Signet farbig_ Format

Bachelorarbeit

**Entwurf und Umsetzung einer**

**Beschreibungssprache für mobile Anwendungen**

Bachelorarbeit von Edmund Senkleiter

Fakultät für Informatik   
Institut für Softwaretechnologie  
Professur für Programmierung kooperativer Systeme  
Prof. Dr. Michael Koch



betreut von:  
Peter Lachenmaier, M. Sc.

Abgabetermin der Arbeit: 14.01.2015

Hiermit versichere ich an Eides statt, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich alle verwendeten Quellen, auch Internetquellen, ordnungsgemäß angegeben habe.

Datum, Unterschrift (Name)

Kurzzusammenfassung

In dieser Bachelorarbeit geht es um die Entwicklung mobiler Anwendungen mithilfe einer hier entworfenen und umgesetzten Beschreibungssprache. Diese ermöglicht Plattform und Anwendungsszenario übergreifende Anwendungen zu erstellen. Das reduziert den Entwicklungsaufwand und ermöglicht sehr kurzfristige Änderungen. Darstellung, Inhalt und Funktionalität der Anwendung wird mithilfe der Sprache beschrieben und von einem nativ entwickelten Interpreter zur Laufzeit geladen und interpretiert. Ein solcher Interpreter wird hier für die Plattform iOS entwickelt und zur Darstellung einer Anwendung genutzt. Die Anwendungsbeschreibung erhält er dabei von einer externen REST-Schnittstelle, die in der Beispielanwendung vom CommunityMashup bereitgestellt wird.

Inhaltsverzeichnis

Kurzzusammenfassung i

Inhaltsverzeichnis ii

Abbildungsverzeichnis iv

Tabellenverzeichnis v

Abkürzungsverzeichnis vi

1 Einleitung 1

1.1 Motivation 1

1.2 Problemstellung 3

1.3 Zielsetzung 3

1.4 Aufbau der Arbeit 4

2 Entwicklung plattformübergreifender mobiler Anwendungen 6

2.1 Bewertungskriterien für Entwicklungsframeworks 6

2.2 Bewertung aktueller Entwicklungsframeworks 9

2.3 Fazit 12

3 Beschreibungssprache 14

3.1 Anforderungen 14

3.2 Entwurf 15

3.2.1 JSON Repräsentation 20

3.3 Fazit 22

4 iOS Interpreter 23

4.1 Architektur 23

4.2 Entwurf 26

4.2.1 Pakete Überblick 27

4.3 Umsetzung 29

4.3.1 Model 29

4.3.2 View 33

4.3.3 Controller 35

4.4 Interpreter für andere Plattformen 37

5 Beispiel iOSTemplateLanguage Applikation 39

5.1 Anforderungen und Voraussetzungen 39

5.2 CommunityMashup 39

5.3 Entwurf 40

5.4 Umsetzung 41

5.5 Fazit 45

5.5.1 Verbesserte Performanz 46

6 Zusammenfassung und Ausblick 48

Literaturverzeichnis 49

Abbildungsverzeichnis

Abbildung 1: Überblick CommunityMashup (Lachenmaier 2013) 2

Abbildung 2: Einstellungen App iOS, WindowsPhone, Android 15

Abbildung 3: Entwurf Beschreibungssprache 16

Abbildung 4: Detaillierterer Entwurf Beschreibungssprache 19

Abbildung 5:Apple Model View Controller 24

Abbildung 6: Paket Model Überblick 27

Abbildung 7: Paket View Überblick 28

Abbildung 8: Paket Controller Überblick 28

Abbildung 9: CoreData Daten Modell Diagramm 30

Abbildung 10: Interpreter Model 31

Abbildung 11: Interpreter Views 33

Abbildung 12: UITableView mit 2 UITableViewCells 33

Abbildung 13: Beispiel UIImageCell 34

Abbildung 14: Beispiel UITextFieldCell 34

Abbildung 15: Interpreter Controller 35

Abbildung 16: Bereich Personen in der Beispiel App 43

Abbildung 17: Bereich Personen mit geladener Liste 44

Abbildung 18: Detaillierte Ansicht einer Person in der Beispiel App 44

Tabellenverzeichnis

Tabelle 1: Bewertungskriterien zur Infrastruktur mobiler AEF und Umgebungen (nach Heitk et al. 2013) 7

Tabelle 2: Bewertungskriterien zur Entwicklung mobiler AEF und Umgebungen (nach Heitk et al. 2013) 8

Tabelle 3: Evaluationsergebnisse zur Infrastruktur mobiler AEF und Umgebungen (nach Heitk et al. 2013) 9

Tabelle 4: Evaluationsergebnisse zur Entwicklung mobiler AEF und Umgebungen (nach Heitk et al. 2013) 11

Tabelle 5: Framework Alternativen für andere Plattformen 38

Tabelle 6: iOSTemplateLanguage App Performanztest Ergebnisse 46

Abkürzungsverzeichnis

AEF Anwendungsentwicklungsframework

App Applikation

Aufl. Auflage

Bd. Band

CSS Cascading Style Sheets

HTML Hypertext Markup Language

JSON JavaScript Object Notation

MIME Multipurpose Internet Mail Extensions

MuC2014 Mensch und Computer 2014 Tagungs App

MVC Model View Controller

REST Representational State Transfer

UML Unified Modeling Language

WLAN Wireless Local Area Network

XML Extensible Markup Language

# Einleitung

## Motivation

Mobile Endgeräte wie Smartphones oder Tablets sind aus unserem Alltag nicht mehr wegzudenken. Die Verkaufsrekorde werden jährlich gebrochen und die klassischen Desktop PCs sowie Laptops langsam verdrängt (Krösmann 2014). Mobile Anwendungen gewinnen dabei natürlich auch immer mehr an Bedeutung und damit auch die Anzahl an Softwareentwicklungen in diesem Bereich. Genauso wie für die „klassischen“ Geräte, existieren auch mehrere unterschiedliche Betriebssysteme für mobile Geräte (Apple iOS, Google Android, Microsoft Windows Phone usw.). Um nun einen möglichst großen Anwenderkreis abzudecken, ist es erforderlich die Anwendung für mehrere Betriebssysteme / Plattformen zu entwickeln, was aber mit erheblichem Mehraufwand verbunden sein kann, gerade auch wenn man an Erweiterungen oder Korrekturen denkt. Die Apps können nativ für jede Plattform extra entwickelt werden, wodurch sie zwar sehr effizient ausgeführt werden und Zugriff auf alle Geräte- /Betriebssystemfunktionen bieten, dies jedoch eben nur auf der dafür entwickelten Plattform. Im Gegensatz dazu werden Web Apps im Browser ausgeführt was derzeit mit fast jedem mobilen Endgerät möglich ist, sodass hier nicht für jede Plattform extra Code geschrieben werden muss. Die grafischen Benutzerschnittstellen entsprechen jedoch eher denen einer gewöhnlichen Webseite als den nativen Schnittstellen. Einen Kompromiss zwischen beiden Varianten bieten hybride Apps, die zwar auch durch Web Technologien wie bspw. HTML5 realisiert werden, zur Laufzeit jedoch nativ in einem WebView Container laufen und somit auch Zugriff auf viele Geräte / Betriebssystemfunktionen bieten. Welche Variante die passendere ist lässt sich an Merkmalen wie Performanz, Nutzung nativer Funktionalitäten, Installation, Erreichbarkeit, offline Nutzbarkeit usw. festmachen (Svanidze 2014). Ständige Leistungssteigerungen erweitern die Entwicklungs- und somit Einsatzmöglichkeiten, jedoch werden mobile Geräte hauptsächlich zur schnellen Informationsversorgung genutzt (Knab 2014), was größtenteils nur einfaches Abrufen und Darstellen bestimmter Informationen aus einer (halb-) öffentlichen Quelle ist.

Im Rahmen dieser Arbeit wird speziell auf mobile Anwendungen eingegangen, bei denen die Darstellung der Informationen im Vordergrund steht. Die Verarbeitung bzw. Berechnung wird an anderer Stelle durchgeführt bspw. über eine REST Schnittstelle zur Verfügung gestellt. Als reales Beispiel geht es hier um Anwendungen, die personenzentrierte Daten aus einem sog. Mashup laden und dem Nutzer zur Verfügung stellen. Für die Mensch und Computer Tagung 2014 wurde bereits eine solche App (MuC2014) für iOS entwickelt und veröffentlicht (Hoferick 2014). Diese unterstützt die Teilnehmer der Tagung, indem Informationen zu Personen (Teilnehmer und Autoren), Organisationen und Inhalten (bspw. Vorträgen) anzeigt werden. Informationen sind beispielsweise:

* Kontaktdaten zu Teilnehmern, Referenten, Organisationen
* Überblick sowie detailliertere Angaben zu einem Vortrag
* Auflistung aller Vorträge zu einer bestimmten Person
* Agenda / Programmplan der Konferenz

Den Nutzern wird außerdem noch die Möglichkeit gegeben selbst Informationen, in Form von Kommentaren und Bewertungen, den Inhalten hinzuzufügen. Das eben genannte Mashup, genauer gesagt CommunityMashup, stellt eine Schnittstelle zwischen externen Social-Software Diensten, wie z.B Facebook, Twitter, Flickr, und Endbenutzer Anwendungen, wie der eben genannten App MuC2014 dar. Im Mashup werden personenzentrierte Daten gesammelt, gefiltert und vereinheitlicht über eine REST-Schnittstelle zugänglich gemacht (Lachenmaier 2013). Die Schnittstelle kann Daten in Form von HTML, XML, JSON und anderen Formaten ausgeben. Über Templates kann die Ausgabe zusätzlich individualisiert werden. Abbildung 1: Überblick CommunityMashup verdeutlicht den Aufbau:



Abbildung 1: Überblick CommunityMashup (Lachenmaier 2013)

## Problemstellung

Die Entwicklung von mobilen Endbenutzer Anwendungen für das CommunityMashup ist momentan mit viel Entwicklungsaufwand verbunden. Gesucht wird eine Möglichkeit diese einheitlich und effizient für alle möglichen Plattformen und Anwendungsszenarien zu entwickeln.

Die Mensch und Computer 2014 Tagungs App ist derzeit nur nativ für Apples Betriebssystem iOS entwickelt worden, sodass eine Portierung auf andere Plattformen wie Android oder Windows Phone mit viel Entwicklungsaufwand verbunden ist. Dasselbe Problem hat man auch bei Änderungen, die mehrere oder sogar alle angebotenen Plattformen betreffen. Außerdem müssen alle Änderungen / Updates bei Apple erneut den sog. „App Review Process“ durchlaufen der mehrere Tage dauern kann, das verhindert kurzfristige Änderungen.

Ein weiteres Problem ist die Initialisierung der App mit Daten. Dies kann auf sehr unterschiedliche Art und Weise erfolgen. Im Beispiel der MuC2014 ist der erste Start sehr langwierig da hier eine Lösung gewählt wurde, bei der zu Beginn alle Informationen abgerufen werden. Das hat den Vorteil, dass die App danach auch fast vollständig offline genutzt werden kann. Der entscheidende Nachteil ist jedoch eben die Dauer der Initialisierung bzw. des ersten Starts. Bei großen Datenmengen und schlechter Konnektivität kann diese mehrere Minuten dauern, währenddessen die App nicht nutzbar ist. Ein solches Verhalten kann bspw. auch zur Ablehnung der App durch das Apple „App Review Process“ Team führen.

## Zielsetzung

Ziel dieser Arbeit ist der Entwurf sowie die Umsetzung einer Beschreibungssprache für mobile Anwendungen um die zuvor genannten Probleme zu lösen. Die Sprache ermöglicht für alle Plattformen eine gemeinsame Anwendungsbeschreibung, die jedoch plattformspezifisch interpretiert wird, sodass nur der Interpreter plattformabhängig entwickelt werden muss. Im Implementierungs- bzw. Umsetzungsteil dieser Arbeit wird dieser nur für iOS als Beispiel und zur Evaluation entwickelt. Weiterhin soll der iOS-Interpreter ein Framework an abstrakten Grundfunktionalitäten anbieten, die über die Sprache den einzelnen grafischen Benutzerschnittstellenelementen zugeordnet werden können. Unter abstrakten Grundfunktionalitäten kann man sich Einträge ins Telefonbuch, das Öffnen eines Links, den Wechsel zu einer anderen Ansicht vorstellen. Darauf wird später noch genauer eingegangen.

Mit Beschreibungssprache sollen Aussehen, Inhalte sowie mögliche Funktionalitäten der App plattformunabhängig definiert werden können. Die Anwendungsbeschreibung soll dann von einer externen Quelle wie dem CommunityMashup heruntergeladen und zur Laufzeit interpretiert werden, was bedeutet, dass das Mashup keine reine Informationsquelle ist, sondern auch das Aussehen sowie die möglichen Funktionalitäten der App mithilfe von Templates über eine REST Schnittstelle definiert. Auf diese Art und Weise können jegliche Korrekturen oder Verbesserungen, die nicht den Sprachumfang bzw. dessen Interpretation betreffen, schnell und mit geringem Aufwand für alle Plattformen gleichzeitig umgesetzt werden. Weiterhin bietet sie den Vorteil, für verschiedene Anwendungsszenarien denselben Interpreter verwenden zu können. Möchte man bspw. für eine andere Tagung andere Informationen und Darstellungen anbieten, hat dies keinen Einfluss auf die nativen Interpreter Implementierungen sondern ändert nur etwas an der Anwendungsbeschreibung.

Ein weiteres Ziel ist die Verkürzung der App Initialisierung durch sog. „lazy loading“ und „caching“. Bei ersterem werden nur dann Beschreibungsinformationen heruntergeladen werden, wenn sie das erste Mal gebraucht oder eine Aktualisierung gewünscht bzw. notwendig ist. Der zweite Begriff bezeichnet das lokale Speichern bereits geladener Informationen damit diese in Zukunft schneller und auch offline verfügbar sind. Damit reduzieren sich die benötigten Informationen bei Anwendungsstart auf das Notwendigste, nämlich der ersten Ansicht. Ein weiteres Ziel ist die entwickelte Sprache sowie den dazugehörigen Interpreter für iOS zu nutzen um eine App nach Vorbild der MuC2014 umzusetzen. Dabei liegt der Schwerpunkt nicht darauf, die App in derselben Form einzusetzen bzw. zu veröffentlichen sondern zu zeigen, dass mit der Sprache alle grundlegenden Funktionen umgesetzt und die Benutzbarkeit, insbesondere die Ladezeiten sich verbessert haben.

## Aufbau der Arbeit

In Kapitel 2 Entwicklung plattformübergreifender mobiler Anwendungen werden aktuelle Beschreibungssprachen anhand einiger Bewertungskriterien näher untersucht. Abschließend wird begründet warum eine eigene Beschreibungssprache entwickelt werden soll.

Die Entwicklung der Beschreibungssprache wird in Kapitel 3: Beschreibungssprache vorgestellt. Dazu werden zunächst alle Anforderungen beschrieben, die sich aus Kapitel 2 und den Zielen der Arbeit ergeben. Im folgenden Entwurf werden diese dann umgesetzt und eine Sprache basierend auf JSON definiert.

Die Evaluation der entworfenen Sprache beginnt bereits in Kapitel 4 mit der Entwicklung eines Interpreters für iOS. Hierfür wird der Vorteil einer bekannten Sprache (JSON) bzw. dessen Syntax als Grundlage für die eigene Sprache genutzt. Ein Framework übernimmt das Parsen sowie die Speichern der Daten.

Nachdem der Interpreter entworfen ist, folgt die Umsetzung einer Beispielanwendung nach Vorbild der MuC2014 App (Kapitel 5: Beispiel iOSTemplateLanguage Applikation). Hierbei werden wieder Anforderungen und die Verbindung zum CommunityMashup erläutert. Anschließend folgen Entwurf und Umsetzung der App. Zum Schluss gibt es noch ein Fazit das insbesondere auf den erreichten Funktionsumfang und die verbesserte Performanz eingeht.

Im letzten Kapitel Zusammenfassung und Ausblick werden die zu Beginn genannten Ziele in Zusammenhang mit den erreichten Ergebnissen gebracht und ein Ausblick auf weitere Entwicklungen sowie Verbesserungen gegeben.

# Entwicklung plattformübergreifender mobiler Anwendungen

Für die Entwicklung plattformübergreifender mobiler Anwendungen gibt es 3 verschiedene Ansätze (Heitk et al. 2013):

1. Nutzung einer Beschreibungssprache die zur Laufzeit interpretiert wird
2. Nutzung einer gemeinsamen Codebasis, die plattformspezifisch übersetzt wird
3. Modellgetriebener Ansatz

Der erste Ansatz entspricht den sog. Web und hybriden Apps. Die App wird durch eine gemeinsame Beschreibungssprache für alle Plattformen entwickelt und erst zur Laufzeit interpretiert. Dazu muss jedoch für jede Plattform der Interpreter nativ entwickelt worden sein. Web Apps nutzen diesen Ansatz, wobei als Beschreibungssprachen HTML5, CSS und JavaScript verwendet werden und der Interpreter der native Webbrowser ist. Um deutlich mehr Geräte- / Betriebssystemfunktionen nutzen zu können, kann hier ein eigener Interpreter entwickelt werden, worauf in Kapitel 4 näher eingegangen wird.

Der zweite Ansatz ähnelt dem ersten, jedoch wird die Beschreibung zunächst plattformspezifisch übersetzt und damit gleichzeitig für alle Plattformen eine native Anwendung erstellt. Das hat natürlich den Vorteil, dass nicht erst zur Laufzeit interpretiert werden muss und die Anwendung somit deutlich effizienter ausgeführt wird. Der Ansatz wird jedoch nicht weiter betrachtet, da hier der bereits erwähnte langwierige und umständliche Updateprozess unumgänglich ist.

Der dritte Ansatz versucht, wie der Name schon sagt, über eine Modelldefinition plattformübergreifende Apps zu generieren, wird aber aufgrund fehlender marktreifer Lösungen nicht weiter betrachtet.

Im Folgenden werden zunächst Bewertungskriterien für Entwicklungsframeworks die dem ersten Ansatz entsprechen vorgestellt und anschließend aktuelle Lösungen mithilfe dieser bewertet.

## Bewertungskriterien für Entwicklungsframeworks

Die folgenden 14 Kriterien wurden von Heitk et al. (2013) aus verschiedenen Quellen, wie Softwarefirmen, Literatur, Zusammenstellung von Problemen aus Online Communities sowie Erfahrungen aus der Entwicklung eigener Prototyp Apps zusammengestellt. Die Kriterien werden in 2 Kategorien unterteilt: Infrastruktur und Entwicklung. Mit Infrastruktur sind Kriterien gemeint, die zur Laufzeit von Bedeutung sind wie z.B Benutzbarkeit, Aussehen und Funktionsumfang. In der Kategorie Entwicklung befinden sich Kriterien zu den angebotenen / verwendeten Tools, der Wartbarbeit usw. Die folgende Tabelle listet die ersten 7 Kriterien der Kategorie Infrastruktur auf, dabei stehen links die Kategorie und rechts dazugehörige Unterpunkte bzw. Fragen.

Tabelle : Bewertungskriterien zur Infrastruktur mobiler AEF und Umgebungen (nach Heitk et al. 2013)

|  |  |
| --- | --- |
| Lizenz und Kosten | Welche Lizenz wird verwendet?  Freie Software / evtl. sogar Open Source  Nutzung für kommerzielle / private Zwecke  Supportkosten |
| Unterstütze Plattformen | Welche Plattformen werden unterstützt?  Wie gut werden diese unterstützt? |
| Zugang zu plattformspezifischen Funktionen | Gibt es Zugang zu Hardwarekomponenten wie GPS, Kamera usw. ?  Vergleich zu nativer und Web App |
| Zukunftsfähigkeit | Kurze sowie reguläre Updatezyklen  Unterstützung aktueller mobiler Betriebssysteme  Aktive Community sowie viele Entwickler  Kommerzielle Unterstützer |
| Erscheinungsbild und Benutzbarkeit | Unterstützung nativer / Web Benutzerschnittstellen  Anwendungen eigenen sich zur mobilen Nutzung |
| Anwendungsgeschwindigkeit | Benötigte Startzeit  Geschwindigkeit zur Laufzeit |
| Distribution | Veröffentlichungsprozess  Nutzung der plattformspezifischen App Stores  Updatemöglichkeiten |

Die folgende Tabelle listet die 7 Bewertungskriterien der Kategorie Entwicklung auf:

Tabelle : Bewertungskriterien zur Entwicklung mobiler AEF und Umgebungen (nach Heitk et al. 2013)

|  |  |
| --- | --- |
| Entwicklungsumgebung | Funktionalitäten (Autovervollständigung)  Vorhandene / unterstütze Tools (IDE , Debugger, Emulator)  Einfachheit der Umgebungseinrichtung |
| GUI Design | Möglichkeiten Benutzerschnittstellen zu erstellen (Tools zur Unterstützung und Test)  GUI Vorschau vorhanden? |
| Einfachheit der Entwicklung | Qualität der API und Dokumentation  Erlernbarkeit (Tutorials, Beispiele)  Nutzung bekannter Softwareentwicklungsparadigmen |
| Wartbarbeit | Zur Evaluation wird die Anzahl der benötigten Codezeilen verwendet. |
| Skalierbarkeit | Unterstützung größerer Entwicklungsteams und Projekte  Modularität des Frameworks und der Anwendung |
| Möglichkeiten für zukünftige Entwicklungen | Wiederverwendung von Code |
| Entwicklungsdauer und Kosten | Dauer des Entwicklungsprozesses sowie Faktoren die ihn behindern  Kosten werden nicht explizit angegeben, stehen in Zusammenhang mit der Entwicklungsdauer |

## Bewertung aktueller Entwicklungsframeworks

Die Studie „Evaluation of mobile Web applications“ (Heit et al. 2013) bewertet 4 aktuelle Lösungen / Frameworks im Hinblick auf die zuvor genannten Kriterien. Als vierte Lösung werden die nativ entwickelten Anwendungen betrachtet, auf die hier jedoch nicht mitaufgeführt werden, da diese nicht dem ersten Ansatz entsprechen weil sie keine Beschreibungssprache zur Laufzeit interpretieren. Die erste Lösung sind die gewöhnlichen Web Apps. Die zweite ist ein bekanntes Open Source Framework für die Entwicklung von hybriden Apps: PhoneGap[[1]](#footnote-1). Die Anwendungen werden mit HTML, CSS und JavaScript entwickelt, jedoch wird der Webinterpreter um eine JavaScript API in Kombination mit einer Brücke zur Hardware erweitert. Ein Beispiel für die Nutzung eines eigenen Interpreters liefert das Framework Titanium Mobile[[2]](#footnote-2). Die Benutzerschnittstelle, Logik und Daten werden in JavaScript programmiert und zur Laufzeit interpretiert. Es folgen nun die Evaluationsergebnisse:

Tabelle : Evaluationsergebnisse zur Infrastruktur mobiler AEF und Umgebungen (nach Heitk et al. 2013)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Web Apps | | Phone Gap | Titanium Mobile | |
| Lizenz und Kosten | | | | |
| Meisten Tools und Frameworks Open Source  Support durch große Communities  Kosten für den Betrieb der Webseite | | Open Source Software unter Apache Lizenz 2.0 / GPL/MIT Lizenz  Anbieter verkauft Support zwischen 25$ und 2000$ pro Monat | Community Edition ist kostenlos, der Funktionsumfang jedoch limitiert  Proprietäre Edition enthält alle Funktionen sowie Support  Eher geschlossenes System | |
| Unterstütze Plattformen | | | | |
| Alle Plattformen mit Webbrowser | | iOS, Andorid, BlackBerry OS, Windows Phone 8, Ubuntu, Firefox OS (Phone Gap 2014) | iOS, Android, BlackBerry (Titanium Mobile 2014) | |
| Zugang zu plattformspezifischen Funktionen | | | | |
| Verbessert durch HTML5: lokaler Speicherplatz begrenzt nutzbar  Video, Audio Dateien abspielbar, Nutzung Multi Touch Gesten | | Zugang zu fast allen Hardwarefunktionen[[3]](#footnote-3) | Ähnlich zu Phone Gap | |
| Zukunftsfähigkeit | | | | |
| Ja - Nutzung fest etablierter Standards (HTML, CSS, Java Script) | Aktive Community, Unterstützer: Adobe und IBM, als Apache Projekt geführt, regelmäßige Updates | | | Große Community und viele Entwickler, folgt aktuellen Trends, nutzt neue Funktionalitäten der Betriebssysteme, regelmäßige Updates  Abhängig von einer einzigen Firma |
| Erscheinungsbild und Benutzbarkeit | | | | |
| CSS definiert Erscheinungsbild, native Benutzerschnittstellenelemente nicht möglich  HTML5 verbessert Benutzbarkeit durch lokale Speichernutzung (bspw. Wiederherstellung Appzustand nach Pause) | Nutzt keine nativen Benutzerschnittstellenelemente, CSS definiert plattformspezifisches Aussehen(hoher Aufwand)  Nutzt Events um Laufzeitveränderungen zu behandeln (z.B bei Anruf, Pause) | | | Über Java Script werden native Benutzerschnittstellenelemente genutzt – erfordert viel Kenntnisse der API  Gute Benutzbarkeit implementierbar |
| Anwendungsgeschwindigkeit | | | | |
| Abhängig von der Netzwerkverbindung, optimierte Browser ermöglichen performante Ausführung | Vergleichbar mit nativer App | | | Startzeit ähnlich zu anderen Frameworks, zur Laufzeit jedoch Probleme bei zu vielen Objekten |
| Distribution | | | | |
| Sehr einfach: den Nutzern die URL mitteilen | Normalerweise im jeweiligen App Store (Apple verbietet reine Web Apps) | | | Im jeweiligen App Store |

Tabelle : Evaluationsergebnisse zur Entwicklung mobiler AEF und Umgebungen (nach Heitk et al. 2013)

|  |  |  |
| --- | --- | --- |
| Entwicklungsumgebung | | |
| Nutzung vorhandener Umgebungen aus dem Bereich Webentwicklung | Ähnlich zu Web Apps, nicht jede IDE unterstützt die PhoneGap API | Eclipse basierte DIE Titanium Studio, Autovervollständigung für die Titanium API, unterstützt Ausführung sowie Veröffentlichung im App Store, einfache Einrichtung |
| GUI Design | | |
| Identisch zur Webentwicklung | Identisch zur Webentwicklung (z.B Adobe Dreamweaver) | Mit JavaScript programmiert – mühsam und zeitaufwendig, kein Editor vorhanden |
| Einfachheit der Entwicklung | | |
| Viele Dokumentationen und Beispiele / Tutorials in HTML, CSS und Java Script , Beachtung Limits mobiler Geräte (CPU, GPU usw.) | Gute Dokumentation, viele Beispiele, wenig Framework spezifisches Wissen | Gute Dokumentation, wenig Codebeispiele, viel Framework spezifisches Wissen erforderlich |
| Wartbarkeit | | |
| Wenig Codezeilen nötig | Ähnlich Web Apps, für Hardwarezugang wird extra Code benötigt, dennoch insgesamt sehr kurz und klar strukturiert | Viele Codezeilen, jedoch sehr modular aufgebaut |
| Skalierbarkeit | | |
| Modularisierbar, Aufteilung in mehrere Dateien | Siehe Web Apps | sehr einfache Modularisierung |
| Möglichkeiten für zukünftige Entwicklungen | | |
| Gut, einfach portierbar zur PhoneGap App, kann auch im WebView von Titanium Mobile angezeigt werden | Ohne Nutzung von plattformspezifischer Funktionen ähnlich Web Apps | Schlecht da viel Titanium spezifischer Code benutzt wird |
| Entwicklungsdauer und Kosten | | |
| Web App Prototype Entwicklung war am kürzesten, mächtige Entwicklerwerkzeuge beschleunigen und begünstigen den Prozess | Ähnlich zu Web Apps mit zusätzlichem Entwicklungsaufwand für plattformspezifische Funktionalitäten | Viel spezielle Erfahrung nötig, verlangsamte Entwicklung der UI aufgrund der notwenigen Ausführung im Emulator oder auf dem Gerät |

## Fazit

Die Ergebnisse zeigen die Stärken und Schwächen der unterschiedlichen Frameworks und des rein webbasierten Ansatzes. Es gilt nun zu prüfen inwiefern die aktuellen Lösungen für die eigene verwendetet werden können. Die Web App ist sehr einfach umzusetzen und die mobilen Browser nutzen bereits das Anfangs erwähnte „caching“ und „lazy loading“. Dafür bietet eine Web App keinen Zugriff auf Funktionen wie das Telefonbuch, sodass hier das Anlegen eines neuen Kontakts nicht umgesetzt werden kann. PhoneGap und Titanium Mobile hingegen bieten den gewünschten Zugriff auf solche Geräte / Betriebssystem Funktionalitäten. PhoneGap überzeugt gerade im Punkt Einfachheit der Entwicklung, bietet jedoch keine nativen Benutzerschnittstellenelemente an, sodass die App wie eine Web App aussieht. Mit Benutzerschnittstelle ist im weiteren Verlauf der Arbeit immer die grafische gemeint. Titanium hingegen bietet diese nativen Elemente an und ist in vielen anderen Punkten mit Phone Gap auf einer Ebene. Das größte Problem bei Titanium liegt jedoch im Punkt Lizenz und Kosten. Der nicht freie Quellcode ermöglicht keine Anpassungen bzw. Optimierungen für spezielle Anforderungen oder Verwendungen. Weiterhin ist die Nutzung der vollständigen Funktionalität mit Kosten verbunden.

Daraus ergeben sich einige Gründe für ein eigenes Framework, das eine vereinfachte Sprache nutzt und mit einem daraufhin entwickelten Interpreter arbeitet. Die Entwicklung einer eigenen Beschreibungssprache ermöglicht eine sehr einfache und präzise auf die benötigten Funktionalitäten und Eigenschaften angepasste Anwendungsentwicklung. Im Kontext des CommunityMashup und dessen Endbenutzerschnittstellen bedeutet das, dass der Schwerpunkt auf der Darstellung der Inhalte / Informationen liegt und nicht auf deren Verarbeitung oder Berechnung. Weiterhin ist sehr wenig Funktionalität bzw. Logik erforderlich um den Endbenutzer auch interagieren zu lassen. Dies führt zu der Idee einer eigenen, speziell angepassten Beschreibungssprache, die im nächsten Kapitel entworfen wird.

# Beschreibungssprache

In diesem Kapitel geht es um die Entwicklung der Anfangs erwähnten Beschreibungssprache. Dazu werden die bereits genannten Anforderungen klarer definiert und erweitert. Anschließend folgt der Entwurf der Sprache. Diese wird zunächst mithilfe von UML Diagrammen dargestellt, woraus anschließend die JavaScript Object Notation (JSON) Darstellung abgeleitet und präsentiert wird.

## Anforderungen

Die Anforderungen an die Beschreibungssprache lassen sich leicht aus Kapitel 2 und dem Ziel der Arbeit ableiten. Der Schwerpunkt ist hierbei die reine Beschreibung (Deklaration) des Aufbaus und nicht die Definition eines bestimmten Verhaltens. Die Sprache dient in erster Linie dazu, bereits vorhandene Informationen darzustellen und nicht diese erst zu generieren. Eine Schwierigkeit und zugleich sehr wichtige Anforderung ist eine gemeinsame und möglichst generische Beschreibungsform für alle Plattformen zu finden. Dazu ist die Nutzung gemeinsamer Benutzerschnittstellen (bspw. Listen, Tabellen) von Vorteil und nicht gemeinsamer / sehr plattformspezifische Benutzerschnittstellen eher zu vernachlässigen. Der Sprachumfang sollte möglichst gering und einfach sein, sodass keine Programmierkenntnisse erforderlich sind und somit die gesamte App Entwicklung sehr einfach gehalten wird. Um neben der reinen Darstellung von Informationen auch die Möglichkeit zu haben bestimmte vordefinierte Aktionen / Events auszuführen sowie Informationen zu senden, müssen den einzelnen Benutzerschnittstellen Aktionen zugeordnet werden können. Die Beschreibung bzw. Zuordnung sollte sehr generisch bzw. flexibel gestaltet sein, um den Funktionsumfang schnell und einfach erweitern zu können. Dies führt zu einer wichtigen Anforderung, der Erweiterbarkeit. Die Sprache soll auch für zukünftige Benutzerschnittstellen und benötigte Aktionen genutzt werden können, sodass eine Erweiterung um diese ohne großen Aufwand möglich sein muss. Weiterhin ist die Nutzung der Syntax einer bereits vorhandenen Sprache sinnvoll, um die Entwicklung eines eigenen Parsers zu umgehen. Auf dieser sog. Grundsprache basiert dann die eigentliche Beschreibungssprache. Im Folgenden werden die Anforderungen noch einmal aufgelistet.

* Beschreibung von Darstellung und Funktionalität
* Nutzung gemeinsamer Benutzerschnittstellen
* Nutzung einer bereits vorhandenen Sprachsyntax
* Zuordnung von Aktionen zu Benutzerschnittstellenelementen
* Einfachheit der Sprache
* Erweiterbarkeit

## Entwurf

Unter den in 3.1 genannten Anforderungen wurde eine Beschreibungssprache entwickelt, die im Folgenden vorgestellt wird. JSON[[4]](#footnote-4) dient dabei als Grundlage auf der die Beschreibungssprache aufgesetzt ist. Der Vorteil sind die vorhandenen Parser bzw. Frameworks, bspw. RESTKit, auf das später (4.1 Architektur) noch genauer eingegangen wird. XML wäre hier auch eine Option, jedoch ist JSON deutlich performanter und benötigt weniger Ressourcen (Nurseitov et al. 2009). Weiterhin sind JSON Dateien von der Größe her deutlich kleiner, was natürlich erheblichen Einfluss auf die Übertragungszeit hat. Um nun möglichst einfach Apps beschreiben zu können, werden Benutzerschnittstellen aus den bekannten Betriebssystem iOS (Apple Inc. 2014), Android (Google Inc. 2015) und Windows Phone (Microsoft Corporation 2015) verwendet. Dazu gehören Reiter (Tabs), Ansichten (Views), Tabellen mit Zellen und Schaltflächen (Buttons). Die Muc2014 diente ebenfalls als Vorbild bei der Auswahl der Benutzer-Schnittstellen. Im Folgenden werden die englischen Bezeichner für die Schnittstellen verwendet. Die Nutzung dieser Schnittstellen erkennt man auch an einigen bekannten Apps wie WhatsApp[[5]](#footnote-5), Facebook App[[6]](#footnote-6), Telefonbuch App[[7]](#footnote-7)(iOS) sowie die Einstellungen App. Die folgende Abbildung zeigt als Beispiel die Einstellungen App der drei Plattformen iOS, Windows Phone und Android um die Nutzung von Listen zu verdeutlichen. Jeder Eintrag ist ein Listenelement. Die Tabs sind bei dieser Abbildung nur beim WindowsPhone sichtbar.

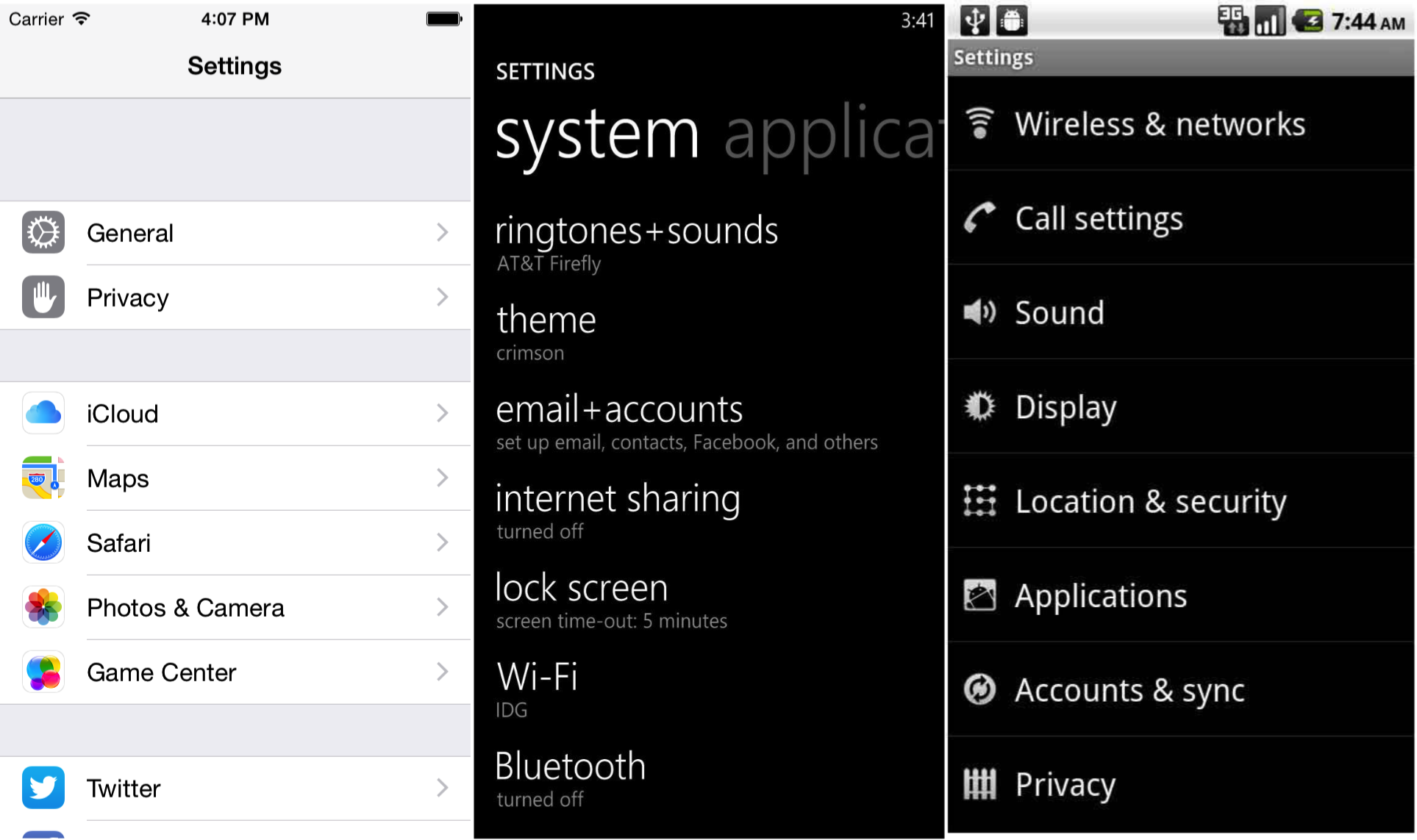


Abbildung 2: Einstellungen App iOS[[8]](#footnote-8), WindowsPhone[[9]](#footnote-9), Android[[10]](#footnote-10)



Abbildung 3: Entwurf Beschreibungssprache

Die Beschreibung der App beginnt mit einer Kollektion an Tabs. Dies ist eine Methode um Inhalte in bestimmte Kategorien / Reiter zu kapseln. Jeder Tab besitzt eine sog. Hauptview (oder auch Startview), die als erste dargestellt wird und wiederum, über Actions, auf andere Views verweisen kann. Die Views sind eigentlich Tabellen die aus beliebig vielen Sektionen (Sections) mit Zellen (Cells) bestehen. Diesen Cells können Buttons oder direkt Actions zugeordnet werden. Der grobe Entwurf für die Sprache steht fest und muss nun verfeinert werden um deutlich vielfältiger und präziser Anwendungen beschreiben zu können.

Bei dem Feinentwurf wird sich neben dem Ziel der Arbeit und der MuC2014 noch an den Apple iOS Human Interface Guidelines (Apple Inc. 2014) und Erfahrungen aus der App Entwicklung orientiert. Die Guidelines anderer Plattformen werden nicht miteinbezogen. Das bedeutet jedoch nicht, dass damit die Sprache nicht plattformübergreifend wäre. Es gilt im Einzelfall zu prüfen welche weitere Attribute andere Plattformen benötigen oder wünschenswert wären um eine detailliertere Beschreibung zu ermöglichen. Der Feinentwurf beinhaltet ebenfalls nicht alle Attribute für iOS um die angebotenen Möglichkeiten der Darstellung von Benutzerschnittstellen abzudecken. Das ist auch nicht Ziel des ersten Entwurfs der Sprache.

Ein Tab hat einen Titel und den Namen (viewPath) seines Hauptviews. Der viewPath kann bspw. auch eine URL sein, von wo aus die Beschreibung für den Hauptview heruntergeladen werden kann. Weiterhin kann hier über das Attribut autoLoadView angegeben werden. Dessen Wert bestimmt, ob der Inhalt des Views automatisch geladen wird oder ob der Ladezeitpunkt durch bspw. den Benutzer manuell festgelegt wird. In Hinblick auf die Entwicklung der Beispiel iOSTemplateLanguage Applikation können noch Icons für den selektierten und nicht selektierten Zustand des Tabs angegeben werden. Ein View hat einen Identifier, der dem eben genannten viewPath entspricht, um ihn eindeutig zu identifizieren und Verweise darauf zu ermöglichen. Weiterhin kommt ein optionaler Titel hinzu. Eine Section hat einen optionalen Titel und hält sonst nur noch eine geordnete Kollektion ihrer Cells. Eine Cell besitzt das Attribut *type*, womit der genaue Zelltyp angeben werden muss. Je nach Zelltyp kommen noch weitere Attribute hinzu. Weiterhin ist hier bereits das Attribut *styleClass* vorhanden, womit der Cell ein individuell definiertes Aussehen gegeben werden kann. Bei allen anderen darzustellenden Sprachelementen wurde dies der Einfachheit halber weggelassen.

#### Cell Typen

In diesem Entwurf wurden bereits drei unterschiedliche Cell Typen mitentworfen:

* *CellPrototype* – Standard Zelle mit Titel, Text und evtl. einem Bild (Abbildung 12: UITableView mit 2 UITableViewCells)
* *CellImage* – Stellt ein Bild zentriert in der Zelle dar (Abbildung 13: Beispiel UIImageCell)
* *CellTextField* – Hält ein Textfeld in dem Text eingeben werden kann, sowie einen Button dessen Action den Text als Parameter erhält. Eine sinnvolle Action wäre *ActionPostTextWithURL*, welche im nächsten Abschnitt vorgestellt wird. (Abbildung 14: Beispiel UITextFieldCell)

#### Action Typen

Die den Cells zuordenbaren Actions haben einen Typ sowie dessen jeweilige Attribute. Hier wurden bereits 5 konkrete Actions entworfen, die von den Attributen her meistens identisch sind, jedoch keiner gemeinsamen Action zugeordnet sind.

* *ActionOpenAppWithURL* – Öffnet eine andere Applikation, wenn auf dem Gerät vorhanden, über eine angegebene URL.
* *ActionSegueIntoView* – Wechselt zu einer anderen View. Das *target* Attribut ist hier der *identifier* des Ziel Views. Der Wert von *autoLoadView* entscheidet wie bei den Tabs auch, ob beim Übergang der darzustellende View automatisch geladen werden soll oder nicht.
* *ActionAddToContacts* – Fügt einen neuen Kontakt zum Telefonbuch hinzu. Die Action fügt momentan nur einfache Standartangaben(Name, Telefonnummer, Webadresse) hinzu und sollte noch erweitert werden.
* *ActionPostTextWithURL* – Die Action erhält über einen Parameter einen Text, fügt ihn in das URL Attribut ein und öffnet diese anschließend.
* *ActionWebViewWithURL* – Öffnet den Browser des Geräts mit einer über das URL Attribut definierten Webseite.

#### Button Typen

Einer Cell vom Typ *CellTextField* muss ein Button zugeordnet werden, wovon es in diesem Entwurf nur einen konkreten gibt, den ButtonStandard. Dieser erbt von seiner Oberklasse den Button Typ sowie Text. Dem Button selbst muss noch eine Action zugeordnet werden, die bei Berührung ausgeführt wird. Im folgenden Diagramm, das den Aufbau verdeutlichen soll, gibt es den Datentyp String?. Dieser stellt ein optionales String Attribut dar, welches ebenfalls in Apples Programmiersprache Swift[[11]](#footnote-11) so gekennzeichnet wird.



Abbildung 4: Detaillierterer Entwurf Beschreibungssprache

### JSON Repräsentation

Die Beschreibungssprache nutzt JSON als Sprachgrundlage und stellt sich wie folgt dar: am Anfang steht das App Objekt dem die Bedeutung des Startsymbols zukommt und anschließend folgt ein JSON Array an möglichen Tabs. Ein Beispiel dazu folgt. Vorher wird zunächst auf einige grundlegende Repräsentationen eingegangen. Ein Attribut wird in JSON folgendermaßen repräsentiert:

”<Attributname>” : ”<Attributwert>“

Mehrere Attribute werden durch ein Komma getrennt. Kompositionen starten mit ihrem Namen worauf die gewöhnliche JSON Objekt Beschreibung mit den jeweiligen Attributen folgt:

”<Kompositionsname>” : {}

Ein einzelner Tab mit einem Icon Objekt sieht demnach folgendermaßen aus (der graue Code stellt Kommentare dar, die dem Verständnis dienen und nicht Bestandteil der Sprache sind):

{

”id” : ”<Nummer>”, //Attribut

”title” : ”<TitelString>“,

”viewPath” : ”<viewPathString>,

"autoLoadView" : <boolean>,

”icons” : { //Komposition

”unselected“ : ”<iconString1”,

”selected” : ”<iconString2”

}

}

Es folgt nun ein Beispiel einer konkreten Anwendungsbeschreibung mit 2 Tabs. Im weiteren Verlauf wird jedoch nur der View des ersten Tabs beschrieben.

{

"app": { //Startsymbol der Appbeschreibung

"tabs": [ //Array an tabs

{

"id" : "0",

"title": "Titel1",

"viewPath":"viewIdentifier1",

"autoLoadView" : "false",

"icons": {

"unselected": "icon-unselected1",

"selected": "icon-selected2"

}

},

{ //Beginn des zweiten Tabs

"id" : "1",

"title":"Titel2",

"viewPath":"viewIdentifier2",

"autoLoadView" : "false",

"icons": {

"unselected": "icon-unselected2",

"selected": "icon-selected2"

}

}

]

}

}

Ein View beginnt in JSON mit dem Startsymbol „view“. Danach stehen die eben beschriebenen (Abbildung 4: Detaillierterer Entwurf Beschreibungssprache) Objektrepräsentation für alle Attribute und Kompositionen des Views. Ein Beispiel für den Hauptview vom Tab Nummer 0 wäre:

{

"view": {

"identifier": "viewIdentifier1",

"title": "Titel1",

"sections": [

{

"title": "Sektion1",

"cells": [

{

"type": "prototype",

"title": "Name1",

"detailTitle": "Herkunft1",

"image": {},

"action": {

"type": "segueIntoView",

"autoLoadView" : "true",

"target": "viewIdentifier3"

}

}

]

}

]

}

}

Dieser View lädt bei Anzeige nicht automatisch seinen Inhalt da im zugehörigen Tab der Wert von autoLoadView auf false gesetzt wurde. Im View existiert nur eine Section mit der Überschrift / Titel „Sektion1“. Diese hat nur eine Cell vom Typ CellPrototype mit dem Titel „Name1“ und dem detaillierteren Text „Herkunft1“. Ein Bild ist nicht vorhanden, jedoch eine Action die bei Berührung der Zelle zu einem anderen View mit der Identifikation „viewIdentifier3“ wechselt. Dieser View hingegen wird bei Auslösung der Action sofort angezeigt.

## Fazit

Die Nutzung der anderen Actions oder Zelltypen erklärt sich anhand des Diagramms (Abbildung 4: Detaillierterer Entwurf Beschreibungssprache) von selbst, was gerade zur Einfachheit der Sprache beiträgt. Erweiterungsmöglichkeiten ergeben sich bei den Zelltypen, Aktionstypen sowie Schaltflächentypen. Diese sind in dem ersten Entwurf bewusst gering gehalten worden und reichen aus, um Teile der Muc2014 App im späteren Verlauf dieser Arbeit umzusetzen. Eine mächtige und ausgereifte Sprache ist nicht Ziel dieser Arbeit, sondern eher Konzepte und Möglichkeiten zu finden, auf denen später weiter aufgebaut werden kann. Weitere Ergänzungen, insbesondere Attribute der einzelnen Klassen, sind nötig um deutlich detailliertere Anwendungsbeschreibungen zu ermöglichen, jedoch sollte die Komplexität dabei nicht außer Acht gelassen werden. Jede Änderung an der Sprache bringt auch Änderungen an allen entwickelten Interpretern mit sich, wovon die iOS Variante im nächsten Kapitel entworfen und umgesetzt wird.

# iOS Interpreter

In diesem Kapitel geht es um die Entwicklung eines iOS Interpreters für die in Kapitel 3 entworfene Beschreibungssprache. Ziel ist es einen voll funktionsfähigen Interpreter zu entwerfen, sodass dieser in Kapitel 5 für die Entwicklung einer App benutzt werden kann. Der Interpreter ist eine nativ entwickelte App, welche zur Laufzeit die Anwendungsbeschreibung lädt und interpretiert. Das sog. „caching“ und „lazy loading“ wird hier ebenfalls realisiert, sodass die Beschreibung, nachdem sie zunächst extern heruntergeladen wurde auch lokal vorliegt und nicht ständig neu geladen werden muss. Dies ermöglicht Teile der Applikation auch offline zu nutzen. Im nächsten Abschnitt werden zunächst Architektur und verwendeten Technologien / Frameworks vorgestellt.

## Architektur

Da der Interpreter nativ auf Apples Betriebssystem iOS laufen soll, ist ein Teil der Architektur und der verwendbaren Technologien / Frameworks vorgegeben.

#### Apple iOS

Als iOS Version wird hier 8.1 verwendet, welche im Dezember 2014 erschienen ist. Damit ist auch die Wahl der Programmiersprache verbunden, diese fällt nämlich auf Swift. Apple führte diese Sprache mit iOS 8 im September ein und ergänzt damit die alte Sprache Objective-C. Mit Ergänzen ist gemeint, dass beide Sprachen gleichzeitig in einer Anwendung verwendet werden können. In Swift können also auch Frameworks, die in Objective-C geschrieben worden sind, verwendet werden. Besonderheiten von Swift[[12]](#footnote-12) sind:

* Typinferenz / Ableitung
* Angabe von Modulen anstatt Header-Dateien
* Closures
* Tupel sowie multiple Rückgabewerte
* Generics
* Funktionale Programmiermuster (bspw. map und filter)

Für die App Entwicklung unter iOS wird ein Mac mit Mac OS X 10.9.4 oder neuer benötigt, sowie XCode 6 als Entwicklungsumgebung. Die Umgebung kann kostenlos über den AppStore heruntergeladen werden und bietet neben einem Quelltexteditor und GUI Editor noch viele weitere Features (Apple Inc. 2014). Ein Gerätesimulator für iPhones sowie iPads ist ebenfalls enthalten, jedoch ersetzt dieser nicht das Testen auf realen Endgeräten. In diesem Fall ist das ein iPhone 5 mit iOS 8.1.

#### Model View Controller

Model View Controller (MVC, deutsch Modell Präsentation Steuerung) ist ein Entwurfsmuster zur Strukturierung der Softwareentwicklung, das Apple[[13]](#footnote-13) gerade für die Entwicklung von Apps vorschlägt und auch selber verwendet. Das Model kümmert sich um die Daten der App, in diesem Fall ist das die Anwendungsbeschreibung. Der View zeigt die Benutzerschnittstellen und die Daten aus dem Model an. Der Controller ist für die Zusammenarbeit der beiden verantwortlich und steht somit zwischen ihnen. Er nimmt Benutzereingaben entgegen und reagiert auf Modelländerungen. Vorteil dieses Musters ist die strikte Trennung der Daten bzw. des Modells von seiner Darstellung, sodass bspw. eine Darstellungsänderung keinen Einfluss auf das Modell hat.

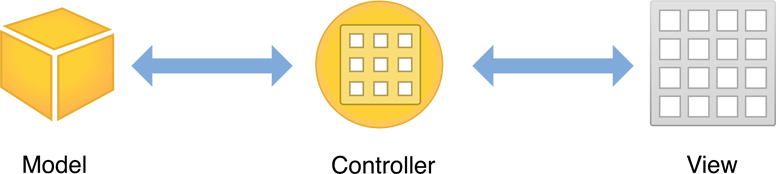


Abbildung 5:Apple Model View Controller

#### CoreData

Core Data ist ein von Apple entwickeltes Framework zum Verwalten von Objekt Graphen sowie deren Persistenz (Apple Inc. 2014). Das Datenmodell wird mithilfe von Entitäten und ihren Relationen zueinander beschrieben. Darüber hinaus können noch Abrufoperationen definiert werden. Einige Besonderheiten sind zum Beispiel:

* Änderungsverfolgung und deren Zurücknahme
* Relationen Verwaltung
  + Veränderungspropagation, Konsistenz der Objektrelationen
* Automatische Validierung von Objekteigenschaften
* Intelligentes Laden
  + Objekte werden erst bei Gebrauch geladen („lazy loading“)
* Hochentwickelte Abfragemöglichkeiten
  + Keine SQL Abfragen, Klasse NSPredicate übernimmt Abfragen

CoreData ist keine relationale Datenbank oder ein Datenbank Management System. Es nutzt beispielsweise SQLite um Daten persistieren zu können, ist aber selber keine Datenbank (Apple Inc 2014). In diesem Fall wurde CoreData gewählt, weil es wie iOS auch von Apple über Jahre entwickelt, getestet und für die Entwicklung von Apps optimiert wurde. Das zeigt sich bspw. an der guten Integration in XCode. Dort gibt es einen Editor für CoreData Datenmodelle sowie die Möglichkeit aus dem entworfenen Modell für jede Entität Klassen zu generieren. Ebenfalls kann bei Projektanlegung CoreData angewählt werden, wodurch das Framework automatisch in das Projekt eingebunden und Zugriffscode generiert wird. Der Hauptgrund für die Verwendung von CoreData ergibt sich jedoch aus dem folgenden Framework.

#### RESTKit

RESTKit[[14]](#footnote-14) ist ein Objective-C Framework zur einfachen und schnellen Nutzung von Webdiensten mit REST-Schnittstelle (RestKit 2014). Dies wird benötigt um die Anwendungsbeschreibung über eine solche Schnittstelle herunterzuladen, zu parsen und lokal zu speichern. Wichtige Funktionalitäten von RESTKit sind:

* Einfaches HTTP Abfrage / Antwort System
  + Framework beinhaltet HTTP Klienten mit hilfreichen Methoden zur Inspektion von MIME Typen sowie Status Codes
* Objekt Zuordnungssystem (engl. Object Mapping System)
  + Deklarative Zuordnung von geladenen Daten zu nativen Objekten
  + Automatische Typkonvertierung bspw. von String nach Datum (NSDate)
* CoreData Unterstützung
  + ermöglicht direktes persistieren von geladenen Objekten
  + setzt Objektrelationen automatisch
  + API zur einfacheren Konfiguration von CoreData und Abfrage von Daten

Neben JSON Unterstützt RESTKit auch andere MIME Typen, die in diesem Fall jedoch nicht von Bedeutung sind. Das Objekt Mapping nutzt ein Entwurfsmuster namens Key-Value Coding. Dies wird verwendet um indirekt Zugriff auf Objekteigenschaften / Attribute zu erhalten, indem Strings bzw. die Bezeichner der Attribute verwendet werden anstelle von Zugriffsmethoden oder dem direkten Zugriff über eine Instanz (Apple 2014). RESTKit macht sich dieses Muster zunutze um aus den geladenen Daten zuordnungsfähige Inhalte zu identifizieren und die lokalen Objekte und Relationen zu initialisieren (RESTKit 2014). Dies ist der Grund für die einfache deklarative Zuordnung von geladenen Daten zu nativen.

#### PixateFreestyle

PixateFreestyle[[15]](#footnote-15) ist ein Open Source Framework um nativen Benutzerschnittstellen ein individuelles Design zuzuordnen. Die unterstützten Plattformen sind iOS und Android. Das Design wird über Cascading Style Sheets[[16]](#footnote-16) (CSS) Dateien definiert. In iOS werden die Benutzerschnittstellen um das Attribut styleClass erweitert damit man ihnen in der CSS-Datei ein Design / Style zuordnen kann.

#### SDWebImage

SDWebImage ist eine Bibliothek die eine Kategorie enthält. Über Kategorien kann man in Objective-C andere Klassen erweitern. Das passiert in diesem Fall mit der Klasse UIImageView die zur Darstellung eines Bildes zuständig ist. Die Erweiterung bietet asynchrone Bild Downloads und eine automatische Cache Verwaltung für diese an. Das erleichtert die Verwendung externer Bilder und verlangsamt gleichzeitig nicht den Aufbau einer Ansicht, da die Bilder nebenläufig nachgeladen werden.

## Entwurf

Der Entwurf und die Umsetzung des Interpreters orientiert sich stark am zuvor beschriebenen Entwurfsmuster Model View Controller. Daher ergeben sich auch die folgenden drei Pakete: Model, View und Controller. Deren Inhalte stehen zu Beginn im Vordergrund, sodass die einzelnen Abhängigkeiten und Verbindungen innerhalb der Pakete zunächst vernachlässigt werden. Dies wird jedoch in den Abschnitten zu den jeweiligen Paketen nachgeholt. Auf das allgemeine Zusammenspiel der einzelnen Pakete wird ebenfalls nicht näher eingegangen, da dies aus dem Verständnis des Entwurfsmusters hervorgeht.

### Pakete Überblick

#### Model



Abbildung 6: Paket Model Überblick

Die Inhalte des Pakets *Model* ergeben sich aus dem Entwurf der Beschreibungssprache aus Kapitel 3. Die Sprachelemente Tab, Icons, Section und View werden jeweils mit einer eigenen Klasse repräsentiert. Im CoreData Daten Modell bezeichnet man diese als Entitäten woraus später die konkreten Klassen generiert werden. Aufgrund der geforderten Erweiterbarkeit erhalten die Sprachelemente Cell, Action und Button jeweils eine gemeinsame abstrakte Oberklasse, von der die konkreten Klassen erben.

#### View



Abbildung 7: Paket View Überblick

Das Paket View setzt sich insgesamt aus nur zwei Klassen zusammen. Die anderen Klassen werden hier nicht explizit erwähnt, da es sich um die Standarttypen der jeweiligen Controller handelt und diese vollkommen zur gewünschten Darstellung ausreichen. Bei einer individuell gewünschten Darstellung müssen diese ersetzt bzw. überschrieben werden. Die Klasse *UITextFieldCell* ist für die Darstellung der Modellklasse *CellTextField* zuständig und stellt neben einem Textfeld noch eine Schaltfläche zur Verfügung. Die zweite Klasse ist für die Darstellung einer *CellImage* zuständig. Abbildungen für diese Cell folgen im Abschnitt Umsetzung.

#### Controller



Abbildung 8: Paket Controller Überblick

Im Paket Controller befinden sich die 2 Hauptcontroller *TabBarController* und *ViewController*. Die Klasse *WebViewController* ist für die Action *WebViewWithURL* von Bedeutung und verwaltet dort den WebView. Der TabBarController ist der erste Controller der nach Anwendungsstart die Kontrolle übernimmt, weshalb er auch als Initialcontroller bezeichnet wird. Dieser benötigt daher auch die Adresse der Anwendungsbeschreibung. Er versucht zunächst eine lokal gespeicherte Anwendungsbeschreibung zu laden. Ist das nicht möglich, greift er über die eben genannte Adresse auf eine REST-Schnittstelle zu und lädt die Anwendungsbeschreibung herunter. Der ViewController ist der wichtigste Controller und für die Verwaltung eines einzelnen Views zuständig. Für jeden View wird demnach eine eigene Instanz dieses Controllers erzeugt. Ebenfalls befindet sich hier das Paket *Parser*. Es enthält zwei Klassen: RestParser und RestObjectMappings. Die erste bietet eine Schnittstelle zum RESTKit Framework an und ermöglicht somit das Herunterladen und Persistieren der Anwendungsbeschreibung. In der zweiten Klasse werden die Objektzuordnungen definiert. Diese wurden zur Verbesserung der Wartbarkeit des Codes hierhin ausgelagert.

In den nächsten drei Abschnitten wird die Umsetzung der einzelnen Klassen vorgestellt.

## Umsetzung

### Model

In diesem Abschnitt werden die Klassen des Pakets Model entworfen. Die Beschreibungssprachenelemente werden als Entitäten im CoreData Model Editor von XCode modelliert um daraus automatisch die Swift-Klassen zu generieren die sich persistieren lassen. Die generierten Klassen müssen dann noch um einige Methoden erweitert werden.

#### CoreData Daten Modell



Abbildung 9: CoreData Daten Modell Diagramm

Die Abbildung 9: CoreData Daten Modell Diagramm zeigt den Entwurf aus Kapitel 3 in der CoreData Model Editor üblichen Darstellung. Bei dieser werden die Attributs Typen nicht angezeigt und es fehlt auch die Information welche der Relationen optional oder obligatorisch sind. Diese Informationen sind jedoch im nächsten Diagramm vorhanden.



Abbildung 10: Interpreter Model

Das Interpreter Modell ist in Abbildung 10: Interpreter Model nun vollständig. Alle Klassen sind nun Unterklassen von *NSManagedObject*, sodass diese mithilfe von CoreData nun verwaltet und persistiert werden können (aus Gründen der Übersichtlichkeit nicht im Diagramm dargestellt). Ebenfalls sind zum CoreData Daten Modell noch Methoden hinzugekommen. Diese stellen eine Vermischung von Model und Controller dar, da sie Aufgaben des Controllers übernehmen. Das hat jedoch den Zweck die Controller zu vereinfachen und stellt eine bessere Repräsentation der Sprachelemente in Swift-Klassen dar. Das eigentliche Model liegt als CoreData Model vor und entspricht demnach dem MVC Muster. Die Darstellung bleibt weiterhin von Model und Controller getrennt. Elemente mit dem Titel *<< datatype >>* sind in dem verwendeten Diagrammeditor[[17]](#footnote-17) notwendig um eigene Datentypen im Diagramm verwenden zu können und haben sonst keinerlei Bedeutung.

Im Model hat die abstrakte Cell Klasse nun eine Methode *getCell()* welche von den Unterklassen überschrieben wird. Trifft ein Controller auf eine Cell, ruft er einfach dessen *getCell()* Methode auf und erhält ein Objekt vom Typ *UITableViewCell* die er in seinem *UITableView* anzeigen lassen kann. Genaueres dazu folgt im Abschnitt Controller. Die Methode instanziiert einen der Cell zugeordneten View. Zum Beispiel gibt *getCell()* in der Klasse *CellImage* eine *UIImageCell* zurück. *CellPrototype* verwendet als View die Standartklasse *UITableViewCell*.

Alle Unterklassen von Action überschreiben die *performAction()* Methode und bieten ähnlich zu den Cells dem Controller eine Schnittstelle an Actions auszuführen. Teilweise kommt auch eine Überladung der Methode vor, da einige Parameter zur Ausführung der Action gebraucht werden. Diese Parameter sind Objekte aus dem aufrufenden Controller, da diese Methode diesem zuarbeitet. Ein Beispiel wäre *ActionWebViewWithURL*, dass zur Darstellung des WebViews noch Zeiger auf den aktuellen *UINavigationController* und das *UIStoryboard* benötigt um diesen darauf anzuzeigen.

Die Aufgabe der *getButton()* Methode ähnelt den beiden zuvor genannten Methoden und wird deshalb nicht genauer beschrieben.

Das Model kann nun von den Controllern genutzt werden um alle Sprachelemente in Swift Objekte zu parsen, sowie deren Verhalten und Darstellung zu steuern.

### View

In diesem Abschnitt werden die unterschiedlichen Views zu den Cell Typen entworfen und ihre Darstellung an Beispielen gezeigt.



Abbildung 11: Interpreter Views

#### UITableViewCell

UITableViewCell ist die Oberklasse der beiden darauffolgenden View-Klassen. Sie ist im iOS SDK bereits enthalten und muss demnach nicht noch entworfen werden. Als Attribute zur Anzeige von Inhalten stehen drei zur Verfügung:

* *textLabel* – Zeigt ein Label (Formatierter Text) innerhalb der Cell an
* *detailTextLabel* – Zeigt ein Label unterhalb des *textLabel* an. Die Textgröße ist geringer im Vergleich zum *textLabel*.
* *imageView* – Zeigt auf der linken Seite ein Bild an. Wenn ebenfalls ein *textLabel* oder *detailTextLabel* angegeben wurde, werden diese weiter rechts angezeigt.



Abbildung 12: UITableView mit 2 UITableViewCells

Die Abbildung zeigt einen Ausschnitt aus einem UITableView mit unterschiedlichen 2 UITableViewCells. Die erste Cell hat ein *textLabel* und ein *detailTextLabel*. Die zweite Cell hat ein *imageView* und ein *textLabel*.

#### UIImageCell

Die Klasse *UIImageCell* erhält für die individuelle Darstellung eines Bildes das Attribut *customImageView*, obwohl es bereits einen ImageView von der Oberklasse geerbt hat. Der Grund ist die bessere Konfigurierbarkeit eines eigenen ImageViews. Die Konfiguration erfolgt über den GUI Editor in XCode und nicht direkt per Code. Sie wird so konfiguriert, dass das Bild zentriert in der Zelle liegt und eine bestimmte Größe hat. Das Bild wird mithilfe von SDWebImage zur Laufzeit asynchron geladen falls es noch nicht im Cache ist. Die folgende Abbildung zeigt ein Beispiel für eine UIImageCell die das Bild einer Person anzeigt.



Abbildung 13: Beispiel UIImageCell

#### UITextFieldCell

Diese Klasse enthält für die Darstellung eines Textfeldes sowie einer Schaltfläche zwei Attribute: *textField* und *button*. Die Konfiguration erfolgt hier ebenfalls über den GUI Editor in Xcode. Die folgende Abbildung ist ein Screenshot aus dem iOS Simulator und zeigt eine solche Cell. Im Textfeld hat der Nutzen den neuen Namen „Andreas“ eingeben der nun über die Schaltfläche „Senden“ abgeschickt werden kann.



Abbildung 14: Beispiel UITextFieldCell

### Controller



Abbildung 15: Interpreter Controller

#### TabBarController

Der *TabBarController* ist der erste Controller beim Start der App. In der *loadView*() Methode wird zunächst wie in allen weiteren Controller mit *activityIndicator*, dieser initialisiert. Anschließend wird die *loadTabsFromDB*() Methode aufgerufen, sodass der Interpreter zuerst versucht vorhandene Tabs aus der lokalen Datenbank in das Attribut *tabs* zu laden. Sollten keine Tabs in der lokalen Datenbank vorhanden sein, wird die loadTabsFromWeb() Methode aufgerufen. Diese holt sich eine Instanz des *RESTParser* und übergibt ihm die festcodierte Appinitialisierungs-URL. Die Klasse RKObjectManager aus dem RESTKit übernimmt nun das herunterladen und parsen der Anwendungsbeschreibung und speichert diese gleichzeitig in der lokalen Datenbank ab. Nachdem diese nun in der Datenbank vorhanden sind, wird erneut die *loadTabsFromDB*() Methode aufgerufen. Bei vorhandenen Tabs endet diese Methode mit dem Aufruf von *tabsDidLoad*(). Diese erzeugt für jeden Tab einen eigenen *ViewController* und lässt den ersten anzeigen. Ebenfalls wird beim erzeugten Controller das Attribut autoLoadView gesetzt, damit es bei der Anzeige entscheiden kann ob der View automatisch oder erst bei Benutzerinteraktion aus dem Web geladen werden soll. Diese Information erhält der TabBarController aus dem geladenen Tab Objekt, das ja das Attribut autoLoadView enthält.

#### ViewController

Der *ViewController* verhält sich ähnlich zum *TabBarController*. Der *viewPath* im TabBarController ist identisch zum *identifier* im *ViewController*. In diesem Fall stehen dort eine URL zur REST-Schnittstelle. Der *RESTParser* nutzt diese Adresse um von dort die Beschreibung für den jeweiligen View herunterzuladen. Doch zunächst versucht der *ViewController* seinen *identifier* in der Datenbank zu finden. Ist dieser nicht vorhanden, entscheidet der Wert von autoDidLoad ob automatisch nachgeladen wird oder erst eine Benutzerinteraktion nötig ist. Zur Interaktion muss der Benutzer den View nach unten ziehen. Dies wurde der MuC2014 nachempfunden. Das Nachladen übernimmt der *RESTParser* und anschließend wird erneut versucht den View aus der Datenbank zu laden. Ist dies erfolgreich folgt der Aufruf von *viewRESTDidLoad*(). Dies lässt die Tabelle neu laden und stellt damit den beschriebenen View da. Im Diagramm sind Methoden mit dem Präfix *tableView* sowie die Methode *numberOfSectionsInTableView()* aufgeführt. Das sind iOS SDK spezifische Delegationsmustermethoden und werden beim Laden der Tabelle aufgerufen. Wie den Bezeichnern zu entnehmen ist, definieren sie:

* Anzahl der Sections sowie deren Titel
* Anzahl der Cells in den Sections
* Cell Typen (ruft die *getCell()* Methode der jeweiligen Cell auf)
* Actions bei Berührung einer Cell

Alle benötigten Informationen und Objekte werden dem geladenen View Objekt entnommen.

#### Parser

Die Klasse *RESTParser* ist die Schnittstelle zum RESTKit und kümmert sich dementsprechend um die Initialisierung mit einer Basis URL sowie um die Objektzuordnungen (Mapping). Die statischen Methoden der *RESTObjectMappings* Klasse helfen ihr dabei nach dem Entwurfsmuster Delegation. Dies dient zur Übersicht und Wartbarkeit des Codes da die Objektzuordnungen sehr viele Zeilen in Anspruch nehmen. An dieser Stelle wäre es sinnvoll einen Generator für den Objektzuordnungscode zu entwickeln.

#### WebViewController

Der WebViewController ist für die Anzeige eines WebViews mit einer entsprechenden URL zuständig. Diese erhält er bei Initialisierung als Parameter. Der Controller wird ausschließlich von der Action *ActionWebViewWithURL* verwendet.

## Interpreter für andere Plattformen

Nachdem der Interpreter für iOS nun fertig entworfen und umgesetzt wurde, stellt sich die Frage ob dieser Entwurf auch auf anderen Plattformen umsetzbar ist. Das Model wurde ohne iOS spezifische Anteile entworfen und ist demnach nicht an diese Plattform gebunden. Es wurden lediglich nach dem MVC Entwurfsmuster eine klare Trennung zwischen Model und View gemacht. Diese Trennung verwenden auch andere Plattformen. Android verwendet nicht direkt MVC, dessen *Activity* Konzept (Google Inc. 2015b) ähnelt aber Apples *ViewController* (Apple Inc. 2015b). Für Windows Phone gibt es eine Abwandlung von MVC: ModelView-ViewModel (MVVM) Entwurfsmuster (Microsoft Corporation 2014), worauf hier nicht näher eingegangen wird. Für alle Plattformen kann das entworfene Model demnach umgesetzt werden. Weitere Plattformen wie Blackberry wurden hierbei nicht untersucht. In Hinblick auf die verwendeten Frameworks lassen sich für andere Plattformen ähnliche Lösungen finden oder müssen selbst umgesetzt werden:

Tabelle : Framework Alternativen für andere Plattformen

|  |  |  |
| --- | --- | --- |
| iOS | Android | Windows Phone |
| RESTKit | Restlet[[18]](#footnote-18) | RestSharp[[19]](#footnote-19) |
| PixateFreestyle | PixateFreestyle | Kein vergleichbares Framework gefunden |
| SDWebImage | Android-Universal-Image-Loader[[20]](#footnote-20) | Kein vergleichbares Framework gefunden |

# Beispiel iOSTemplateLanguage Applikation

Um die entwickelte Beschreibungssprache einer ersten Evaluation zu unterziehen, bedarf es einer Beispiel App. Diese erhält hier den Namen iOSTemplateLanguage App. Der Funktionsumfang beschränkt sich auf Teile der MuC2014.

## Anforderungen und Voraussetzungen

Der Interpreter wird mit einer festkodierten Initialisierung-URL sowie integrierten Icons und einem Stylesheet ausgeliefert. Der Grund dafür ist, dass der Interpreter noch nicht in der Lage ist über die Sprache definierte Icons und Stylesheets selbstständig zu beziehen. Zur URL kommen noch einige Parameter hinzu die notwendig für den Zugriff auf die CommunityMashup REST API sind. Ebenfalls ist der Zugriff auf die Template-Dateien notwendig um diese zu entwickeln. Die gesamte App Entwicklung findet an dieser Stelle statt. Die lokale Datenbank ist zu Beginn leer und enthält somit keinerlei Beschreibungsinformationen. Zum Test der einzelnen Sprachelemente ist es sinnvoll alle Cell, Action und Button Typen einzusetzen. Für den später folgenden Performanz Test sollten ähnlich große Datensätze wie bei der MuC2014 dargestellt werden.

## CommunityMashup

Das CommunityMashup wurde in der Einleitung bereits als Datenquelle genannt. Dessen REST API soll den Zugriff auf die Anwendungsbeschreibung ermöglichen. Auf das zugrundeliegende Datenmodell des Mashup sowie die detaillierte Nutzung der REST API wird in dieser Arbeit nicht genau eingegangen. Einen Überblick dazu liefert Hoferick(2013). Dennoch soll zum Verständnis an einigen Beispielaufrufen die Nutzung der API gezeigt werden. Die folgenden URL sind verkürzt und zeigen mögliche Aufrufe.

Eine Abfrage aller im Mashup befindlichen Personen sieht folgendermaßen aus: *mashup/getPersons*

Die Person mit der Identifikation a\_4403 erhält man bspw. mit diesem Aufruf: */mashup/getPersonWithIdent?ident=a\_4403*

Für eine Template gesteuerte Ausgabe wird noch der Parameter *tpl* benötigt, der den Namen des Templates angibt. In diesem Fall soll das Template *ios* verwendet werden: */mashup/getPersonWithIdent?ident=a\_4403&tpl=ios*

Weitere mögliche Aufrufe sind der Mashup Dokumentation[[21]](#footnote-21) zu entnehmen.

#### Templates

Die Generierung der Anwendungsbeschreibung ist Aufgabe der Templates. Der Interpreter ruft die gewünschten REST Funktionen mit dem Template als Parameter auf und erhält die Beschreibung im JSON Format. Für jedes Objekt (bspw. Liste von Personen, Person, Organisation) enthält das Template eine eigene Datei. Zu Beachten ist, dass für unterschiedliche Darstellungen desselben Objekts auch verschiedene Beschreibungen und demnach verschiedene Templates notwendig sind. Das CommunityMashup verwendet FreeMarker [[22]](#footnote-22)als Template Engine (Rohde 2013). Das ist eine freie Java Klassenbibliothek und wird dazu verwendet Text zu generieren. Sie ist nur für Anzeige zuständig und generiert keine Daten. Platzhalter in der Template Datei werden durch FreeMarker ersetzt. Bei der Umsetzung der App wird dies dazu benutzt Daten aus dem Mashup in die Anwendungsbeschreibung zu integrieren. Im nächsten Abschnitt soll die iOSTemplateLanguage App nun entworfen und umgesetzt werden.

## Entwurf

Der Entwurf der iOSTemplateLanguage App orientiert sich am Aufbau und einigen Funktionalitäten der MuC2014. Im Zentrum stehen Personen, Organisationen und Inhalte. Da dies Teile des CommunityMashup Datenmodell Kerns sind stehen für alle 3 auch passende Funktionen in der REST API zur Verfügung (Lachenmaier et al. 2012).

Die App ist in 3 Bereiche gegliedert: Personen, Organisationen und Programm. Der Bereich Programm zeigt bestimmte Inhalte aus dem Mashup an. Jeder Bereich wird erst nach einer Benutzerinteraktion geladen um den Anwendungsstart nicht zu verzögern. Im Bereich Personen wird eine Liste aller teilnehmenden Personen mit ihrem Profilbild und vollständigem Namen angezeigt. Zu jeder Person ist weiterhin eine detailliertere Ansicht vorhanden, die direkt aus der Liste erreichbar ist. Zu der detaillierteren Ansicht gehören, falls vorhanden:

* Vergrößertes Profilbild
* Name
* Twitter Account
* Webseite

In dieser Ansicht kann man die Person als Kontakt im Telefonbuch ablegen und es besteht eine Verbindung zur Twitter App um Profilseite der Person automatisch aufzurufen. Weiterhin kann man von hier aus direkt die angegebene Webseite aufrufen.

Im Bereich Organisationen werden alle vorhandenen Organisationen mit ihrem Namen aufgelistet. Jede Organisation kann ebenfalls detaillierter betrachtet werden:

* Geschäftsführer
* Unternehmenswebseite
* Organisationsmitglieder die an der Konferenz

Von hier kann man ebenfalls wieder zur detaillierteren Ansicht der einzelnen Personen gelangen.

Im Bereich Programm werden alle Programmpunkte mit ihrem Titel aufgelistet. Auch hier gibt es eine detailliertere Ansicht zu den Inhalten die folgendes umfasst (für Programmpunkte wie die Mittagspause machen die folgenden Informationen natürlich keinen Sinn):

* Sprache
* Autor / Referent
* Mitwirkende
* Kommentarbereich

Da der Autor eine Person ist, gibt es wieder die Möglichkeit diesen genauer anzuschauen. Der Kommentarbereich ermöglicht es den aktuellen Programmpunkt zu kommentieren. Vorhandene Kommentare werden hier jedoch noch nicht angezeigt.

## Umsetzung

Die Umsetzung der App findet in den Template Dateien auf dem Mashup Server statt. Dafür wurde extra das Template *ios* angelegt. Begonnen wird mit der Beschreibung der vorhandenen Tabs, ihrer URL zum Hauptview (*viewPath* Attribut) und Icons. Da die App nach Entwurf in 3 Bereiche unterteilt werden soll, empfiehlt es sich hier 3 Tabs zu erstellen. Für diese Anfangsbeschreibung wird die Template Datei *tpl\_DataSet* angelegt. Das Objekt *DataSet* beinhaltet den gesamten Mashup Datensatz und eignet sich für die Startbeschreibung der App. Demnach wird folgende Appinitialisierung-URL fest in den Interpreter kodiert: [*http://muc2014.communitymashup.net/j3/mashup/*](http://muc2014.communitymashup.net/j3/mashup/). Weiterhin sind folgende Parameter notwendig:

* tpl = ios – Angabe des zu verwendenden Standard Templates
* appKey = abc – Zugriffscode auf das Mashup (sehr unsicher!)
* wrap = false – Schaltet ein Template ab, das die Ausgabe um einige Informationen erweitert. Das ist hier nicht notwendig.

Die Datei *tpl\_DataSet* beschreibt 3 Tabs mit den Titeln: Personen, Organisationen und Programm. Die *viewPath* Attribute der einzelnen Tabs beinhalten den Pfad zur REST Funktion ausgehend von der Appinitialisierung-URL die als Basis URL fungiert. Der *viewPath* für den Tab mit der Liste an Personen ist: *getPersons?appKey=%appKey&wrap=%wrap*

Die Parameterwerte mit Präfix ”%” bedeuten für den Interpreter, dass er den Wert aus den Initialisierungsparametern nehmen soll. Sinn macht das bspw. beim *appKey* damit dieser hier nicht im Klartext auftaucht. Die Icons sind, wie in den Anforderungen und Voraussetzungen beschrieben, bereits im Interpreter integriert und müssen nur noch per String angegeben werden.

Die drei Bereiche stellen jeweils drei unterschiedliche Objekte da. Das bedeutet, dass für Personen, Organisationen und Programm jeweils drei verschiedene Template Dateien benötigt werden:

* tpl\_Person\_list – Stellt die Liste an Personen dar
* tpl\_Person\_list\_item – Stellt ein einzelnes Listenelement dar
* tpl\_Person – Stellt eine Person detailliert dar

Für Organisationen und Programm existieren äquivalente Template Dateien.

Im Folgenden wird nur die Beschreibung für den Bereich Personen vorgestellt um den Rahmen dieser Arbeit nicht zu sprengen. Begonnen wird mit der Template Datei *tpl\_Person\_list* die den Hauptview des Tabs Personen generiert. Hier wird erstmals die eigene Sprache von FreeMarker verwendet um die erhaltene Liste an Personen zu durchlaufen. Dabei wird zu jeder Person ihre Listendarstellung abgerufen. Das Attribut *autoLoadData* wird hier auf *false* gesetzt um die Personenliste erst nach Benutzerinteraktion darzustellen. Die folgende Abbildung zeigt die App mit 3 Tabs sowie den Hauptview des Personen Tabs.



Abbildung : Bereich Personen in der Beispiel App

Im unteren Bereich sieht man die Tableiste mit den 3 Tabs. Die Personenliste wird erst geladen, nachdem der Nutzer das Bild von oben nach unten zieht.

Die Beschreibung der einzelnen Listenelemente folgt in der *tpl\_Person\_list\_item* Datei. Der Cell Typ ist hier CellPrototype. Dem *title* Attribut wird der Name der Person zugewiesen und als *image* wird das Profilbild der Person verwendet, falls dies vorhanden ist. FreeMarker bietet durch bedingte Anweisungen die Möglichkeit ein Ersatzbild zu verwenden. Die Cell hat auch eine Action vom Typ *ActionSegueIntoView*, die als *target* den Pfad zur detaillierten Darstellung der Person hat. Dieser Pfad ist natürlich gleichzeitig *identifier* eines anderen Views der in der Datei tpl\_Person beschrieben wird.

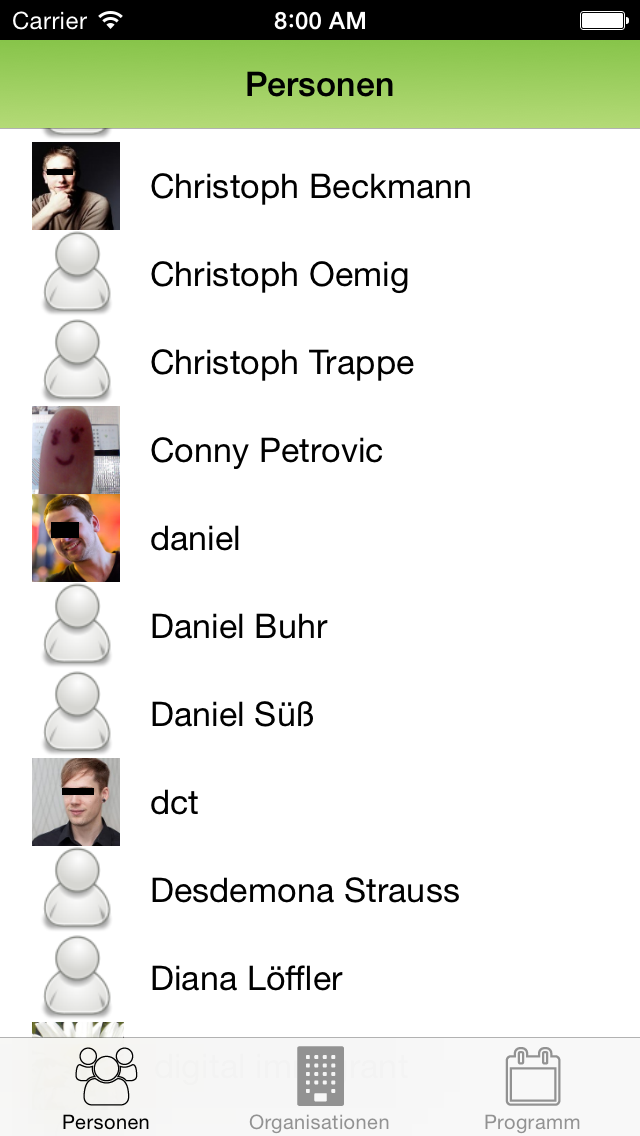


Abbildung : Bereich Personen mit geladener Liste

Zu sehen ist eine Liste von Personen mit ihrem Profilbild und Namen. Personen ohne Profilbild erhalten ein Standardbild. Im nächsten Bild ist der detailliertere View einer einzelnen Person zu sehen.



Abbildung : Detaillierte Ansicht einer Person in der Beispiel App

Der View besteht aus mehreren Section die jeweils eine Cell enthalten. Nur die erste Cell ist vom Typ *CellImage* und enthält das Profilbild einer Person. Alle anderen Cell sind vom Typ *CellPrototype*. Die zweite hält im *title* Attribut den Namen der Person und hat die Action vom Typ *ActionAddToContacts*. Sie fügt beim Drücken auf die Cell Namen, Telefonnummer und Webseite dem Telefonbuch hinzu, falls der Eintrag noch nicht vorhanden ist. Als nächstes folgt eine Cell mit Action Typ ActionOpenAppWithURL. Dabei wurde folgende URL verwendet: *twitter://user?screen\_name=.* FreeMarker setzt am Ende den passenden Twitternamen ein und iOS öffnet die App insofern sie auf dem Gerät vorhanden ist. Die letzte Cell hat die Action vom Typ ActionOpenWebWithURL und bekommt als *url* Attribut die Webseite der Person.

Der hier nicht näher beschriebenen Bereich Programm verwendet, um die erwähnten Anforderungen und Voraussetzungen einzuhalten, noch den Cell Typ CellTextField sowie die Action ActionPostTextWithURL um dem Benutzer das Kommentieren eines Inhalts zu ermöglichen.

## Fazit

Die App wurde nach den Anforderungen und den genannten Voraussetzungen umgesetzt und kann bereits verwendet werden. Der Entwicklungsprozess der App fand ausschließlich in den Template Dateien auf dem CommunityMashup Server statt. Natürlich wurden dabei auch einige Fehler bzw. fehlende Funktionen des Interpreters entdeckt und behoben. Das sind insbesondere Design schwächen, bei denen der Interpreter nicht die gewünschte Darstellung der Benutzerschnittstellen lieferte. Ebenfalls wurde parallel das Style Sheet (CSS) immer wieder angepasst und neue Designs ausprobiert. Während der Entwicklung wurden auch zu Testzwecken andere Tabs beschrieben und die Änderbarkeit der App überprüft. Dies funktionierte bisher zufriedenstellend, bedarf aber einer wissenschaftlichen Evaluation, die nicht Bestandteil dieser Arbeit ist. Im Hinblick auf die Geschwindigkeit gibt es erste Ergebnisse im nächsten Abschnitt. Die Vorzüge einer mächtigen Entwicklungsumgebung wie XCode fehlen hier natürlich ebenfalls. Autovervollständigung sowie eine direkte Fehleranzeige existieren noch nicht. Der Code wurde in dem Texteditor TextMate[[23]](#footnote-23) geschrieben der zumindestens eine Syntaxhervorhebung für JSON Code anbietet. Das erleichterte die Entwicklung. Dennoch wird fehlerhafter Code erst zur Laufzeit erkannt und erfordert somit ständiges Hochladen und Testen was den Entwicklungsprozess verlangsamt. Die Beschreibungssprache und der Interpreter sind an diesem Punkt dennoch fähig als Endbenutzer Anwendung für das CommunityMashup zu fungieren.

### Verbesserte Performanz

In diesem Abschnitt wird auf die verbesserte Performanz insbesondere durch „caching“ und „lazy loading“ eingegangen. Als Vergleichswerte werden Messungen vom CommunityMashup iOS Framework verwendet, das auch in der MuC2014 zum Einsatz kommt (Hoferick 2013). Die Messungen des iOS Interpreters wurden in dem iOS-Simulator 8.1 auf einem MacBook Pro von 2011 und auf einem iPhone 5 mit iOS 8.1 durchgeführt. Die Datenladezeit ist die benötigte Zeit für das Herunterladen und Persistieren der Anwendungsbeschreibung in Sekunden. Die Datenladezeit Tabs ist die Zeit in der die Beschreibung für alle Tabs heruntergeladen und persistiert wird (generierte Beschreibungsinformationen aus *tpl\_DataSet*). Die Datenladezeit View bezeichnet denselben Vorgang für einen View. Die folgenden Werte ergeben sich beim Laden bzw. Darstellen der Personenliste im Bereich Personen mit 203 Einträgen (Datengröße 72 Kb).

Tabelle : iOSTemplateLanguage App Performanztest Ergebnisse

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Datenladezeit | Plattform | WLAN | 3G | EDGE |
| Tabs | iOS-Simulator 8.1 | 0,2 | 0,5 | 1,8 |
| iPhone 5 64GB | 0,3 | 2,0 | 6,2 |
| View | iOS-Simulator 8.1 | 0,5 | 1,5 | 4,3 |
| iPhone 5 64GB | 1 | 1,2 | 1-60 oder Zeitüberschreitung |

Im WLAN sind die Ladezeiten der beiden Plattformen sehr kurz und konstant. Das liegt auch daran, dass Simulator und Smartphone im selben WLAN gemessen wurden. Die 3G und vor allem EDGE Verbindung war am Messort nicht sehr optimal. Zum Laden des View wurden im EDGE-Netz sehr unterschiedliche Werte gemessen. Teilweise kam es auch zu einer Zeitüberschreitung der Anforderung. Eine stabilere Verbindung sollte Messwerte ähnlich zum Simulator liefern. Schließlich ist die Beschreibung für die Personenliste nur 72Kb groß.

Im Vergleich zum iOS Framework sind dies Topleistungen. Dieses lädt ja alle Inhalte zu Beginn herunter und parst diese sofort, wo hingegen der Interpreter nur den gewünschten Teil lädt und speichert. Der Interpreter benötigt im WLAN etwa eine Sekunde um die Tabs und den ersten View zu laden. Das Framework braucht hier im Simulator 7,5 Sekunden und auf einem realen Endgerät 80 Sekunden im Schnitt. Auf die 3G und EDGE Werte wird hier nicht genauer eingegangen, da diese sich ähnlich verhalten.

Aufgrund dieser kurzen Ladezeiten wurde das Ziel erreicht. Die Anwendung wird deutlich schneller initialisiert und kann auch bei schlechteren Verbindungen gut genutzt werden. Weiterhin zeigen die Messwerte, dass Änderungen bzw. Updates zur Laufzeit kein Problem darstellen, da diese in kurzer Zeit geladen und dargestellt werden können.

# Zusammenfassung und Ausblick

In dieser Bachelorarbeit ging es um die Entwicklung einer Beschreibungssprache für mobile Anwendungen. Ziel dieser Sprache war es eine Plattform- und Anwendungsszenario unabhängige mobile Anwendungsentwicklung zu ermöglichen. Dazu wurden in Kapitel 2 aktuelle Technologien bzw. Frameworks vorgestellt. Diese hatten ihre Vor- und Nachteile, sodass letztendlich Gründe für ein eigenes Framework, insbesondere einer eigene Sprache festgestellt wurden. Daraufhin wurde eine eigene Beschreibungssprache entwickelt, die möglichst plattformübergreifende Benutzerschnittstellen verwendet, auf einer bereits vorhandenen Syntax (JSON) aufbaut und dazu noch sehr einfach ist. Inwiefern diese Sprache tatsächlich auch für andere Plattformen geeignet ist, bedarf es weiterer Forschung.

Ab Kapitel 4 beginnt jedoch eine erste Evaluation der Sprache für Apples Betriebssystem iOS , indem ein Interpreter als native App entwickelt wurde. Dabei orientiert sich die Entwicklung stark an grundlegenden Prinzipien und Entwurfsmustern der App Entwicklung unter iOS. Dabei zeigt der Abschnitt 4.4 Interpreter für andere Plattformen, dass der Entwurf auch auf anderen Plattformen möglich ist und dort auch ähnliche Frameworks für die Umsetzung vorhanden sind. Der fertige Interpreter wurde daraufhin in Kapitel 5 in einem ersten Anwendungsentwicklungsprozess ausprobiert. Die iOSTemplateLanguage App wurde nach dem Vorbild der MuC2014 App entwickelt. Diese bringt nur einen Teil der Funktionalität ihres Vorbildes mit sich. Für mehr Funktionalität ist nicht nur eine Weiterentwicklung der App Beschreibung, sondern auch der Sprache und des Interpreters notwendig. Schwerpunkte dabei sind Darstellungen(Design) und angebotene Funktionalitäten (Actions).

Die Anfangs definierten Ziele wurden jedoch erreicht. Die Beschreibungssprache definiert Aussehen, Inhalte und Funktionalitäten der App. Zur Entwicklung und vor allem für spätere Änderungen muss kein nativer Code geschrieben werden. Im Punkt Geschwindigkeit setzt die App im Vergleich zu Muc2014 neue Maßstäbe. Durch die optimierte Verwaltung der Beschreibungsinformationen ist eine App schneller einsatzbereit und kann effizienter aktualisiert werden. Der Aktualisierungsprozess ist jedoch noch nicht ausgereift und müsste weiter optimiert und vor allem benutzerfreundlicher gemacht werden. Ein Lösung wäre, veraltete Views für den Benutzer kenntlich zu machen, sodass dieser dann Interagieren kann und die neue Version erhält.

Letztendlich zeigt diese Arbeit Entwürfe und Möglichkeiten auf mobile Anwendungen Plattform- und Anwendungsszenario unabhängig zu entwickeln. Anhand dieser Ergebnisse können weitere Apps oder Sprachen entwickelt werden.

# 

# Literaturverzeichnis

Apple Inc. (2014). iOS Human Interface Guidelines (p. 228).

Apple Inc. (2015). View Controller Programming Guide for iOS. Retrieved January 10, 2015, from https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/Introduction/Introduction.html

Google Inc. (2015). Android API Guides User Interface. Retrieved January 09, 2015, from <http://developer.android.com/guide/topics/ui/index.html>

Google Inc. (2015b). Android API Guides Activities. Retrieved January 10, 2015, from <http://developer.android.com/guide/components/activities.html>

Heitk, H., Hanschke, S., & Majchrzak, T. A. (2013). Evaluating Cross-Platform Development Approaches for Mobile Applications. In J. Cordeiro & K.-H. Krempels (Eds.), Web Information Systems and Technologies (pp. 120–138). Springer Berlin Heidelberg.

Hoferick, S. (2013). iOS Api Performance. In wiki.informatik.unibw-muenchen.de. Retrieved from http://wiki.informatik.unibw-muenchen.de/confluence/display/unui/iOS+Api+Performance

Hoferick, S. (2014). Konzeption einer kontextsensitiven Tablet-Anwendung zur Unterstützung von wissenschaftlichen Konferenzen. (P. Lachenmaier, Ed.). Universität der Bundeswehr München.

Knab, S. (2014). 2014-I : Das Leben in der digitalen Welt Mobile Effects 2014-I : Inhalt (pp. 1–63). Retrieved from http://www.tomorrow-focus-media.de/marktforschung/digitalmarkt/info/mobile-effects-2014-i/

Krösmann, C. (2014). Mobilgeräte verändern den Markt für Unterhaltungselektronik. Retrieved December 09, 2014, from http://www.bitkom.org/de/presse/8477\_80234.aspx

Lachenmaier, P., Ott, F., & Koch, M. (2013). Model-driven development of a person-centric mashup for social software. Social Network Analysis and Mining, 3(2), 193–207. doi:10.1007/s13278-012-0064-x

Microsoft Corporation. (2014b). Implementing the Model-View-ViewModel pattern for Windows Phone 8. Retrieved January 10, 2015, from http://msdn.microsoft.com/en-us/library/windows/apps/gg521153(v=vs.105).aspx

Microsoft Corporation. (2015). User Interface for Windows Phone 8. Retrieved January 09, 2015, from <http://msdn.microsoft.com/en-us/library/windows/apps/ff967556(v=vs.105).aspx>

Nurseitov, N., Paulson, M., Reynolds, R., & Izurieta, C. (2009). Comparison of JSON and XML Data Interchange Formats : A Case Study, 157–162.

RESTKit. (2014). RestKit Object Mapping. Retrieved January 02, 2015, from https://github.com/RestKit/RestKit/wiki/Object-mapping

Rohde, C. (2013). Entwurf und Umsetzung eines generischen Webframeworks zur Präsentation von Mashup-Daten. (P. Lachenmaier, Ed.). Universität der Bundeswehr München.

Svanidze, D. (2014). NATIVE APPS VS: WEB APPS VS: HYBRIDE APPS. Retrieved December 09, 2014, from http://app3null.com/native-hybride-web-apps/#Unterschied

1. http://phonegap.com [↑](#footnote-ref-1)
2. http://www.appcelerator.com/titanium/ [↑](#footnote-ref-2)
3. http://phonegap.com/about/feature/ [↑](#footnote-ref-3)
4. http://www.json.org [↑](#footnote-ref-4)
5. https://www.whatsapp.com/?l=de [↑](#footnote-ref-5)
6. https://de-de.facebook.com/mobile/ [↑](#footnote-ref-6)
7. https://www.apple.com/de/ios/what-is/ [↑](#footnote-ref-7)
8. Apple iOS-Simulator 8.1 [↑](#footnote-ref-8)
9. http://core0.staticworld.net/images/article/2013/02/system-settings-100025063-orig.jpg [↑](#footnote-ref-9)
10. http://www.onlineandroidtips.com/images/o18.jpg [↑](#footnote-ref-10)
11. https://developer.apple.com/swift/ [↑](#footnote-ref-11)
12. https://developer.apple.com/swift/ [↑](#footnote-ref-12)
13. https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/DesignPatterns.html#//apple\_ref/doc/uid/TP40011343-CH5-SW1 [↑](#footnote-ref-13)
14. http://restkit.org [↑](#footnote-ref-14)
15. http://www.pixate.com [↑](#footnote-ref-15)
16. http://www.w3.org/Style/CSS/ [↑](#footnote-ref-16)
17. https://www.genmymodel.com [↑](#footnote-ref-17)
18. http://restlet.com [↑](#footnote-ref-18)
19. http://restsharp.org [↑](#footnote-ref-19)
20. https://github.com/nostra13/Android-Universal-Image-Loader [↑](#footnote-ref-20)
21. http://muc2014.communitymashup.net/doc/index.html [↑](#footnote-ref-21)
22. http://freemarker.org [↑](#footnote-ref-22)
23. http://macromates.com [↑](#footnote-ref-23)