die Philopsophie [Lip14], dass eine App für den Testprozess nicht neu kompiliert werden muss, also genau die Anwendungsdatei die später in einem Appstore geladen werden soll, wird für die Testausführung verwendet. Ein weiterer Punkt ist, dem Entwickler bzw. Tester kein Framework aufzubinden. Stattdessen soll dieser in der Lage sein sich seine benötigten und verwendeten Frameworks und Programmiersprachen völlig frei und Appium unabhängig aussuchen zu können. Die Applikation läuft hierbei nativ auf dem Gerät, mit Hilfe des Webtreibers kann in den unterstützten Sprachen mit der Anwendung bzw. der WebView kommuniziert werden. Appium versucht eine standardmäßige und projektübergreifende Automatisierungs Spezifikation zu entwickeln und mit Hilfe der Open Source Community zu wachsen und das Projekt stetig voran zu treiben. So kann Appium in Verbindung mit den Frameworks Jasmine (Abschnitt 4.4.1), Mocha (Abschnitt 4.4.2) oder QUnit (Abschnitt 4.4.3) verwendet werden. Neben dem Support für Node.js und Javascript eignet sich Appium auch zur Verwendung mit anderen Programmiersprachen, wie bspw. Python, Java oder Perl. Protractor (Abschnitt 4.3.3) bietet zwar keine offizielle Unterstützung für Appium, kann jedoch diese als Laufzeitumgebung für Tests verwenden und die Tests darin ausführen. Um die Verbindung mit der zu testenden Applikation herzustellen wird erneut Selenium (Abschnitt 4.2.1) verwendet. Appium als Testautomatisierungstool eignet sich vor allem für End-to-End Tests. Zu diesem Zweck wurde es entwickelt und bietet Support für native Android und IOS UI Automatisierungstechniken; grundsätzlich lassen sich jedoch auch Unit Tests in Appium ausführen.
Ein großer Vorteil von Appium ist, dass die Testausführung auf einem physischen Gerät oder nativen Emulator stattfinden kann. Damit lassen sich die Probleme von nicht vorhandenen Sensoren oder das Problem der fehlenden Device ID lösen und alle Funktionalitäten die ein Smartphone bieten kann, können damit getestet werden. Emulieren lassen sich fehlende Sensoren allerdings auch unter der Verwendung von Appium nicht. Änderungen an der Ausrichtung oder Position des Geräts lassen sich schwer automatisiert testen. Die Bewegung festzustellen ist allerdings kein Problem der Entwicklung hybrider Applikationen für Android oder IOS, sondern stellt auch für native Applikationen eine Herausforderung dar. Da Appium die App „nativ“ ausführt, können für diesen Einsatzzweck Tools oder Hardware verwendet werden die sonst in nativen Android oder IOS Testumgebungen Anwendung finden.

4.5.3. Ripple

Ripple ist ein Emulator der entwickelt wurde, um Apache Cordova Anwendungen zu emulieren und das Testen dieser Applikationen zu ermöglichen. Ripple wird dabei komplett im Browser

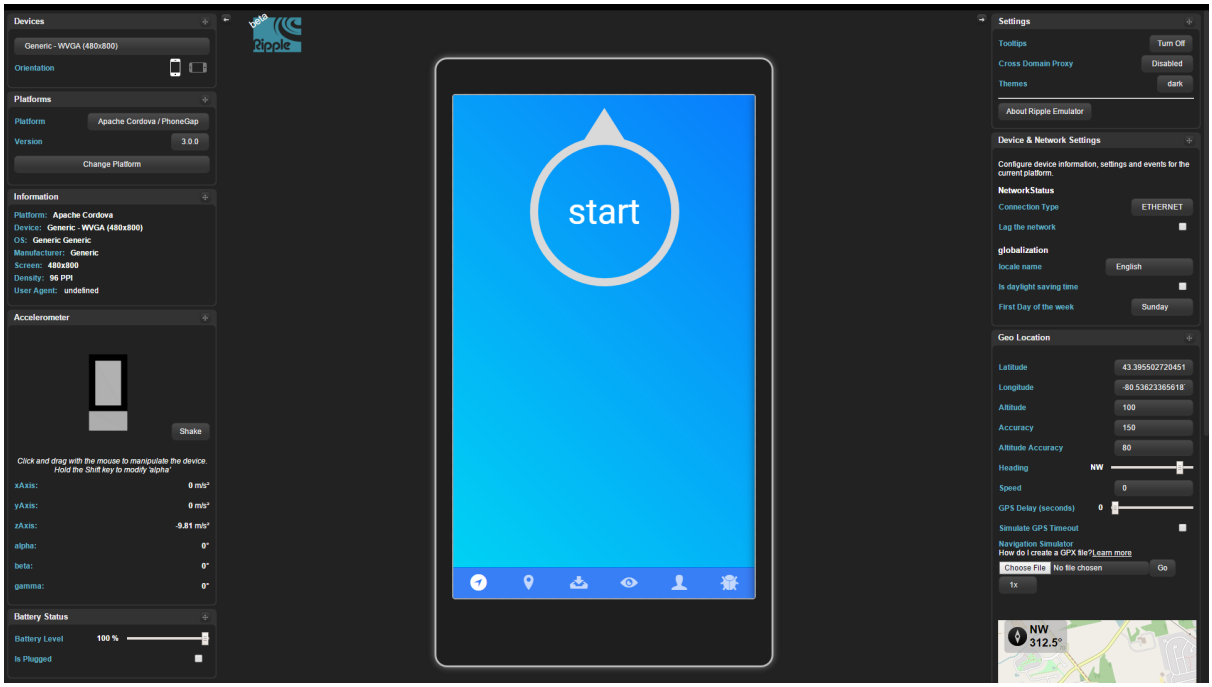


Abbildung 4.4.: Ripple Emulator Screenshot

ausgeführt und unterscheidet sich wenig von einer Webseite. Abbildung 4.4 zeigt eine Ausführung von Nifino im Ripple Emulator. Ein großer Vorteil ist hierbei die Unterstützung vieler Features, die vor allem auf Mobilgeräten zum Einsatz kommen und die vom Cordova oder ngCordova Framework bereitgestellt werden. Während bei Appium (Abschnitt 4.5.2) ein echtes Gerät oder ein nativer Emulator und das Kompilieren der Appdatei zum Testen notwendig ist, sind diese Schritte bei Ripple nicht nötig. Ripple erlaubt es frei virtuelle Positionsdaten, Bewegungs- oder Rotationswerte an die App weiterzugeben, sodass Funktionen wie Standort- oder Bewegungserkennung einfach und schnell getestet werden können. Die Entwickler beschreiben daher Ripple als „goto emulator for rapid development of mobile web applications“ [Rip]. Ein großes Problem ist hierbei allerdings die Steuerung. Es gibt kein Interface abseits der Webseite die für das virtuelle Bewegen oder Rotieren des Smartphones verwendet werden kann. Stattdessen muss der Entwickler oder Tester diese Bewegungen von Hand am Browser eingeben um solche Testfälle durchführen zu können. Dies hindert Appium am Einsatz für die kontinuierliche Integration, bei der Tests vollständig automatisiert ablaufen sollen. Der Einsatz ist insofern möglich, jedoch müssen in diesen Fällen konstante Positionen und Ausrichtungen betrachtet werden. Theoretisch ist es möglich, in den Testfällen auf die Oberfläche von Ripple zuzugreifen und die Modifikationen in den Testfällen vorzunehmen. Dies ist allerdings aufwendig und muss in zukünftigen Versionen von Ripple nicht mehr zwangsläufig unterstützt werden.
Wie bei Appium wird bei Ripple auf Selenium (Abschnitt 4.2.1) und dessen Webtreiber gesetzt, um mit dem Testrunner und Testframework kommunizieren zu können.

Name	Async. Test	Promise/Spy	Unit/E2E	(non)AngularJS	Erweiterung nötig
Jasmine	✓	✓/ ✓	✓/ ✓	(✓) / ✓	✗
Mocha	✓	✓/ ✗	✓/ ✓	(✓) / ✓	✓
QUnit	✓	✗/ ✗	✓/ ✓	(✓) / ✓	✓

Tabelle 4.2.: Überblick der genannten Testframeworks

4.6. Zusammenfassung

Um die Testprozedur weitestgehend zu automatisieren ist es sinnvoll Testrunner zu verwenden. Hierfür eignet sich vor allem Karma (Abschnitt 4.3.1) für die Unit Tests und Protractor für die End-to-End Tests (Abschnitt 4.3.3). Zum einen bringen diese viele der benötigten Features mit sich, zum anderen werden diese auch vom AngularJS Team für genau diese Zwecke empfohlen. Testem (Abschnitt 4.3.2) ist Karma oftmals ziemlich ähnlich bei der Unterstützung von Frameworks oder bei der Verwendung von Präprozessoren, lässt sich allerdings im Vergleich zu Karma schlechter erweitern. Letztlich bleibt dem Nutzer die Wahl welchen der beiden Testrunner, Karma oder Testem, er verwenden möchte. Unterschiede sind schwer auszumachen. Für Karma spricht, dass die AngularJS Entwickler dieses zur Unittest Testausführung empfehlen [Angd]. Zuletzt ist auch die große Anzahl an Erweiterungen ein Grund der für die Verwendung von Karma spricht.

Da die Unterstützung für Continuous Integration bei Protractor jedoch nicht so ausgeprägt ist, können Taskrunner wie bspw. Gulp oder Grunt eingesetzt werden um die Testausführung noch weiter zu automatisieren [Bai15]. Diese Automatisierung bietet den Vorteil, dass Unit Tests, End-to-End Tests oder automatisiertes Testen nicht mehr manuell angestoßen werden müssen. Diese beiden Tools erlauben eine Anbindung an ein Continuous Integration System und können die Testausführung der beiden Testrunner gebündelt starten.

Tabelle 4.2 zeigt nochmals die genannten Testframeworks auf und vergleicht diese. Beim Framework sticht vor allem Jasmine (Abschnitt 4.4.1) mit seinen Features hervor. Promises sind häufig in AngularJS Code vorzufinden und das Testframework bietet alle Voraussetzungen, um diese vernünftig testen zu können. Ebenfalls bieten Spies eine weitere nützliche Funktionalität um aufgerufene Funktionen zu überwachen. Auch Mocha (Abschnitt 4.4.2) bietet viele der geforderten Funktionalitäten. Allerdings fällt neben der fehlenden Unterstützung für Spies auch die starke Abhängigkeit von anderen Bibliotheken, bspw. bei der Wahl der Assertion Bibliothek, auf. Wir wollen den Einsatz verschiedener Tools möglichst gering halten, zum einen lassen sich Fehlerquellen so leichter eliminieren, zum anderen ist eine höhere Kompatibilität auch in Zukunft damit gewährleistet. Sofern eine Assertion Bibliothek Verwendung finden soll, lässt sich diese in die anderen Testframeworks integrieren. Ein weiteres Problem von Mocha ist die eingeschränkte Unterstützung von Protractor, welches als Testrunner von End-to-End Tests eingesetzt werden soll. Da ein einziges Testframework für alle Testphasen

Name	(virt.) Sensoren	Pushsystem	Sensormodifikation	Ausführung
Browser	(✗) / ✗	✗	✗	Webbasiert
PhantomJS	(✗) / ✗	✗	✗	Webbasiert
Appium	(✗) / ✓	✓	✗	native Anwendung
Ripple	(✓) / ✗	✗	-	Webbasiert

Tabelle 4.3.: Überblick der End-to-End Laufzeitumgebungen

verwendet werden soll, eignet sich Jasmine besser. Auch QUnit (Abschnitt 4.4.3) unterstützt die meisten Anforderungen, jedoch fehlt hier die einfache Möglichkeit Promises zu testen, was das Framework für AngularJS Applikationen etwas in den Hintergrund rücken lässt. Grundsätzlich eignen sich alle Javascript Testframeworks zum Testen von AngularJS Komponenten, jedoch müssen häufig, wie auch in QUnit, externe Abhängigkeiten manuell aufgelöst werden. Das manuelle Auflösen ist fehleranfälliger als das Automatisierte und zudem aufwendiger, wodurch die Verwendung von Jasmine vorteilhafter erscheint.

Bei der Wahl des Webtreibers, Selenium (Abschnitt 4.2.1), bleiben wenig Wahlmöglichkeiten. Jedoch bietet Selenium alles was von einem solchen Tool erwartet wird und eignet sich besonders für den Einsatz mit einer Testumgebung und Protractor als Grundlage für End-to-End Tests.

Die Wahl der passenden Laufzeitumgebung ist nicht einfach. Für Unit Tests kann PhantomJS (Abschnitt 4.5.1) verwendet werden, u.a. da er sich ideal in existierende Continuous Integration System integrieren lässt und er headless agiert. Vorteile bieten Umgebungen wie Chrome, Ripple oder Appium nicht, zumal diese länger zur Testausführung benötigen.
Das Auswählen der passenden Laufzeitumgebung für End-to-End Tests ist hingegen schwer, da sowohl Appium und Ripple einzigartige Vorteile bieten, die das jeweilige Pendant nicht bieten kann. Ein einfacher Browser oder PhantomJS eignen sich hierfür, sofern es sich bei der Anwendung um eine Webapp handelt, die v.a. im Browser aufgerufen und ausgeführt werden soll. Da Nifino jedoch mobile Endgeräten als Zielplattform hat, eignen sich diese Umgebungen weniger. Zur Wahl bleiben also noch Appium und Ripple.
Ripple hat den Vorteil, dass die Applikation nicht kompiliert werden muss und damit die Testumgebung und Testausführung schnell gestartet werden kann. Wie bei Appium werden Sensoren unterstützt, diese sind hier allerdings nicht physisch sondern virtuell. Damit ist man von der konkreten Position unabhängig und kann dem System, unabhängig des eigentlichen Standorts, jeden beliebigen Standort vortäuschen. Da Ripple selbst nur eine Webseite ist, ist es grundlegend möglich sich ein Skript zu schreiben, das Modifikationen an den Sensoren über die Weboberfläche durchführt. Dies ist allerdings aufwendig und kann durch Änderungen an Ripple in Zukunft zu Inkompatibilitäten des Testsystems führen.
Appium bietet hingegen den Vorteil eine native Applikation auszuführen, analog zum eigentlichen Endanwender. Zudem werden hier neben den physischen Sensoren auch Push-Notifications und Benachrichtigungen sowie plattformspezifische Plugins unterstützt. Um

diese zu Testen ist es unerlässlich Appium als Laufzeitumgebung für die Testausführung zu verwenden. Eine Möglichkeit, die Vorteile beider Systeme zu kombinieren, könnte darin bestehen beide Laufzeitumgebunden zu verwenden. Da sich beide in Protractor und damit in Continuous Integration Systeme integrieren lassen wäre dies ein akzeptabler Overhead, sofern ein Skript zur Modifkation der Weboberfläche von Ripple existiert. Andernfalls bietet Ripple zu wenig Vorteile um hier zweigleisig zu fahren.

5. Implementierung

Zu den Aufgaben dieser Arbeit zählen einerseits eine geeignete Datenstruktur für die Beacons zu erstellen. Auf das Beaconkonzept wurde bereits in der Arbeit „Concept and Implementation of Digital Beacons“ von Muhammad Bilal Chughtai[Chu15] eingegangen. Andererseits ist es Teil dieser Arbeit zu untersuchen, wie sich Nifino nach den Maßgaben der Softwaretechnik testen lässt und hierfür passende Tools zu finden. In diesem Kapitel werden diese beiden Aufgaben, aufgetretene Probleme und deren Lösungen beschrieben.

5.1. Beacons

Es gibt einige Anforderungen, die das Beaconsystem zu erfüllen hat. Neben den konkret implementieren Funktionalitäten soll es in Zukunft möglich sein, das Beaconsystem möglichst einfach zu erweitern um neue Funktionalitäten in späteren Iterationen hinzufügen zu können. Abbildung 5.1 zeigt den Use Case für die Nifino Beacons die im Rahmen dieser Arbeit implementiert wurden. Der Anwender soll die Möglichkeit haben, Beacons zu erstellen. Dazu muss der Anwender bei Nifino angemeldet sein und dem Beacon einen Namen geben, optional können weitere Eigeschaften des Beacons festgelegt werden.
Neben dem Erstellen von Beacons ist es möglich existierende Beacons zu aktualisieren und neuen Bedürfnissen anzupassen. Zudem kann auch das eigene Benutzerprofil aktualisiert werden um bspw. ein neues Profilbild zu verwenden oder seine Interessensliste zu bearbeiten. Erstellte Beacons sollen von anderen Benutzern gefunden und lokalisiert werden können. Das Auffinden dieser kann entweder über die Umgebung oder über eine Namenssuche oder Tagsuche durchgeführt werden.
Die genauen Use Case Beschreibungen sind in Anhang A.1 zu finden.

Zunächst wird allerdings die implementierte Datenstruktur beschrieben.

5.1.1. Datenstruktur

Beacon ist ein relativ abstrakter Begriff und kann im Rahmen von Nifino für viele Dinge stehen, die sich lokalisieren lassen oder zu denen eine Navigation gestartet werden soll. Hierfür kommen z.B. Sehenswürdigkeiten einer Stadt, Haltestellen öffentlicher Verkehrsmittel oder

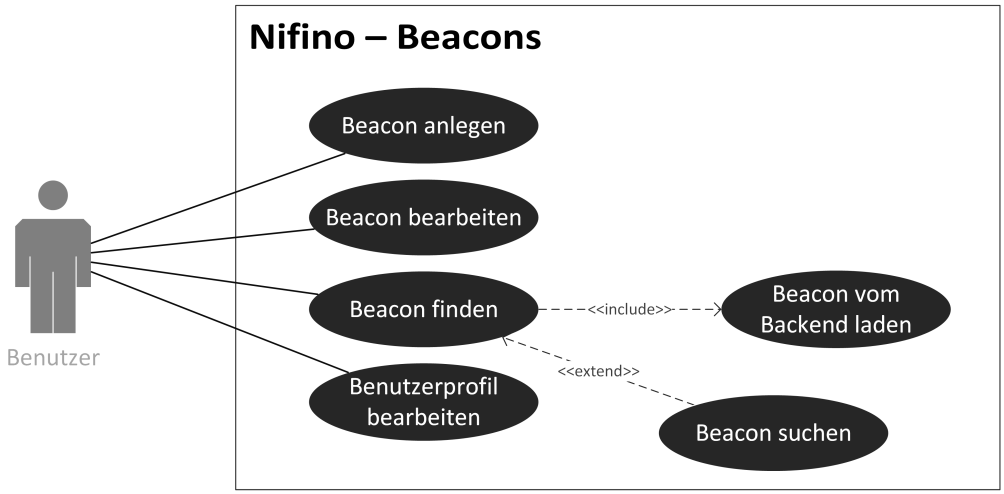


Abbildung 5.1.: Beacon Use Case

Einkaufszentren in Frage. Neben diesen statischen Beacons, die ihre Position beibehalten, steht der Beacon Begriff allerdings auch für den Benutzer selbst. Denn auch zu menschlichen, mobilen Beacons soll eine Navigation gestartet werden können, denn eine Fußgängernavigation zur gesuchten Person ist die Grundidee hinter Nifino.
Bei der Suche nach Beacons gibt es zwischen Benutzern und Objekten erstmal wenig Unterschiede. Daher war es auch ein Ziel die Beacons in einer gemeinsamen Tabelle im Backend zu verwalten und zu speichern und hierfür nicht unterschiedliche Tabellen anzulegen. Dennoch muss eine Unterscheidung später möglich sein, um etwaige Privatsphäre-Einstellungen für den Benutzer zu erlauben.
Ein weiteres Ziel war das lokale Beacon System so unabhängig vom verwendeten Backend zu halten wie nur möglich. Das noch für die Implementierung verwendete Backend Parse wird es ab dem Jahr 2017 nicht mehr geben, sodass eine Migration der Daten erfolgen muss. Um die Änderungen an der Applikation selbst so gering wie nur möglich zu halten wird die Verwaltung der Beacons mit dem Backend unabhängig der Datenstruktur implementiert. Neben einer Unabhängigkeit vom Backend sollte die verwendete Datenstruktur auch möglichst gut zu erweitern sein, denn es sind weitere, bisher noch nicht implementierte Features für die Beacons angedacht, die auch eine Erweiterung der verwendeten Datenstruktur vorsehen.
Abbildung 5.2 zeigt das Datenstruktur eines Beacons auf, das bereits so in Nifino implementiert ist. Viele Eigenschaften sind bisher allerdings nocht nicht implementiert. Die Funktionalität dieser wird erst mit weiteren Iteration der Applikation erscheinen.

Zu den bereits implementierten Funktionen gehört u.a. die `id` eines Beacons. Mit Hilfe dieser lässt sich das Beacon eindeutig identifizieren und Duplikate lassen sich vermeiden. Der Zweck des Namens und der Beschreibung sind offensichtlich. Im Falle eines menschlichen Beacons entspricht die Beschreibung etwa dem Status der Person, die nach aktueller Stimmung oder Verfassung den eigenen Wünschen angepasst werden kann.
Auch das Speichern und Verwalten der Position ist bereits implementiert, hierbei gibt die

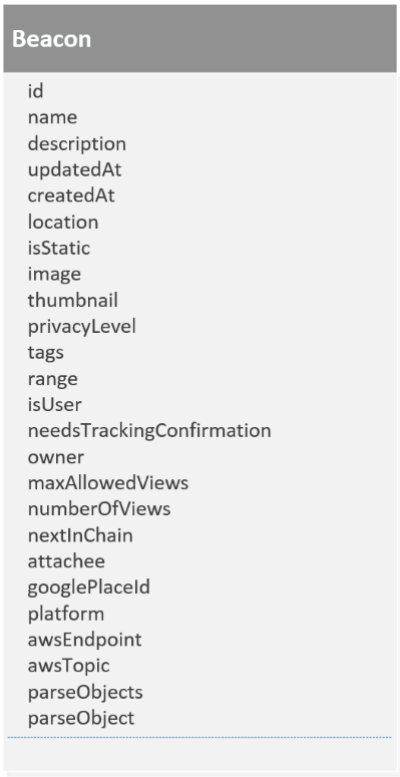


Abbildung 5.2.: Beacon Datenstruktur

Position den Standort des aktuellen Beacons an. Bei mobilen Beacons oder Menschen kann sich dieser jederzeit ändern, sofern sich die zu lokalisierende Person bspw. bewegt. Ein wichtiges Kriterium ob eine App bei den Endnutzern ankommt und häufig Verwendung findet liegt auch am Akkuverbrauch, den diese mit sich bringt. Um Energie zu sparen werden nur Positionsupdates angefragt und an das Backend übertragen, sofern dies auch benötigt wird, also genau dann wenn dieser Person gefolgt wird. Andernfalls sind diese Updates nicht von Nöten und auch die Genauigkeit der Lokalisierung wird heruntergestuft.
Desweiteren wird auch gespeichert ob ein Beacon statisch ist, sofern es sich um eine Person handelt ist dies nicht der Fall. Aber auch Objektbeacons können ihre Position verändern, bspw. Demonstrationen oder mobile Einrichtungen wie Wahlstände. Um neben dem Energieverbrauch auch den Datenverbrauch zu senken, werden die Bilder der Beacons in zwei Variationen gespeichert. Einmal in der Vorschau als Thumbnail und einmal in der vollen Auflösung für die Verwendung auf der Beaconansicht.

Zukünftig soll es möglich sein die Sichtbarkeit von Beacons einzuschränken und diese bspw. nur Freunden anzeigen zu lassen oder, dass man als Nutzer nicht mehr über die Suche gefunden werden kann. Hierzu existiert bereits das Element des `privacyLevel` in der Datenstruktur und im Backend. Die passende Funktionalität um diese Privatsphäre-Einstellungen vorzunehmen wurden jedoch noch nicht implementiert.

Neben diesen Features soll es in Zukunft auch möglich sein, den Radius, in dem das Beacon gefunden werden kann einzuschränken oder zu erweitern. Auch hierfür existiert bereits ein Attribut, zu dem die passende Funktionalität nur auf Applikationsebene nachimplementiert werden muss.
Ebenso soll es möglich sein, die Anzahl der Klicks auf ein Beacon zu beschränken um Aktionen wie „Die ersten 100 Besucher erhalten einen Rabatt“ leichter realisieren zu können. Analog dazu kann bereits die aktuelle Anzahl an Klicks auf eben dieses gespeichert werden. Ebenso kann Nifino in die Richtung virtueller Schnitzeljagden erweitert werden, sodass ein Beacon erst besucht werden muss, bevor ein hiermit verknüpftes Beacon getrackt werden kann.

Um der Umgebung einige Beacons hinzuzufügen ohne auf eine Community setzen zu müssen, können testweise Beacons aus der Datenbank von Google Places erstellt und getrackt werden. Dies ist v.a. zu Beginn nützlich, da ansonsten kaum statische Beacons in der Umgebung vorhanden sind, die getrackt werden können. Diese Funktionalität wurde im Rahmen dieser Arbeit implementiert. Hierzu werden aus der Umgebung des Anwenders relevante Google Places geladen. Die Darstellungs- und Suchlogik von Nifino soll unkompliziert gehalten werden, sodass nicht zwei Datenstrukturen, eine für die Beacons, eine für die Google Places eingesetzt werden. Stattdessen werden letztere in Beacons konvertiert und mit dem Backend synchronisiert. Dieses Laden und Konvertieren der Google Placees wird asynchron ausgeführt um die Anwendung in dieser Zeit nicht zu blockieren. Diese „Google Beacons“ werden als statische Beacons ohne Besitzer behandelt und sind von jedem in der Umgebung einsehbar. Um zu verhindern, dass Duplikate dieser Beacons entstehen können wird zusätzlich die eindeutige Google Places ID im Backend gespeichert. Anhand dieser kann später auch entschieden werden, ob dieses Beacon Nifino exklusiv ist oder ursprünglich von einer anderen Datenbank bezogen wurde.

Um Beacons nachträglich bearbeiten zu können wird zudem auch der Besitzer gespeichert. Um die Synchronisation mit dem Backend zu vereinfachen werden zudem die jeweils geladenen Objekte zum jeweiligen Beacon gespeichert.

Javascript bietet zwar Unterstützung für Objekte, nicht jedoch für Klassen. Hierzu werden in Javascript Funktionen verwendet, die Objekte mit Attributen ausstattet, die diese, ähnlich wie bei einem Konstruktor, mit Werten belegen und anschließend das erzeugte Objekt zurückgeben. Funktionen die sich auf diesem Objekt ausführen lassen, können wie die Attribute dem Objekt zugewiesen werden oder als Properties auf dem Objekt definiert werden. Die implementierte Beaconkonstruktor Funktion verwendet letztere Option.

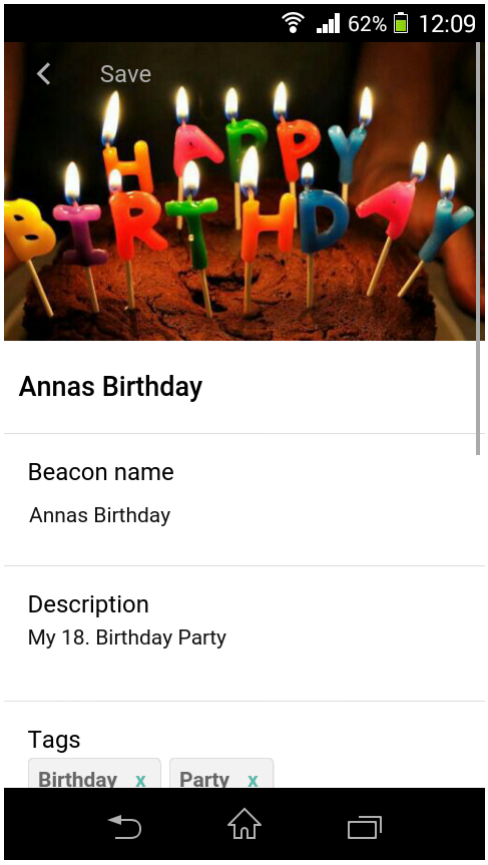


Abbildung 5.3.: Beacon erstellen

5.1.2. Erstellen und Bearbeiten von Beacons

Spätestens seit Twitter sind (Hash-)Tags aus der digitalen Welt kaum noch wegzudenken. Diese werden neben Twitter auch in diversen anderen Plattformen wie Facebook oder Instagramm verwendet. Sie dienen immer häufiger dazu Einträge nach eben diesen Tags zu gruppieren, wobei diese verschiedene Stimmungslagen oder Interessen ausdrücken. Daher soll auch ein Tagkonzept für die Beacons verwendet werden. Je nach Privatsphäre-Einstellungen soll es möglich sein Personen mit demselben Interesse zusammenzuführen und dadurch neue Bekannt- und Freundschaften entstehen zu lassen. Eventbeacons können über ein solches Tagkonzept mögliche Interessenten und damit ein genaues Zielpublikum ansprechen. Um diese Verknüpfung schaffen zu können ist es sinnvoll, dass sich Nutzer selbst über Beacons beschreiben und definieren können.
Zusätzlich sollte es auch möglich sein, neue Tags zu erstellen, die bisher noch nicht vorhanden sind. Hier kann es Sinn machen ein Ranking von Tags pro geographischem Gebiet zu machen, um meist verwendete Tags in der Umgebung kennenzulernen.

Listing 5.1 Map Direktive

```
directive('googleMapDirective', function () {
      var directive = {
      scope: {
            markerPos: '='
            },
            templateUrl: 'templates/editBeacon_GoogleMap.html',
            link: function (scope, iElement, iAttrs) {
                  var myLatlng = {lat: 48.7451328, lng: 9.1064215 };
                  var domIDMap = iElement.find('div')[0];
                  var map = new google.maps.Map(domIDMap, {
                        center: scope.markerPos,
                        zoom: 14
                  });

                  var marker = new google.maps.Marker({
                        position: scope.markerPos,
                        map: map,
                        clickable: true,
                        draggable: true,
                  });

                  marker.addListener('dragend', function(e) {
                        var pos = e.latLng;
                        scope.markerPos.lat = pos.lat();
                        scope.markerPos.lng = pos.lng();
                  });
            }
      };
      return directive;
})
```

Um das Tagsystem möglichst intuitiv zu gestalten wurden auch hierfür Direktiven implementiert, die sowohl für das Suchen von Beacons als auch für das Erstellen von Tags verwendet werden. Hierbei kann in ein Textfeld Freitext eingegeben werden, die Direktive bekommt eine Liste an Einträgen die durchsucht werden sollen und zeigt, sofern eine Übereinstimmung gefunden wurde, diese als Vorschlag an. Sofern es sich um einen noch nicht vorhanden Tag handelt, wird auch dieser als Vorschlag dargestellt und kann vom Anwender ausgewählt werden. Wie auch in Abbildung 5.3 zu sehen ist, werden gewählte Tags in einer Tagliste dargestellt. Diese einzelnen Tags können durch klicken einfach wieder entfernt, oder durch eine erneute Suche hinzugefügt werden. Auch für diesen Zweck wurde eine eigene Direktive implementiert, die die gewünschte Funktionalität für diverse Einsatzzwecke ermöglicht. Dieser Taglist muss lediglich ein Array an gewählten Elementen übergeben werden, welches dann wie in Abbildung 5.3 dargestellt wird. Das Array kann beliebig zur Laufzeit modifiziert werden.

Abbildung 5.3 zeigt, wie das Erstellen von Beacons in der Applikation aussieht. Viele Attribute der Beacon Datenstruktur (Abschnitt 5.1.1) können hierbei gesetzt werden, wie z.B. der

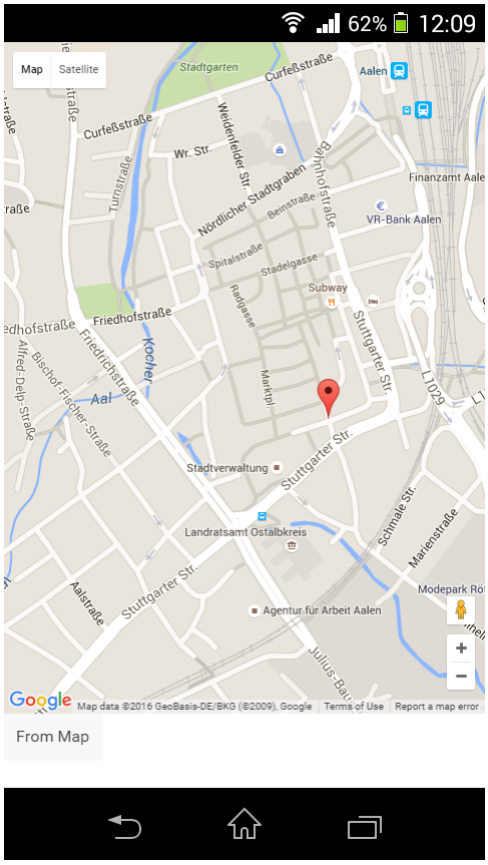


Abbildung 5.4.: Google Maps Direktive

Name, die Beschreibung oder auch die Tags. Plant man ein zeitlich begrenztes Event, wie in diesem Beispiel einen Geburtstag, kann zusätzlich ein Start- und Enddatum gesetzt werden. In Zukunft ist geplant, dass nur Beacons gefunden werden können, die in der Zukunft liegen. Vergangene Beacons sind uninteressant und könnten die Liste vorhandener Beacons unnötig unübersichtlich machen.

Grundsätzlich gilt, dass erstellte Beacons nie Personen sind. Personenbeacons werden nur beim Anlegen eines neuen Nutzerprofils erstellt. Die Bearbeitung eigener Beacons oder des Nutzerprofils ist ähnlich zu dem Erstellen neuer Objektbeacons. Zusätzlich kann auch noch der Ort bestimmt werden, an dem das Beacon anzutreffen ist. Hierzu kann Nifino entweder die Position dank GPS Sensoren automatisch bestimmen oder der Benutzer wählt einen Ort von der Karte aus. Hierzu wurde eine eigene Direktive implementiert. Listing 5.1 zeigt deren Implementierung. Im Template wird ein `div` Element erzeugt, das den Bildschirm füllt. In dieses Element wird in der Direktive die Google Map Ansicht geladen. Zudem wird ein Marker erstellt, der angeklickt und auf der Karte verschoben werden kann. Die Darstellung kann in Abbildung 5.4 betrachtet werden.

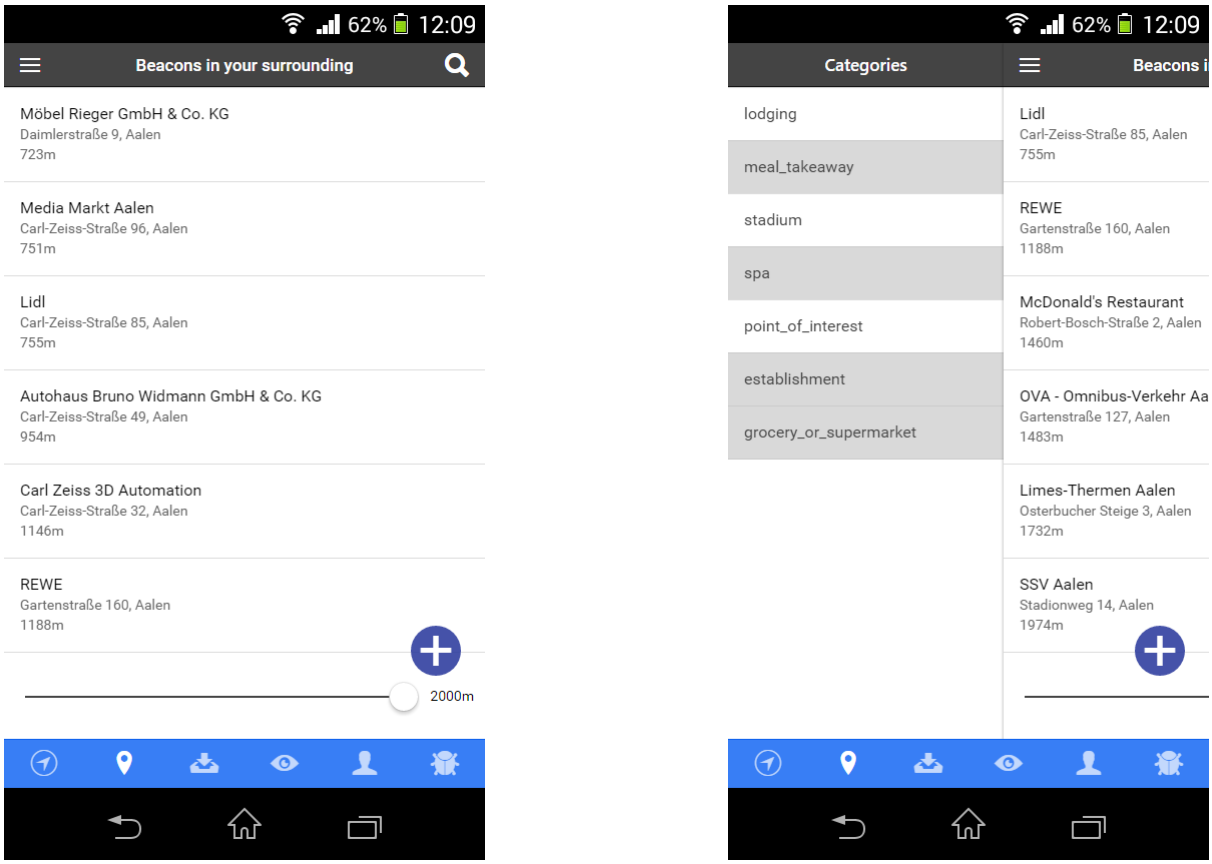
Das Profil- oder Beaconbild kann einfach gesetzt werden indem auf die Bildfläche geklickt wird. Der folgende Dialog erlaubt es dem Benutzer ein bereits gemachtes Bild aus der Bibliothek zu laden und dort darzustellen oder die Kamera zu aktivieren um ein neues Bild zu schießen.

Um die Funktionalität der Beaconerstellung auch für die Bearbeitung verwenden zu können, wurde auch der Beaconeditor hauptsächlich als Direktive implementiert. Dieser kann, bei Verwendung als HTML Attribut, eine Flag angehängt werden, mit Hilfe dieser dann unterscheidet wird, ob ein Beacon zu bearbeiten ist oder ob ein neues Beacon erstellt werden soll.

Erstellte Beacons können bearbeitet werden, hierzu wird dieselbe Direktive erneut aufgerufen, jedoch werden die Parameter so angepasst, dass das zu bearbeitende Beacon übergeben wird. Vorhandene Attribute wie bspw. der Name oder die Beschreibung des Beacons werden dann automatisch eingetragen.

5.1.3. Beaconsuche

Der Beaconfinder ist der interne Entwicklungsnamen für die Möglichkeit Beacons zu finden. Auch hierfür wurde eine Direktive entwickelt, die diese Aufgabe erledigt. Grundsätzlich werden die Namen, Beschreibungen und Tags aller existierender Beacons im Umkreis von zwei Kilometern vom Backend bezogen. Bei Bedarf können weitere Informationen wie bspw. Bilder vom Backend bezogen werden. Um die Suche möglichst angenehm zu gestalten kann durch wischen nach rechts eine Sidebar auf der linken Seite geöffnet werden. Abbildung 5.5b zeigt diese Sidebar. Dort werden alle vorhandenen Kategorien dargestellt. Diese können selektiert oder deselektiert werden um die jeweiligen Beacons in der Ansicht darzustellen oder zu verbergen. Ebenso befindet sich am unteren Bildschirmrand ein Slider, der die maximale Entfernung zu den Beacons beschränken kann.
Intern werden alle Beacons in einer Map, genauer einem Javascript Objekt gespeichert. Der Property Name entspricht hierbei der ID des Beacons, der Wert ist das Beaconobjekt selbst. Beacons sollen grundsätzlich mit aufsteigender Distanz dargestellt werden. Insofern sollte für die Darstellung eine Datenstruktur benutzt werden, die, im Gegensatz zu einem Set oder einer Map, eine Reihenfolge besitzt aber es dennoch erlaubt in möglichst kontanter Zeit abzufragen, ob ein Element bereits in dieser Datenstruktur vorhanden ist oder nicht. Das Hinzufügen oder Entfernen soll ebenfalls möglichst schnell funktionieren. Für diesen Einsatzzweck wurde ein Hybrid aus Map und Liste entwickelt. Eine Map speichert, ob das Element enthalten ist und die jeweilige Position des Elements in der sortierten Liste, sodass ein Zugriff auf dieses Element in konstanter Zeit möglich ist. Ebenso kann mit konstantem Zeitaufwand festgestellt werden, ob ein Element enthalten ist oder nicht. Etwas aufwendiger ist das Einfügen oder Entfernen von Elementen aus der Liste. Im schlimmsten Fall müssen beim Entfernen alle Indizes der Map angepasst werden, was eine lineare Laufzeit ergibt. Dargestellt werden die Beacons in derselben Reihenfolge in der sie in der Liste gespeichert sind.
Muss die Darstellung aktualisiert werden, wird über alle bezogenen Beacons iteriert und



(a) Beacons der Umgebung

(b) Auswahl der Kategorien

Abbildung 5.5.: Beacons finden

diese werden bei Bedarf in die Hybriddatenstruktur eingetragen bzw. von dieser entfernt. Um Beacons zu suchen wird dieselbe Direktive verwendet, die auch beim Erstellen von Beacons (Abschnitt 5.1.2) verwendet wird um nach Tags zu suchen. Abbildung 5.5 zeigt den Beaconfinder von Nifino. Hierbei kann ein seitliches Menu geöffnet werden in dem die Kategorien der verfügbaren Beacons dargestellt sind. Diese können nach Belieben selektiert oder deselektiert werden. Entsprechend wird die Darstellung der Beacons gefiltert. Viele bekannte mobile Applikationen verwenden Floating Buttons, in Nifino wird an der rechten unteren Ecke ein Button dargestellt, über den die Funktion zum Erstellen eigener Beacons aufgerufen werden kann.

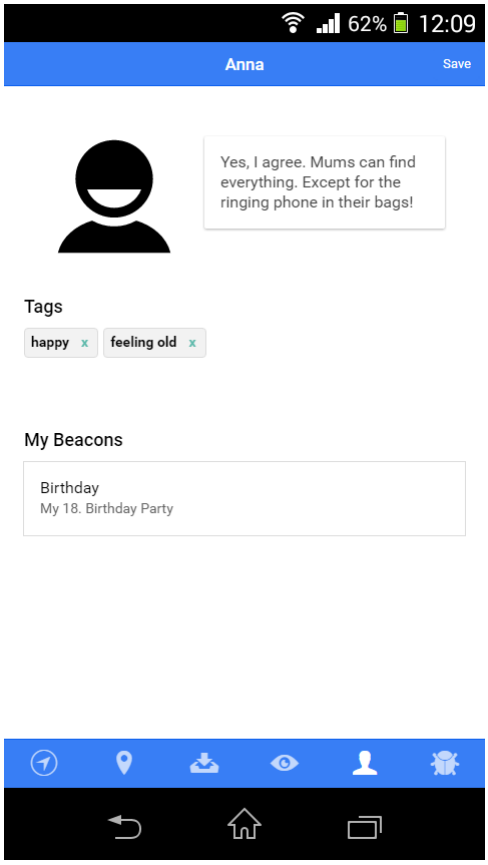


Abbildung 5.6.: Benutzer Profil

5.1.4. Benutzerprofil

Wie bereits erwähnt werden auch Benutzer in Nifino als Beacons realisiert, jedoch gibt es Unterschiede zwischen den menlischen Beacons und den Objektbeacons. Das Benutzerprofil wird daher auch nicht auf dieselbe Art und Weise dargestellt wie Beacons. Abbildung 5.6 zeigt ein Benutzerprofil. Der Anwender kann hierbei auch für sich selbst Tags festlegen um bspw. seine aktuelle Gemütslage zum Ausdruck bringen zu können. Ebenso kann dieser seine Beschreibung anpassen, was in diesem Fall eher als Statusnachricht angesehen werden kann. Diese wird, wie auch das benutzerdefinierte Profilbild auf der eigenen Seite dargestellt und kann modifiziert werden.
Neben diesen Informationen werden dem Nutzer auch seine eigenen Beacons angezeigt, auf die er über die eigene Profilseite zugreifen kann.
Die in Abschnitt 5.1.1 vorgestellte Datenstruktur unterstützt noch viele weitere Einstellungen für das Benutzerprofil, die noch implementiert und als Optionen auf dieser Seite dargestellt werden können.

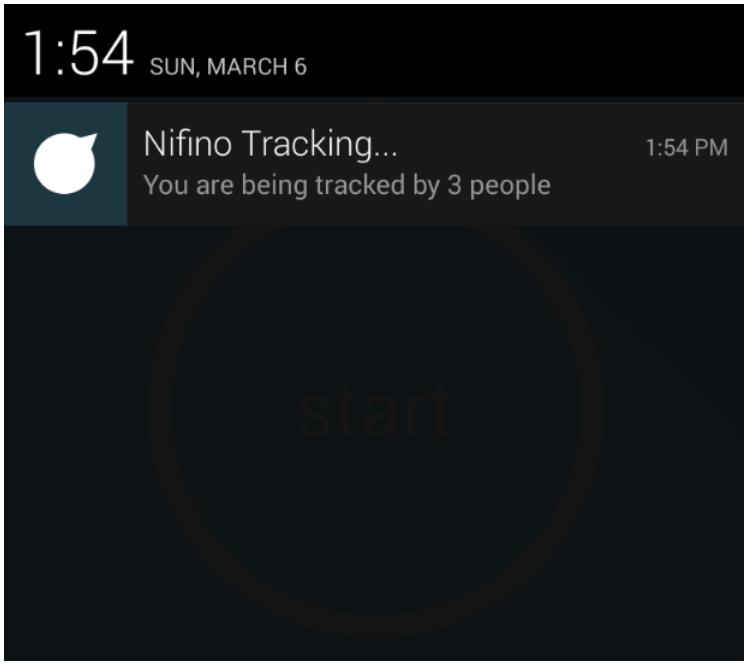


Abbildung 5.7.: Benachrichtigungs Plugin

5.1.5. Notifications

Um die Einbindung in das Android Ökosystem besser zu gestalten wurde ein Cordova Plugin geschrieben, um neben der in Javascript implementierten Funktionalität zusätzlich nativen Code ausführen zu können. Dieses Plugin erlaubt es, analog zu einem Mediaplayer, Benachrichtigung in der Benachrichtungsleiste von Android darzustellen. Der Titel und Text der dem Nutzer beim Betrachten angezeigt werden soll, kann jederzeit manipuliert und den Umständen angepasst werden. Die Benachrichtigung soll angezeigt werden, sofern eine Navigation aktiv ist. Dies ist entweder der Fall wenn die Person gerade von einer anderen Person getracked wird, oder wenn die Person selber zu einem Beacon navigiert werden möchte.
Abbildung 5.7 zeigt das Plugin in Aktion, wie beim einem Mediaplayer kann diese Benachrichtigung nicht deaktiviert werden. Dies geschieht erst dann, wenn keine aktive Navigation mehr ausgeführt wird.

5.2. Tests

Um eine möglichst problemfreie Applikation zu liefern wollen wir sicherstellen, dass Nifino bestmöglichst getestet wird und alle Features und Möglichkeiten, die die Applikation erlaubt, durch Testfälle abgedeckt wird. Da sich die hybride Nifino App in einem frühen Entwicklungsstadium befindet ist es zusätzlich wichtig zuverlässige Regressionstests (Abschnitt 3.2.1) zu implementieren. Dadurch wird gewährleistet, dass durch Erweiterungen der Funktionalität oder Refactorings die in diesen oder zukünftigen Entwicklungsstadien geschehen, die Funktionsweise nicht negativ beeinflusst wird. Im Kapitel 4 wurden bisher einige Tools erwähnt, die sich potentiell zum Testen von Nifino eignen. Im Folgenden werden zuerst die Unit-Tests genauer betrachtet.

5.2.1. Unit Tests

Wahl der Tools und Frameworks

In Kapitel 4 wurden Karma (Abschnitt 4.3.1) und Testem (Abschnitt 4.3.2) als Testrunner vorgestellt. Um Nifino Unit Tests zu unterziehen empfiehlt sich die Verwendung eines Testrunners, um die Ausführung zu automatisieren. Ansonsten müsste dies vom Entwickler selbst durchgeführt werden, was zum einen länger dauert und zum anderen deutlich fehleranfälliger ist. Testem und Karma sind sich sehr ähnlich, da sie sich in wenigen Punkten unterscheiden, dadurch fällt es schwer sich für eines der beiden Frameworks zu entscheiden. Auch aufgrund der Tatsache, dass Karma bereits länger existiert und Karma vom AngularJS Team selbst entwickelt wird, kommt es bei den Unit Tests von Nifino zu Verwendung.
Der Testrunner selbst kann zwar Testfälle ausführen, eignet sich jedoch nicht um Testfälle mit Hilfe von diesem Framework zu schreiben. Hierfür eignet sich Jasmine (Abschnitt 4.4.1), da alle unsere Anforderungen erfüllt werden. Daher wird dieses Framework Verwendung finden.

Einrichtung

Karma (Abschnitt 4.3.1) basiert auf Node.js und ist als NPM Paket für die Installation verfügbar, optional kann für die bessere Verwendung ein Command Line Interface mitinstalliert werden. Das Herz eines, durch Karma ausgeführten Testdurchlaufs ist die Konfigurationsdatei, diese kann von Hand angelegt werden, Karma bietet hierfür jedoch einen mitgelieferten Generator der eine passende Konfigurationsdatei erstellt.
Listing 5.2 zeigt eine Konfiguration des Karma Testrunners und Tabelle 5.1 erklärt die verwendeten Einstellungen und Optionen.
Für das Testframework Jasmine (Abschnitt 4.4.1) benötigt es keine spezielle Einrichtung. Das Framework muss nur installiert werden.

Listing 5.2 Konfiguration für Karma Testrunner

```
module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    files: [
      '../www/lib/ionic/js/ionic.bundle.js',
      '../www/lib/moment.min.js',
      '../www/app/**/*.js',
      '../bower_components/angular-mocks/angular-mocks.js',
      'unit-tests/**/*.js'
    ],
    exclude: [
    ],
    preprocessors: {
    },
    reporters: ['progress'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['PhantomJS'],
    singleRun: false
  })
}
```

`basePath`	Hauptverzeichnis, alle relativen Pfadangaben werden ab hier aufgelöst
`frameworks`	Liste an verwendeten Frameworks (bspw. Jasmine, QUnit, Mocha)
`files`	im Browser zu ladende Dateien
`exclude`	Dateien die im Browser nicht geladen werden sollen
`preprocessors`	Präprozessoren erlauben Dateien zu manipulieren bevor sie verwendet werden, hier Map von zu verwendeten Präprozessoren
`reporters`	zu verwendender Testreporter
`port`	zu verwendender Port
`colors`	Farbige Darstellung auf der Command Line
`logLevel`	eines von `config.LOG_DISABLE`, `config.LOG_ERROR`, `config.LOG_WARN`, `config.LOG_INFO`, `config.LOG_DEBUG`
`autoWatch`	Sollen Dateien überwacht werden und bei Änderung Test erneut ausgeführt werden
`browsers`	zu verwendender Browser
`singleRun`	Kontinuierliche Integration Ausführung

Tabelle 5.1.: Karma Konfigurationen

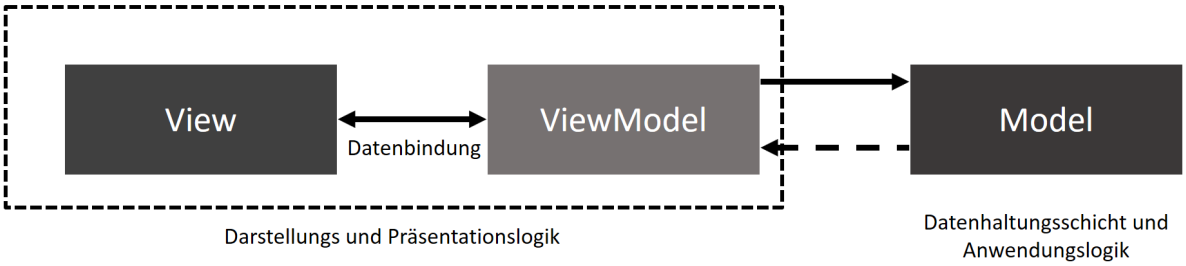


Abbildung 5.8.: Model-View-ViewModel

Testen von AngularJS Komponenten

AngularJS erlaubt es, Ansichten und HTML Seiten dynamisch anzupassen und zu modifizieren ohne dabei auf DOM Elemente, wie es bspw. bei jQuery der Fall ist, zugreifen zu müssen. Hierfür wird clientseitig HTML Code generiert, der aber Fragmente enthält die über den HTML Standard hinausgehen.
Um diese Funktionalität zu erreichen, verwendet eine AngularJS Applikation verschiedene Elemente, hauptsächlich **Controller**, **Services**, **Directiven** und **Filter**. Diese erfüllen alle verschiedene Aufgaben und tragen zur gesamten Funktionalität von AngularJS bei. Da diese vom Aufbau und der Funktionsweise sehr unterschielich sein können ist es nötig, auch in den Testfällen auf diese unterschiedlichen Elemente mit einer jeweilig eigenen Strategie vorzugehen. Diese einzelnen Elemente können in verschiedenen Modulen gruppiert werden, welche mit Paketen in Java verglichen werden können. Wie auch in Java existieren Abhängigkeiten voneinander, die für die korrekte Ausführung der Applikation aufgelöst werden müssen.

Controller Controller in AngularJS basieren auf dem Model-View-ViewModel Prinzip. Abbildung 5.8 zeigt hierbei das MVVM Muster auf, dieses unterscheidet sich hierbei kaum vom gewöhnlichen und bekannten Model-View-Controller Pattern. Den einzigen Unterschied zwischen den Patterns macht die Datenbindung aus. Während das MVC Pattern für gewöhnlich auf eine unidirektionale Beziehung setzt, wird im MVVM Pattern eine bidirektionale Beziehung zwischen dem View und dem Model verwendet.
In einem AngularJS Controller werden also die Logik zusammen mit dem ViewModel zusammengefasst. Der Controller ist in den meisten Fällen direkt mit einer View oder einem HTML Element verbunden, daher sollte dieser ohne UI Elemente nicht existieren. Für diese Fälle sollen Services eingesetzt werden.
Hierbei ist (nach MVVM Prinzip) die View oder das HTML Element bidirektional mit dem Model verbunden, das heißt Änderungen an der UI, bspw. durch den Nutzer wirken sich auf das Model, programmatische Änderungen wirken sich auf die Darstellung aus. Durch Controller wird der Großteil der darstellungsorientierten Funktionen einer Applikation ausgeführt. Die häufigsten Aufgaben von Controller sind u.a. die richtigen Daten für die Darstellung vorzubereiten (Daten von Server laden, Entscheidungen treffen welche Daten dem Nutzer präsentiert

werden sollen). Desweiteren sollte der Controller sich um die Präsentationslogik kümmern, d.h. welche UI Elemente sollen wie dargstellt werden, und wie diese aussehen sollen. Zudem ist es auch die Aufgabe des Controllers, auf Eingaben des Nutzers zu reagieren und Eingaben zu validieren [SG14].

Eine Eigenschaft von AngularJS ist es, die Geschäftslogik strikt von der Präsentationslogik zu trennen. Dies macht es tendenziell einfach Controller zu testen. Da Controller im globalen Scope nicht sichtbar sind, ist es notwendig den Controller zu Beginn zu injecten. Um Module und Abhängigkeiten zu mocken wird die `angular-mocks` Bibliothek verwendet. Mit dieser können benötigte Module geladen werden und stehen so für den Test zu Verfügung. Ein Testfall beginnt in der Regel damit, das Modul zu laden, dies kann ziemlich einfach realisiert werden, in dem in der `beforeEach` Funktion `module('modulename')` aufgerufen wird. Durch das Laden des Moduls werden auch benötigte Template Dateien durch einen *GET* Request angefordert. Der Request muss in den Testfällen explizit erwartet werden, ansonsten wird die Testausführung nicht erfolgreich durchlaufen.
Betrachten wir den Login Controller so fällt auf, dass dieser externe Abhängigkeiten zum UserManager besitzt, welcher die Login Daten mit Serverrücksprache validiert oder ablehnt. Da dies keine Funktionalität des Login Controllers ist, sollte dieser Teil auch keinen Einfluss auf das Testergebnis haben. Aufgrund dessen wird diese Unit später für sich alleine getestet. Also ist es nötig diese Funktionalität zu mocken. Listing 5.3 zeigt hierbei, wie das Hauptmodul der App geladen wird und wie benötigte Abhängigkeiten gemockt werden können. Im gemockten UserManager gilt es anschließend nur die verwendeten Funktionen zu implementieren. Da Unittests ohne Backend ausgeführt werden, können hartcodierte Logindaten festgelegt werden. Entsprechend wird die zurückgegebene Promise aufgelöst oder abgelehnt. Um auch Promises testen zu können ist es notwendig asynchrone Testfälle ausführen zu können. Um auf abgelehnte oder aufgelöste Promises reagieren zu können ist es nötig den `digest` Zyklus von AngularJS anzustoßen. Der `digest` Zyklus überprüft, ob gebundene Datenfelder manipuliert wurden und reagiert auf die asynchronen Ergebnisse der Promises. Aufgrund der vorhandenen Asynchronität muss auch ein Timeout verwendet werden um auf die Antwort zu warten, andernfalls würde der Testfall zu früh beendet werden.

Listing 5.4 zeigt einen beispielhaften Testfall, der das erfolgreiche Einloggen des Login Controllers testet. Abbildung 4.3 zeigte bereits den Grundaufbau eines Jasmine Test Szenarios. Hier begegnen wir der `it` Funktion, die in einer `describe` Umgebung implementiert ist. In diesem Context besitzen wir Zugriff auf den Scope und können darin Attribute des Scopes oder Parameter frei anpassen. Der Login Name und das Password wird, je nach Testfall, auf einen konstanten Wert gesetzt um anschließend die zu testende `login` Funktion aufzurufen. Die Benutzerdaten werden asynchron überprüft um die Anwendung nicht zu blockieren während der Loginprozess mit dem Backend kommuniziert. Die `login` Funktion ruft die in Listing 5.3 gemockte `loginWithCredentials` Funktion auf, die eine Promise zurückgibt und asynchron aufgelöst wird. Der Parameter (`done`) an die Testfunktion ist hierbei optional und wird dann verwendet wenn Asynchronität in Testfällen auftritt. Jasmine wartet standardmäßig fünf Sekunden bis die `done` Funktion aufgerufen wurde. Sofern der Parameter übergeben wird,

Listing 5.3 Factory Mocking

```
beforeEach(function() {
      module('nifino');

      module(function($provide) {
            $provide.factory('UserManager', [function () {
                  function loginWithCredentials(username, passwd) {
                        var p = $q.defer();
                        setTimeout(function() {
                              if(username == "a" && passwd == "b") {
                                    p.resolve();
                              } else {
                                    p.reject();
                              }
                              $rootScope.$digest();
                        }, 100);
                  return p.promise;
                  }

                  return {
                        loginWithCredentials: loginWithCredentials
                  };
            }]);
      });
})
```

sollte dieser nach der Ausführung der letzten `expect` Funktion erfolgen. Jasmine bietet, wie in Kapitel 4 beschrieben, die Möglichkeit Spies einzusetzen. Mit Hilfe dieser kann überprüft werden, ob Funktionen aufgerufen wurden, wie oft oder mit welchen Parametern diese aufgerufen wurden. Da der Status nach dem erfolgreichen Login wechselt, erwarten wir dies auch in unserem Testfall. Analog muss auch die eigentliche Authentifizierungsfunktion mit den gewählten Login Daten aufgerufen werden.
Um Funktionen beobachten zu können ist es allerdings notwendig Spione darauf anzusetzen. Da die Möglichkeit in mehreren Testfällen Verwendung findet, ist diese Funktionalität in die `beforeEach` Funktion ausgelagert worden. Durch Aufruf der `spyOn` Funktion wird ein Spion auf eine Funktion angesetzt. Hier existiert die Möglichkeit die Implementierung der überwachten Funktion durch Verwendung der `callFake(function)` Funktion anzupassen oder den Aufruf an die eigentlich implementierte Funktion weiterzureichen (`callThrough()`) und dabei nur die Aufrufdaten, wie Parameter für spätere Überprüfungen zu vermerken.

Die Durchführung wird letztlich dadurch ermöglicht, dass der Controller injectet wird, da er ansonsten nicht nach außen sichtbar wäre. Listing 5.5 zeigt hierbei wie dies geschieht. Benötigte Module können als Parameter angefügt werden und in der eigentlichen Funktion an globale Variablen des `describe` Blocks gebunden werden um diese später in einzelnen Testfällen verwenden zu können. Listing 5.3 zeigt das Mocken des UserManager und die Verwendung von diesem Mock Objekt, das in allen Testfällen Verwendung fndet. Auf diesem können benötigte

Listing 5.4 Asynchroner Testfall - erfolgreiches Einloggen

```
beforeEach(function() {
      ctrl = $controller('LoginCtrl', {$scope: $scope});
      spyOn($state, 'go').and.callFake(function(){});
      spyOn(mockedUserManager, 'loginWithCredentials').and.callThrough();

      expectGET($httpBackend);
});

it('should be able to login', function(done) {
      $scope.username = "a";
      $scope.password = "b";
      $scope.login();

      setTimeout(function() {
            expect(mockedUserManager.loginWithCredentials).toHaveBeenCalledWith("a", "b");
            expect($scope.password).toEqual('');
            expect($scope.username).toEqual('');
            expect($scope.loginButtonState).toEqual(true);
            expect($state.go).toHaveBeenCalledWith('mainnavigation');
            done();
      }, 300);
});
```

Listing 5.5 Controller Injection

```
beforeEach(inject(function(_$controller_, _$q_, _UserManager_, _$state_, _$rootScope_,
    _$httpBackend_){
      $controller = _$controller_;
      $q = _$q_;
      mockedUserManager = _UserManager_;
      $state = _$state_;
      $rootScope = _$rootScope_;
      $httpBackend = _$httpBackend_;
}));
```

Funktionen aufgerufen oder Spies angesetzt werden (Listing 5.4). Wir bereits erwähnt ist es hierbei noch nötig den *GET* Request für die zu ladenden Templates zu erwarten. Dies geschieht durch den AUfruf der `expectGET();` Funktion im `beforeEach` Block. Die Implementierung der Funktion ist relativ überschaubar. Für jedes zu ladende Template wird über das httpBackend Objekt ein Request erwartet.

Führt man dann diese Testsuite mit der gegeben Konfiguration aus, erzeugt dies die in Abbildung 5.9 dargestellte Ausgabe.

```
C:\Windows\System32\cmd.exe - karma start unit-tests.conf.js
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\roehrdor\Workspace\nifinoVS\nifinoVS\test>karma start unit-tests.conf.js
21 02 2016 23:06:57.083:WARN [karma]: No captured browser, open http://localhost:9876/
21 02 2016 23:06:57.099:INFO [karma]: Karma v0.13.15 server started at http://localhost:9876/
21 02 2016 23:06:57.108:INFO [launcher]: Starting browser PhantomJS
21 02 2016 23:06:57.968:INFO [PhantomJS 1.9.8 (Windows 8 0.0.0)]: Connected on socket vvyJSuuoOtDhZOUKA
AAA with id 89037718
PhantomJS 1.9.8 (Windows 8 0.0.0): Executed 5 of 5 SUCCESS (0.367 secs / 0.458 secs)
```

Abbildung 5.9.: Karma Ausgabe

Directive Ein Ziel von AngularJS ist es, möglichst unahängig der DOM Elemente arbeiten zu können und damit eine auch möglichst Browser unabhängige Applikation erstellen zu können. In manchen Situationen ist es allerdings nicht möglich auf die Manipulation der DOM Elemente zu verzichten, solche Operationen sollten nur in Direktiven ausgeführt werden [Angc]. Direktiven existieren auch, um über den HTML Standard reichende HTML Elemente und Attribute zu erstellen. AngularJS stellt bereits einige Direktiven bereit die am ng-Namensraum (bspw. ngController) zu erkennen sind.
Direktiven sind eine der mächtigsten Werkzeuge die einem als AngularJS Entwickler zur Verfügung stehen, dabei kann man diese in zwei Gruppen kategorisieren [SG14]. Zum einem gibt es die „Verhaltensmodifizierende Direktiven“, die auf bereits existierenden HTML und UI Elementen arbeiten. `ngShow` wäre ein Vertreter dieser Gruppe, diese Direktive versteckt oder zeigt ein Element basierend auf einer Bedingung die zur Laufzeit ausgewertet wird. Zum anderen gibt es die „Wiederverwendbaren Komponenten“, welche häufiger verwendet werden. In diesen Direktiven werden komplette HTML Strukturen neu erstellt und kombinieren hierbei die Geschäftslogik mit der Präsentationslogik. Sie kümmern sich darum, welche Daten woher geholt werden müssen, wie auf Nutzereingaben reagiert werden soll und welche davon wie dargestellt werden sollen. Direktiven sind daher auch die beste Möglichkeit Third-Party UI Komponenten in der eigenen AngularJS Applikation zu verwenden.

Direktiven müssen auf eine andere Art und Weise getestet werden als Controller. Da Direktiven zur eigentlichen Ausführung der Applikation kompiliert werden, muss dies auch in den Testfällen geschehen. Allerdings muss dieser Prozess explizit angestoßen werden. Um die benötigen Funktionen vewenden zu können, wird die aus dem Controller Test bekannte `inject` Funktion leicht angepasst. Neben dem `$rootScope` und dem `$httpBackend` benötigt man hier noch das `$compile` Element. Diese werden für spätere Verwendung in, für den `describe` Block globalen Variablen, gespeichert. Da auch in diesem Fall das Modul geladen

Listing 5.6 Directive Injection

```
var scope, initFlag;
initFlag = true;
function compile() {
      if(!initFlag)
            return compiledDirective;
      initFlag = false;
      scope = $rootScope;
      scope.makerPos = {};
      compiledDirective = $compile(angular.element('<div google-map-directive
          marker-pos="markerPos"></div>'))(scope);
      scope.$digest();
      return compiledDirective;
}

beforeEach(function() {
      module('nifino');

      inject(function(_$rootScope_, _$compile_, _$httpBackend_){
            $compile = _$compile_;
            $httpBackend = _$httpBackend_;
            $rootScope = _$rootScope_.$new();
      });

      expectGET($httpBackend);
      directiveElem = compile();
});

it('should not be empty', function() {
      expect(directiveElem.html()).not.toEqual('');
});
```

wird, ist es nötig den Request an die Template Dateien zu erwarten.
Die Funktion `compile()` übersetzt das Direktive Element, ruft den `digest` Zyklus des Scopes auf und gibt das kompilierte Element zurück. Dies kann, durch die Zuweisung zu einer globalen Variable in jedem Testfall verwendet werden.

Bei Änderungen von Werten, die in Direktiven verwendet werden, ist es nötig erneut den `digest` Zyklus von AngularJS anzustoßen, sodass die Werte auch in der Darstellung aktualisiert werden.
Direktiven können einen isolierten Scope bieten, der nur von dieser einsehbar ist. Durch Aufruf von `scope.isolatedScope()` kann auf diesen und seine Elemente zugegriffen werden.

Sofern Direktiven auf eine templateURL zurückgreifen, muss diese aufgelöst werden. Um das Problem zu lösen gibt es zwei Möglichkeiten. Entweder die Datei wird, sobald der Request

Controller	Service
Präsentationslogik	Geschäftslogik
Direkt mit der UI / View verbunden	komplett unabhängig von der Darstellung
Steuert die UI	Steuer die Applikation
Spezifisch (für View Komponente)	Wiederverwendbar (Singleton, in mehreren Views verwendbar)
Verantwortlich für: Welche Daten sollen geholt werden, welche angezeigt, wie soll mit Benutzerinteraktion umgegegangen werden und wie soll das UI gestaltet werden	**Verantwortlich für:** Serveraufrufe, Validationen, Speicherungen, wiederverwendbare Geschäftslogik

Tabelle 5.2.: Vergleich zwischen Controller und Services [SG14]

stattfindet, geladen und zurückgegeben. Alternativ lassen sich Karma Präprozessoren einsetzen, die die Dateien vor der Ausführung in einem Template Cache schreiben, sodass diese während der Testausführung geladen werden können.

Service Services sind mit Controllern vergleichbar, nur, dass diese keine View (UI) Komponente besitzen. Stattdessen enthalten Services reine Geschäftslogik und binden externe Ressourcen ein. Zusätzlich unterscheiden sich Services von Controllern in der Hinsicht, dass sie als Singleton instanziiert werden, wohingegen ein Controller pro View instanziiert werden kann. Services werden grundsätzlich lazy instanziiert, d.h. sofern diese zwar als Abhängigkeit angebunden, aber nicht verwendet werden, wird hierfür auch kein Objekt erstellt. Aufgrund der Singleton Eigenschaft der Services erhält jeder Teil unserer AngularJS Applikation daher auch Zugriff zur selben Instanz, wodurch geteilte Zustände, Caches oder Factories implementiert werden können. Tabelle 5.2 zeigt hier den Unterschied zwischen Controller und Services nochmals auf.

Wie auch bei Direktiven ist es bei Services angenehm Third-Party Komponenten einzubinden. Dem Entwickler steht es frei, eigene Services zu implementieren oder bereits existierende Komponenten zu verwenden. AngularJS bietet bereits von Haus aus zahlreiche Services an, die durch das vorangestellte `$` erkennbar sind. Für Ajax-Anfragen existieren bspw. `$http` und `$resource` Objekte. Beide greifen intern auf das `XMLHttpRequest-Objekt` zu.

Trotz einiger Unterschiede zwischen Controllern und Services lassen sich beide auf die gleiche Art und Weise einem Unit Test unterziehen, sodass es keine gravierenden Unterschiede zwischen einem Service Unit Test und dem in Listing 5.4 gezeigten Controller Unit Test gibt.

Filter Filter können auf diverse Ausdrücke in Darstellungstemplates angewandt werden, um die Ausgabe formatieren zu können. Neben Ausdrücken können Filter auch auf das Ergebnis anderer Filter angewandt werden. Ebenso ist es möglich, Filter mit Parametern zu versehen um die Ausgabe entsprechend anzupassen, bspw. den Parameter 2 für die Darstellung eines Geldbetrags mit zwei Nachkommastellen für den Cent Betrag in Euro.

Filter können als Funktionen betrachtet werden, die Daten in ein (besser) lesbares Format bringen. Dennoch sind diese wichtig, da sie Formatierungsaufgaben sowohl von der Präsentationslogik als auch von der Geschäftslogik abstrahieren. Beim Testen dieser Filter gibt es daher auch wenig zu beachten, Listing 5.7 zeigt hierbei die komplette PrettyFilter Testsuite. Diese Testfälle sind auch von anderen Frameworks und Programmiersprachen bekannt. Getestet wird hierbei nur die einzige `filter` Funktion, die einen einzigen Parameter (eine Distanz) übergeben bekommt und darauf basierend einen formatierten String als Ausgabe zurückgibt. Die Testfälle lassen sich einfach herleiten und implementieren. Für Distanzen über einem Kilometer soll auf eine Nachkommastelle (sofern vorhanden) abgerundet werden, für Distanzen unter einem Kilometer soll dies einfach als Distanz in Metern angegeben werden. Negative oder ungültige Werte sollen mit einer leeren Ausgabe quittiert werden.

5.2.2. End to End Tests

Wahl der Tools und Frameworks

Nifino soll auch End-to-End Tests (Abschnitt 3.2.5) unterzogen werden. Als Testrunner empfiehlt sich hier Protractor, da das Framework speziell auf diesen Einsatzzweck ausgelegt ist. Als Testframework haben wir die Wahl. Bspw. können wir hierfür Jasmine (Abschnitt 4.4.1), Mocha (Abschnitt 4.4.2) oder auch QUnit (Abschnitt 4.4.3) verwenden. Die von QUnit verwendete Syntax ist gewöhnungsbedürftig und weniger intuitiv als die bei Mocha oder Jasmine verwendete. Ebenso ist es weitaus aufwendiger QUnit passend zu konfigurieren und auch das Einbinden von Third Party Assertions Bibliotheken gestaltet sich hier schwieriger. Diese Kritikpunkte wiegen zu schwer, sodass auch die vorhanden Pluspunkte, wie die Vollständigkeit und Qualität der vorhandenen Features, sowie die gute Community Unterstützung oder auch die Unterstützgung vieler Continuous Integration Systeme die Wertung nur bedingt aufwerten können.
Im direkten Vergleich eignen sich daher Mocha und Jasmine deutlich besser für die von uns gestellten Anforderungen. Wie auch schon in Kapitel 4 beschrieben sind sich Mocha und Jasmine sehr ähnlich, sowohl was die Unterstützung verschiedener Features angeht, als auch was die verwendete Syntax betrifft. So fällt es auch relativ schwer sich für eines der beiden Frameworks zu entscheiden. Beide Tools lassen sich ziemlich einfach einsetzen, eignen sich für die testgetriebene Entwicklung, lassen sich gut konfigurieren und erweitern oder in CI Systeme integrieren. Da jedoch die Unterstützung Protractors für Mocha (vom Stand Dezember 2013) als beschränkt angegeben wird, fällt unsere Wahl auf das von Protractor preferierte Jasmine.

Listing 5.7 Pretty Filter Test Suite

```
'use strict';
describe('prettyDistance', function() {
      var $filter;

      beforeEach(function() {
            module('nifino');
      })

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

      describe('prettyDistanceTest', function() {
            var filter;
            beforeEach(function() {
                  filter = $filter('prettyDistance');
            });

            it('returns empty when nan', function() {
                  var o = filter(NaN);
                  expect(o).toEqual('');
            });

            it('displays in km when larger than 1 km', function() {
                  var o = filter(1234);
                  expect(o).toEqual("1.2 km");
            });

            it('displays no past comma digit if not needed', function() {
                  var o = filter(2000);
                  expect(o).toEqual("2 km");
            });

            it('displays in m for less than 1 km', function() {
                  var o = filter(575.3124);
                  expect(o).toEqual("575.3124 m");
            });

            it('displays no negative values', function() {
                  var o = filter(-1);
                  expect(o).toEqual('');
            });
      });
});
```

Testumgebung

Nifino verwendet als Browser auf Android Geräten Crosswalk um eine einheitliche Darstellung auf verschiedenen Geräten zu ermöglichen. Crosswalk ersetzt hierbei die Standard Webview Komponente des Android Ökosystems. Je nach verwendetem Handy bzw. ROM und verwendeter Android Version kann sich auch die Webview Komponente unterscheiden, sodass die Darstellung der App auf unterschiedlichen Geräten auch unterschiedlich ist. Dies ist jedoch ein nicht tragbarer Zustand. Durch die Verwendung von Hybrid Anwendungen sollte zudem sichergestellt werden, dass Applikation sowohl auf Android als auch Apple Geräten gleich aussieht und jeweils so nativ wie möglich anfühlt. In Kombination mit Appium und Crosswalk existiert jedoch das Problem, dass durch Crosswalk nicht mehr unmittelbar auf die Webview Komponenten zugegriffen werden kann und damit auch die DOM Elemente nicht betrachtet oder manipuliert werden können, was für einen End-to-End Test jedoch von Nöten ist. Insofern eignet sich auch Appium nicht als alleiniges Framework das zum Testen vom End-to-End Tests verwendet werden kann.
Um die aufgezeigten Probleme zu lösen gibt es letztlich mehrere Möglichkeiten. Zum einen ist es möglich auf Crosswalk zu verzichten, sodass Appium als einzige Testumgebung eingesetzt wird. Dies hat den Vorteil, dass jeder Testfall in der selben Testumgebung ausgeführt werden kann. Auf der anderen Seite wird dann aber Crosswalk nicht mehr verwendet, was oben beschriebene Probleme mit sich bringen kann.
Eine mögliche Lösung hierbei wäre es den Testprozess aufzuteilen. Zum einen wird die Applikation ohne Crosswalk kompiliert, sodass die Mobilgerät spezifischen Testfälle mittels Appium ausgeführt werden. Also solche in denen Benachrichtigungen oder aber auch Pushnachrichten zum Einsatz kommen. Alle anderen Testfälle hingegen können dank Ripple abgedeckt werden. Sodass diese dann in der Ripple Umgebung ausgeführt werden. Da Ripple im Chrome Browser ausgeführt werden kann und Crosswalk selbst auf der Chromium Basis aufbaut sind hier keine unterschiedlichen Ausführungen des selben Codes zu erwarten, beide setzen auf die V8 Javascript Engine. Seit Android 4.4 (KitKat) basiert die standardmäßig eingesetzte WebView Komponente ebenfalls auf dem Chromium Projekt, was den Einsatz von Crosswalk überflüssig machen kann. Es gibt weitere Gründe Crosswalk zu verwenden, jedoch benötigt Nifino diese Features nicht. So ist selbst ohne Verwendung von Crosswalk das Problem der unterschiedlichen Darstellungen in der Standard WebView Komponente gelöst.
Bei der Verwendung von Crosswalk wurden jedoch einige Probleme festgestellt. Manche Callbacks des HTML5 Standards[1] werden nicht mehr zuverlässig aufgerufen, sobald die Anwendung in den Hintergrund gerät. In diesem Kapitel wird Crosswalk nicht mehr für die Darstellung verwendet, stattdessen kommt die Android interne WebView zum Einsatz, was die Verwendung von Ripple als sekundäre Laufzeitumgebung obsolet macht.

[1] Bspw. gibt es Probleme bei der Implementierung der `navigator.geolocation.watchPosition()` - https://crosswalk-project.org/jira/browse/XWALK-4623

Listing 5.8 Protractor Konfiguration

```
exports.config = {
   seleniumAddress: 'http://localhost:4723/wd/hub',

   specs: ['./e2e-tests-appium/*tests.js'],

   capabilities: {
      platformName: 'android',
      deviceName: 'emulator-5554',
      browserName: "",
      autoWebview: true,

      // Absoluter Pfad zur Applikationsdatei
      app: 'C:/users/roehrdor/nifino/app/nifino.apk'
   },
   baseUrl: 'http://10.0.2.2:8000',

   onPrepare: function () {
      var wd = require('wd'),
            protractor = require('protractor'),
            wdBridge = require('wd-bridge')(protractor, wd);
      wdBridge.initFromProtractor(exports.config);
   }
};
```

Einrichtung

Protractor (Abschnitt 4.3.3) ist, wie auch Karma, ein auf Node.js basierendes Programm. Das Tool kann als NPM Paket heruntergeladen und installiert werden. Protractor verwendet standardmäßig Jasmine als Testframework. Bei der Installation wird neben Protractor selbst auch das Tool „webdriver-manager“ installiert. Dieses Hilfstool vereinfacht die Kommunikation zu Selenium. Mit Hilfe dieses kann der Selenium Server ohne großen Aufwand initialisiert und gestartet werden.

Wie bei Karma setzt Protractor eine Konfigurationsdatei ein, die bei der Testausführung geladen wird. Listing 5.8 zeigt die verwendete Konfiguration. Die `seleniumAddress` gibt die Adresse des laufenden Selenium Servers an, mit dem sich Protractor für die Testausführung verbindet. Im `spec` Array werden die Testdateien oder „specs“, wie sie von Protractor benannt werden, aufgelistet. Die `capabilities` beschreiben die Verbindung zur Testausführung, die Applikation wird auf einem Android Gerät mit dem Namen „emulator-5554“ ausgeführt. Der `browserName` ist hierbei leer, da die Standard WebView verwendet wird. Zuletzt wird noch der absolute Pfad (`app`) zur Android APK angegeben.
Die `baseUrl` gibt den Pfad zur laufenden Applikation an. Zusätzlich wird der `onPreprare` Callback implementiert.

`seleniumServerJar`	relativer Pfad zur Selenium Server jar Datei
`localSeleniumStandaloneOpts`	Objekt mit Port und Argument Array, das an Selenium Server übergeben wird
`chromeDriver`	Pfad zum ChromeDriver, bei null wird PATH verwendet
`seleniumAddress`	Adresse eines laufenden Selenium Servers
`seleniumSessionId`	Mit der Session ID kann eine laufende Selenium Instanz verwendet werden
`webDriverProxy`	Proxyserver für die Verbindung zum Selenium Server
`directConnect`	Falls `true` wird eine direkte Verbindung zum Browser hergestellt. Den Pfad geben `firefoxPath` oder `chromeDriver` an. Andere Browser werden hierbei nicht unterstützt
`firefoxPath`	Pfad zu Firefox, falls null wird Standardverzeichnis durchsucht
`noGlobals`	Verwendung von globalen Variablen (wie bspw. `protractor`, `element`, `by`) erlauben. Bei `false` wird nur die globale Variable `protractor` verwendet
`specs`	Array von relativen Pfaden zu Testdateien
`exclude`	Array von auszuschließenden Textdateien
`suites`	Objekt von Testsuites, diese können mit dem Parameter *–suite=<suiteName>* ausgeführt werden
`suite`	standardmäßig zu verwendende Suite
`capabilities`	Objekt zur Kommunikation mit dem WebDriver. Sofern der Test in mehreren Browsern ausgeführt werden soll, sollte `multiCapabilities` Array verwendet werden
`baseUrl`	URL zu der zu testenden Anwendung
`rootElement`	CSS Auswahl für das applikationsumgebende Element, standardmäßig *body*
`allScriptsTimeout`	maximale Zeit für jedes Testscript
`getPageTimeout`	maximale Zeit, für das Laden
`onPrepare()`	Callback der ausgeführt wird nachdem die Umgebung verfügbar ist und bevor die Testfälle ausgeführt werden
`params`	Objekt das an Protractor übergeben wird, das in den Testfällen verwendet werden kann
`resultJsonOutputFile`	Falls gegeben wird an diesem Pfad ein JSON Report erstellt
`framework`	Testframework, *jasmine*, *mocha* oder *custom*
`jasmineNodeOpts`	Optionen für Jasmine Framework
`plugins`	Array für zu verwendende Plugins

Tabelle 5.3.: Protractor Konfigurationen [Anga]

In der verwendeten Konfigurationsdatei wird nur ein Teil der möglichen Konfigurationseinstellungen verwendet. Tabelle 5.3 gibt eine Übersicht über die wichtigsten Einstellungsmöglichkeiten, eine vollständige Übersicht kann auf Github betrachtet werden [Anga]. Weitere Möglichkeiten betreffen v.a. die Verwendung anderer Testframeworks, die für unseren Fall nicht von Nöten sind; oder die Verwendung von Remote Server von Sauce Lab oder Stackuser, auf denen die Testausführung stattfinden kann, auch diese Optionen werden im Rahme dieser Arbeit keine Verwendung finden.

End-to-End Test

Um eine Anwendung einem End-to-End Test unterziehen zu können ist es wichtig auf die Elemente dieser Seite zugreifen zu können. Protractor setzt hierbei „Locators“, zu Deutsch Positionszeiger ein. Diese Positionszeiger helfen Protractor beim Aufspüren eines DOM Elements. Hierzu stellt Protractor das globale `by` Objekt bereit, auf dem Elemente anhand ihrer `id`, des `binding` oder des `model` selektiert werden können. Sofern das zu suchende Element keines der drei Attribute besitzt, kann auch mittels `css` auf dieses zugegriffen werden.
Ein wichtiger Bestandteil hiervon ist die globale Funktion `element()`. Diese nimmt einen solchen Positionszeiger als Argument entgegen und liefert ein `ElementFinder` zurück. Auf Letzterem können verschiedene, u.a. manipulierende Operationen ausgeführt werden. Auf dieses Element kann geklickt werden, der Mauszeiger, sofern vorhanden, kann zu diesen bewegt werden oder eine Texteingabe kann vorgenommen werden. Das Auffinden der einzelnen Elemente findet hierbei asynchron statt [Prod]. Protractor erlaubt es durch Aneinandereihung der `element` Funktion auch Kind Elemente ausfindig zu machen und für diese ein `ElementFinder` zurückzugegeben. Die vollständige von Protractor bereitgestellte API kann hier [Angb] eingesehen werden.

Abbildung 5.10 zeigt die Login Seite von Nifino, die möglichen Interaktionen, die der Nutzer an dieser Seite durchführen kann sind rot umrandet.

- **Username** Textfeld zur Eingabe des Benutzernamens
- **Password** Textfeld zur Eingabe des Passworts
- **Login** Button um sich einzuloggen, dieser wird erst aktiviert sobald ein Name und Passwort eingegeben sind
- **Sign Up** Button um sich als neuen Benutzer bei Nifino anmelden zu können

Auf diese Elemente muss in einem entsprechenden End-to-End Testfall eingegangen werden. Listing 5.9 zeigt einen Ausschnitt aus dem implementierten Testfall.
Auf die Texteingabe Felder muss hierbei nicht explizit eingegangen werden, da diesen alleine keine Funktionalität zugeordnet ist. Im ersten Testfall wird überprüft, ob diese vier Elemente alle gefunden wurden und auf der Seite vorhanden sind. Werden die Assertions in diesem Testfall nicht erfüllt, schlagen auch die anderen Tests fehl. Im nächsten Testfall wird

Listing 5.9 Ausschnitt aus dem Login Testfall

```
describe('login', function(){
      var username, password, loginButton, signupButton;

      beforeEach(function() {
            // (1)
            browser.sleep(2000);
            username = element(by.model('username'));
            password = element(by.model('password'));
            loginButton = element( by.css('[on-touch="login()"]'));
            signupButton = element( by.css('[on-touch="signUp()"]'));
      });

      it('expect elements to exist', function() {
            expect(username.isPresent()).toBeTruthy();
            expect(password.isPresent()).toBeTruthy();
            expect(loginButton.isPresent()).toBeTruthy();
            expect(signupButton.isPresent()).toBeTruthy();
      });

      it('expect signup button to work', function() {
            // (2)
            browser.driver.touchActions().tap(signupButton).perform();

            expect(browser.getLocationAbsUrl()).toMatch('/signup');
      });

      it('expect the login button to validate on input modification', function() {
            expect(loginButton.getAttribute('disabled')).toEqual('true');
            username.sendKeys("user");
            password.sendKeys("pw");
            expect(loginButton.getAttribute('disabled')).toEqual(null);
            username.clear().then(function() {
                  password.clear().then(function() {
                  });
            });
      });

      //
      // ....
      //

});
```

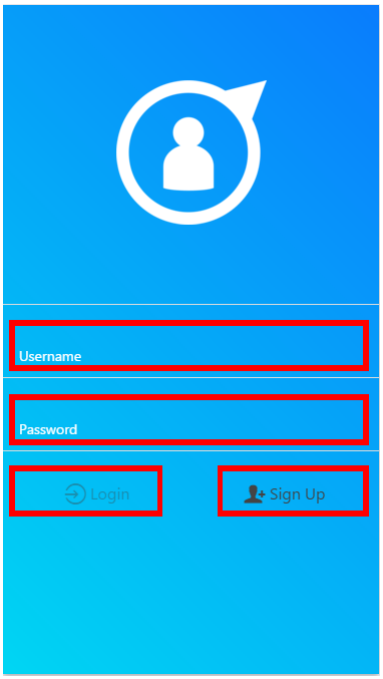


Abbildung 5.10.: Nifino Login Dialog

Listing 5.10 Protractor als Gulp Task

```
gulp.task('e2e', function() {
   exec('protractor test/e2e-tests-appium.js', function(err, stdout, stderr) {
      console.log(stdout);
      console.error(stderr);
   });
});
```

die Funktionalität des `Sign Up` Buttons getestet. Hierzu muss eine `touchAction` über den Browsertreiber gestartet werden um auf den Button „tappen“ zu können. Nach dem Ausführen dieser Aktion erwarten wir, dass wir uns nun auf der Signup Seite befinden.
Der dritte und letzte dargestellte Testfall überprüft, ob das Deaktivieren des Login Buttons bei fehlendem Benutzername oder Passwort entsprechend funktioniert. Ohne Eingabe wird erwartet, dass dieser deaktiviert ist. Nachdem die Daten eingegeben wurden, wird erwartet, dass der Login nun verwendet werden kann. Anschnließend wird die Eingabe wieder entfernt.

Testausführung

Diese Tests werden mittels Appium, wie in Listing 5.8 angegeben, auf einem nativen Android Emulator ausgeführt. Um auch Pushnachrichten testen zu können ist es wichtig, eine Version mit Google API auszuwählen, andernfalls kann nicht auf GCM zugegriffen werden.

Dieser Testfall lässt sich nahezu ohne Änderung auch in einem Webbrowser ausführen. Hierzu müssen die mit *(1)* und *(2)* markierten Stellen in Listing 5.9 angepasst werden. In der `beforeEach()` muss hierzu ein `browser.get('/#/login');` hinzugefügt werden um jeweils von der entsprechenden Seite zu starten. Bei *(2)* kann neben der `touchAction` auch ein einfacher `click()` verwendet werden, der auf dem `signupButton` ausgeführt werden kann. Hierbei wird eine Promise zurückgegeben. Für die Ausführung im Browser ist jedoch eine Modifikation der Konfiguration von Nöten.
Um den Test ausführen zu können muss die Applikation bzw. Appium bereits ausgeführt werden, im Falle des Browsers kann dies mit `ionic serve -nobrowser`, im Falle von Appium einfach mit `appium` geschehen. Die Testausführung wird durch den Aufruf von Protractor mit dem Pfad zur Konfigurationsdatei gestartet.
Gulp kann verwendet werden um Protractor mit einem Taskrunner ausführen zu können. Listing 5.10 zeigt diese Verwendung. Sofern gewünscht, kann in diesem Task auch das erneute Kompilieren der Applikation und Installieren auf einem physischen Gerät oder Emulator vorgenommen werden.

6. Zusammenfassung und Ausblick

Es gibt eine Vielzahl an Möglichkeiten mobile Anwendungen auf Basis von AngularJS zu testen. Einige dieser Tools und Frameworks wurden in dieser Arbeit detaillierter beschrieben und miteinander verglichen. Karma, Protractor und Jasmine wurden für die Implementierung der Testfälle ausgewählt. Diese erfüllten alle Anforderungen die das Testen von Nifino an solche Tools stellte.

An die Laufzeitumgebung für Unit Tests wurden keine hohen Anforderungen gestellt. Hierbei empfiehlt sich v.a. der Einsatz des headless Browsers PhantomJS, da dieser schnell ist und sich für die Verwendung in einer Continuous Integration Umgebung eignet. Für End-to-End Tests wird Appium verwendet, da dies die meisten Anforderungen erfüllt. Lediglich das Simulieren von Sensoren ist hier aufwendig.

Im weiteren Verlauf wurde die Datenstruktur für Beacons erstellt, diese basiert auf einem Javascript Objekt und unterscheidet nicht zwischen Objekt- und Personenbeacons.
Für das Erstellen und Bearbeiten von Beacons wurde, im AngularJS Stil, ein eigener Controller und wiederverwendbare Direktiven geschrieben, sodass die grundlegende Codebasis zwischen dem Erstellen und Bearbeiten nicht unterscheidet.
Direktiven wurden ebenfalls für das Finden von Beacons geschrieben, ebenso wie für den angenehmen Umgang mit dem Tagsystem, welches passende Möglichkeiten vorschlägt und diese übersichtlich darstellt.
Um Nifino auf Android Systemen nativ wirken zu lassen wurde zudem ein Plugin entwickelt, das es erlaubt Benachrichtigungen über das Benachrichtigungsystem darstellen zu lassen.

Ausblick

Nifino befindet sich, wie auch einige in dieser Arbeit vorgestellte Tools, in einer frühen Phase und ist noch nicht am Endpunkt der Entwicklung angelangt. Mit den Beacons wurde ein interessantes Konzept erschaffen, um Entitäten in seiner näheren Umgebung zu entdecken und sich zu diesen führen zu lassen. Die erstellte Datenstruktur eignet sich auch um das Beaconkonzept zu erweitern und es somit für weitere Einsatzzwecke zu verwenden. Eine digitale Schnitzeljagd in Form einer Aneinanderreihung von Beacons, oder Geofences zum Einschränken der Sichtbarkeit, sind hier denkbar. Auch die beschriebenen Testframeworks und

-tools werden sich weiter entwickeln und in Zukunft vielleicht die Möglichkeit bieten, beliebige Sensoren zu virtualisieren und über ein unkompliziertes Interface für die Testausführung dynamisch anzupassen. Auch AngularJS 2, welches auch für Mobilgeräte optimiert ist, befindet sich in einem Beta Stadium und wird in Zukunft einen großen Einfluss auf die Erstellung hybrider Mobilanwendungen haben.
Grundlegende Arbeit wurde auch für ein weiteres Plugin getan, das analog zu den Facebook Chatheads, die Richtung des zu trackenden Beacons, unabhängig der geöffneten App, auf dem Bildschirm anzeigt. Dieses sollte insofern ausgebaut werden, dass mehrere Beacons parallel unterstützt werden.

A. Anhang

A.1. Use Case Beschreibungen

Name:	**Beacon anlegen**
Beschreibung:	Ein Beacon kann von einem Nifino Anwender angelegt werden. Dieses Beacon wird mit dem Backend synchronisiert, kann gefunden und vom Benutzer später bearbeitet werden
Vorbedingung:	Benutzer ist angemeldet
Nachbedingung:	Neues Beacon angelegt und im Backend gespeichert
Fehlersituation:	• Benötigte Felder nicht ausgefüllt • Internetverbindung nicht vorhanden
Systemzustand im Fehlerfall:	Beacon wurde nicht gespeichert
Akteure:	• Benutzer • Nifino Applikation • Backend
Standardablauf:	

1. Benutzer öffnet Seite zum Beacon erstellen
2. Name wird eingegeben
3. Optionale Eigenschaften wie Bild, Beschreibung, Datum oder Tags werden eingegeben
4. Benutzer speichert Beacon

Alternativablauf:

2a. Name wird nicht eingegeben

4a. Benutzer speichert Beacon nicht

Tabelle A.1.: Use Case: Beacon anlegen

Name:	**Beacon bearbeiten**
Beschreibung:	Ein Beacon kann von einem Nifino Anwender bearbeitet und aktualisiert werden
Vorbedingung:	Benutzer ist angemeldet und hat bereits ein Beacon erstellt
Nachbedingung:	Aktualisiertes Beacon wurde gespeichert
Fehlersituation:	• Internetverbindung nicht vorhanden
Systemzustand im Fehlerfall:	Beacon wurde nicht gespeichert
Akteure:	• Benutzer • Nifino Applikation • Backend

Standardablauf:

1. Benutzer öffnet eigenes Beacon
2. Beacon wird modifiziert
3. Benutzer speichert Beacon

Alternativablauf:

3a. Benutzer speichert Beacon nicht

Tabelle A.2.: Use Case: Beacon bearbeiten

Name:	**Beacon finden**
Beschreibung:	Beacons können aus der Umgebung oder durch eine Suche gefunden werden
Vorbedingung:	Benutzer ist angemeldet und Lokalisierungsdienste sind aktiv
Nachbedingung:	
Fehlersituation:	• Internetverbindung nicht vorhanden • Lokalisierungsdienste abgeschaltet
Systemzustand im Fehlerfall:	Beacons werden nicht gefunden und nicht dargestellt
Akteure:	• Benutzer • Nifino Applikation • Backend
Standardablauf: 1. Benutzer öffnet Beaconfinder	
Alternativablauf: 1a. Benutzer passt Kategorien der Suche an 1b. Benutzer passt zu durchsuchenden Radius an 1c. Benutzer sucht den Namen eines Beacons	

Tabelle A.3.: Use Case: Beacons finden

Name:	**Benutzerprofil bearbeiten**
Beschreibung:	Benutzer können ihr eigenes Profil, bspw. das Bild oder die Beschreibung aktualisieren
Vorbedingung:	Benutzer ist angemeldet
Nachbedingung:	Benutzerprofil ist aktualisiert
Fehlersituation:	• Internetverbindung nicht vorhanden
Systemzustand im Fehlerfall:	Aktualisierung ist nicht gespeichert
Akteure:	• Benutzer • Nifino Applikation • Backend
Standardablauf: 1. Benutzer öffnet Benutzerprofil 2. Benutzer passt Profil den Bedürfnissen an 3. Benutzer speichert Änderungen	
Alternativablauf: 3a. Benutzer speichert Modifikation nicht	

Tabelle A.4.: Use Case: Benutzerprofil bearbeiten

Literaturverzeichnis

[06] „Enterprise Service Oriented Architectures: Concepts, Challenges, Recommendations“. In: Dordrecht: Springer Netherlands, 2006. Kap. Event-Driven Architecture, S. 317–355. URL: http://dx.doi.org/10.1007/1-4020-3705-8_8 (zitiert auf S. 17).

[Amr] Y. E. Amrani. *Testgetriebene Entwicklung mit JUnit*. URL: http://www.nt.th-koeln.de/fachgebiete/inf/nissen/sif/ElAmrani_Bericht.pdf (besucht am 26. 02. 2016) (zitiert auf S. 26).

[Anga] Angular. *Protractor Configuration Reference*. URL: https://github.com/angular/protractor/blob/master/docs/referenceConf.js (besucht am 05. 03. 2016) (zitiert auf S. 69, 70).

[Angb] AngularJS. *AngularJS API*. URL: https://angular.github.io/protractor/#/api (besucht am 06. 03. 2016) (zitiert auf S. 70).

[Angc] AngularJS. *AngularJS: Developer Guide: Conceptual Overview - Direktiven*. URL: https://docs.angularjs.org/guide/concepts#directive (besucht am 19. 02. 2016) (zitiert auf S. 62).

[Angd] AngularJS. *AngularJS GitHub FAQ*. URL: https://github.com/angular/protractor/blob/master/docs/faq.md (besucht am 06. 03. 2016) (zitiert auf S. 42).

[Bai15] S. Bailey. *AngularJS Testing Cookbook*. Packt Publishing, 2015 (zitiert auf S. 42).

[Bra14] R. Branas. *AngularJS Essentials*. Community Experience Distilled. Packt Publishing, 2014. URL: https://books.google.de/books?id=hh1VBAAAQBAJ (zitiert auf S. 18).

[Chu15] M. B. Chughtai. *Concept and Implementation of Digital Beacons*. 2015 (zitiert auf S. 19, 45).

[Cor] A. Cordova. *Overview - Apache Cordova*. URL: https://cordova.apache.org/docs/en/latest/guide/overview/ (besucht am 04. 03. 2016) (zitiert auf S. 18).

[Fre09] S. Freeman. *Mock Objects: A Brief History of Mock Objects*. 2009. URL: http://www.mockobjects.com/2009/09/brief-history-of-mock-objects.html (besucht am 05. 03. 2016) (zitiert auf S. 23).

[Gooa] Google. URL: http://www.google.com/about/company/philosophy/ (besucht am 15. 02. 2016) (zitiert auf S. 25).

[Goob] Google. *Set up a GCM Client App on Android*. URL: https://developers.google.com/cloud-messaging/android/client (besucht am 02. 03. 2016) (zitiert auf S. 31).

[Ion] Ionic. *Welcome to Ionic - Ionic Framework*. URL: http://ionicframework.com/docs/guide/preface.html (besucht am 04. 03. 2016) (zitiert auf S. 19).

[Jas] Jasmine. URL: http://jasmine.github.io/edge/introduction.html (besucht am 30. 01. 2016) (zitiert auf S. 36).

[Kara] Karma. URL: http://karma-runner.github.io/0.13/index.html (besucht am 30. 12. 2015) (zitiert auf S. 33).

[Karb] Karma. *Karma - FAQ*. URL: http://karma-runner.github.io/0.10/intro/faq.html (besucht am 24. 02. 2016) (zitiert auf S. 33).

[Karc] Karma. *Karma - Preprocessors*. URL: https://karma-runner.github.io/0.13/config/plugins.html (besucht am 24. 02. 2106) (zitiert auf S. 33).

[Kard] Karma. *Karma - Preprocessors*. URL: https://karma-runner.github.io/0.13/config/preprocessors.html (besucht am 24. 02. 2106) (zitiert auf S. 33).

[Kare] npm Karma Plugins. *npm packages*. URL: https://www.npmjs.com/browse/keyword/karma-plugin?offset=0 (besucht am 24. 02. 2106) (zitiert auf S. 34).

[KGTB07] N. Kicillof, W. Grieskamp, N. Tillmann, V. Braberman. „Achieving both model and code coverage with automated gray-box testing“. In: *Proceedings of the 3rd international workshop on Advances in model-based testing*. ACM. 2007, S. 1–11 (zitiert auf S. 25).

[Lan] L. Lanese. *PhantomJS used in Twitter*. URL: https://twitter.com/leolaneseltd/status/197766152745787392 (besucht am 25. 02. 2016) (zitiert auf S. 39).

[Lip14] J. Lipps. *appium - mobile automation made awesome, JQuery UK Workshop*. 2014. URL: http://appium.io/jqueryuk.pdf (besucht am 10. 01. 2016) (zitiert auf S. 40).

[LL10] J. Ludewig, H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt-Verlag, 2010. URL: https://books.google.de/books?id=Wa3qQgAACAAJ (zitiert auf S. 24, 25).

[Mar02] A. S. Martin Pol Tim Koomen. *Management und Optimierung des Testprozesses. Ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap*. dpunkt.Verlag, 2002 (zitiert auf S. 21).

[Moca] Mocha. *Mocha*. URL: https://mochajs.org/ (besucht am 29. 01. 2016) (zitiert auf S. 37).

[Mocb] Mocha. *Mocha Spies*. URL: https://github.com/mochajs/mocha/wiki/Spies (besucht am 25. 02. 2016) (zitiert auf S. 37).

[Mocc] Mocha. *MochaJA Assertions*. URL: https://mochajs.org/#assertions (besucht am 02. 03. 2016) (zitiert auf S. 37).

[Nif] Nifino. *Nifino Screenshot*. URL: http://a3.mzstatic.com/eu/r30/Purple5/v4/54/d0/95/54d09531-a319-dda3-f2c1-f0adfc4ddd92/screen322x572.jpeg (besucht am 06. 03. 2016) (zitiert auf S. 29).

[Nig] Nightwatch. URL: http://nightwatchjs.org/ (besucht am 10. 02. 2016) (zitiert auf S. 35).

[Nod] NodeJS. *NodeJS*. URL: https://nodejs.org/en/ (besucht am 04. 03. 2016) (zitiert auf S. 17).

[Pha] PhantomJS. *PhantomJS Binaries*. URL: http://phantomjs.org/download.html (besucht am 25. 02. 2016) (zitiert auf S. 40).

[Proa] Protractor. URL: https://angular.github.io/protractor/ (besucht am 24. 12. 2015) (zitiert auf S. 34).

[Prob] Protractor. *Protractor Frameworks*. URL: http://www.protractortest.org/#/frameworks (besucht am 24. 02. 2106) (zitiert auf S. 35).

[Proc] Protractor. *Protractor Plugins*. URL: https://angular.github.io/protractor/#/plugins (besucht am 24. 02. 2016) (zitiert auf S. 35).

[Prod] Protractor. *Protractor - Using Locators*. URL: https://angular.github.io/protractor/#/locators (besucht am 06. 03. 2016) (zitiert auf S. 70).

[QUn] QUnit. URL: https://qunitjs.com/ (besucht am 03. 02. 2016) (zitiert auf S. 38).

[Rip] Ripple. URL: http://incubator.apache.org/projects/ripple.html (besucht am 12. 12. 2015) (zitiert auf S. 41).

[Sel] Selenium. URL: http://www.seleniumhq.org/ (besucht am 24. 01. 2016) (zitiert auf S. 31).

[SG14] S. Seshadri, B. Green. *AngularJS: Up and Running: Enhanced Productivity with Structured Web Apps*. O'Reilly Media, 2014. URL: https://books.google.de/books?id=h5lyBAAAQBAJ (zitiert auf S. 59, 62, 64).

[Tesa] Testem. *Testem - README*. URL: https://github.com/testem/testem/blob/master/README.md (besucht am 24. 02. 2016) (zitiert auf S. 34).

[Tesb] N. U. Testing. URL: http://nightwatchjs.org/guide#unit-testing (besucht am 10. 02. 2016) (zitiert auf S. 36).

Alle URLs wurden zuletzt am 06. 03. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift