



# Bachelorarbeit

zur Erlangung des akademischen Grades "Bachelor of Science" im Fach Informatik

## Implementierung eines generischen Clients für iOS zur Analyse von REST-Services

Mario Stief

geboren am 23.05.1981 in Trier

eingereicht im Dezember 2012

Betreuer:

Prof. Dr. Peter Sturm

Lehrstuhl für Systemsoftware und Verteilte Systeme

Zweitgutachter:

Prof. Dr. Stephan Diehl

Lehrstuhl für Softwaretechnik

## Erklärung zur Bachelorarbeit

Hiermit erkläre ich, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

---

Ort, Datum

---

Unterschrift

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
<b>2</b>	<b>Die Werkzeugkiste</b>	<b>7</b>
2.1	Basissprache: Objective-C . . . . .	8
2.1.1	Protokolle: Data Sources und Delegates . . . . .	10
2.1.2	Properties . . . . .	11
2.2	API: Cocoa Touch . . . . .	12
2.3	IDE: Xcode . . . . .	13
2.4	Interface Builder: Storyboards . . . . .	14
2.5	Entwicklerlizenz: Testen auf echter Hardware . . . . .	17
2.6	Distributionsalternative: Der Cydia Store . . . . .	17
<b>3</b>	<b>Schnittstelle mit der REST-Welt</b>	<b>20</b>
3.1	RESTful Webservices . . . . .	21
3.1.1	Begriffseinführung . . . . .	21
3.1.2	Richtlinien . . . . .	21
3.1.3	Beispiel . . . . .	25
3.2	XML . . . . .	26
3.3	JSON . . . . .	28
<b>4</b>	<b>Implementierung</b>	<b>30</b>
4.1	Ein erster Überblick . . . . .	31
4.2	Hauptklasse: ViewController . . . . .	31
4.2.1	Senden eines Requests . . . . .	35
4.2.2	Empfangen einer Response . . . . .	38
4.3	Hilfsklassen . . . . .	45
4.3.1	ResourceTableViewController . . . . .	45
4.3.2	HeaderKeysViewController . . . . .	46
4.3.3	XMLParser . . . . .	46
4.3.4	LogOutputViewController . . . . .	47
4.3.5	HistoryElement . . . . .	48

4.3.6 ... und die AppDelegate? . . . . .	49
<b>5 Schlusswort und Ausblick</b>	<b>50</b>
<b>Literaturverzeichnis</b>	<b>54</b>

# Kapitel 1

## Einleitung

Im Programmieralltag eines Informatikers an der Universität Trier beschäftigt man sich hauptsächlich mit der plattformunabhängigen objektorientierten Programmiersprache Java. Sie eignet sich aufgrund ihrer Portabilität hervorragend als Einsteigersprache, denn die für das Ausführen benötigte Laufzeitumgebung kann man sich für nahezu jedes Betriebssystem herunterladen und installieren. Eine Ausnahme bildet das mobile Betriebssystem iOS, für welches Apple die Entwicklung einer Java Runtime nicht gestattet. Maschinennähere Erfahrungen sammelt man in C++, welches auf dem prozeduralen ANSI-C basiert, jedoch zahlreiche Erweiterungen in Hinblick auf die objektorientierte Programmierung bietet, seien es Möglichkeiten zur Datenkapselung, Vererbung oder Polymorphie, um nur einige zu nennen. Dazu wurden Sprachkonstrukte von C abgeändert, so dass manche C-Programme abgepasst werden müssen, um sich als C++-Programm übersetzen zu lassen. Einen etwas anderen Ansatz geht Apple mit der Programmiersprache Objective-C, welche ursprünglich als Basis für das Betriebssystem NextStep verwendet wurde. Mit der Übernahme zu Apple wurde die aus NextStep hervorgehende OpenStep-API ebenfalls Grundlage für OS X, womit sich auch die Systemsprache von Pascal auf Objective-C änderte. Anders als C++ greift Objective-C für die objektorientierte Erweiterung auf die Sprache Smalltalk zurück, in der mit Nachrichtenkommunikation gearbeitet wird und in der es nur drei eingebaute ausführbare Konstrukte gibt: Eine Botschaft wird an ein Objekt gesendet, einer Variable wird ein Objekt zugewiesen oder ein Objekt wird als Rückgabewert einer Methode zurückgeliefert. Die Smalltalk-Erweiterung ist von der gewöhnlichen C-Syntax strikt getrennt, somit gibt es kein Problem, herkömmlichen C-Code zu übersetzen. Dieser andersartige Ansatz macht neugierig, einmal selbst in die Tiefen des auf Nachrichten basierenden C-Derivates abzutauchen und einen genaueren Blick auf das Konzept hinter den vielen eckigen Klammern zu werfen, mit denen sich ein in Objective-C geschriebenes Programm präsentiert. Der TIOBE-Index, der seit 2001 ein monatlich aktualisiertes Ranking von Programmiersprachen nach ihrer Popularität publiziert, listet Objective-C hinter C und Java derzeit auf Platz 3 der am meisten verwendeten Programmiersprachen.<sup>1</sup> Die Tendenz: kontinuierlich steigend.<sup>2</sup> Diese Arbeit bietet eine attraktive Möglichkeit, sich im universitären Rahmen in diesen Bereich einzuarbeiten und einen Blick über den Tellerrand zu werfen. Eine Möglichkeit, die ich hiermit gerne ergreifen möchte.

Für die Entwicklung auf seinen Systemen stellt Apple mit Xcode eine Entwicklungsumgebung, die nach eigenen Angaben mächtig genug ist, dass sie von Apple selbst zur Entwicklung von Betriebssystemen und Anwendungen eingesetzt wird - und in den aktuellen Versionen angeblich eine sehr einfach zu erlernen-

---

<sup>1</sup>TIOBE Programming Community Index for November 2012, Headline: Objective-C on its way to become "language of the year" again

<sup>2</sup>TIOBE Programming Community Index Objective-C

de Oberfläche für den Entwurf der GUI mit sich bringt. Neben der Entwicklung einer Anwendung für einen Apple Macintosh bietet sich mit einer relativ neuen Technologie jedoch noch ein weiteres Stück Hardware des Unternehmens aus Kalifornien an: Im Jahr 2010 landete mit dem iPad die Sparte der “Post-PC”-Tablets im Mainstream-Bereich, die 2005 ihren ersten Kandidaten mit dem Nokia 770 Internet-Tablet hatte. Diese Post-PCs waren keine herkömmlichen Notebooks mit Tastatur mehr, die über für Stift-Bedienung ausgelegte Touchscreen verfügten, sie waren wesentlich leichter und dünner, liefen mit eigenen, angepassten Betriebssystem und verfügten über eine innovatives Bedienkonzept für Fingerberührungen - auch wenn das beim Nokia 770 noch nicht der Fall war. Auch heute liegt der Marktanteil von Apples iPad noch bei über 50 %<sup>3</sup>, so ist es sicherlich eine lohnende Erfahrung, den “Blick über den Tellerrand” zusätzlich auf dieses neuartige Stück Hardware zu richten und eine Anwendung zu schreiben, die auf Apples mobilem Betriebssystem iOS läuft.

Als Anwendungsbeispiel für diese ersten Schritte in einer neuen Umgebung wird ein Client implementiert, der die Verbindung mit einem mir ebenfalls neuen Bereich herstellt: RESTful Webservices. Die Abkürzung REST steht für Representational State Transfer und ist ein Programmierparadigma für Webanwendungen, welches im Grunde so alt ist wie das Internet selbst. Dieser erst 2000 in einer Dissertation eingeführte Standard beschreibt die Implementierung eines Webservices, mit dem die bereits im HTTP-Protokoll definierten Methoden GET, PUT, POST, DELETE und HEAD auf statischen Inhalten arbeiten. Diese werden wiederum auf permanente URLs abgebildet. Da die Funktionalität REST-konformer Webservices immer auf diese HTTP-Methoden abgebildet wird, sollte es möglich sein, einen generischen Client zu entwickeln, der mit all diesen Webservices irgendwie eine gemeinsame Kommunikationsschnittstelle bildet. Es sollen Anfragen gesendet und Antworten ausgewertet werden. Diese Antworten bekommt man in der Regel im Format XML oder JSON zurückgeliefert. Der Client sollte in der Lage sein, diese Daten zu analysieren, die potentiellen Ressourcen darin zu erkennen, sie zu validieren und aufzubereiten und dem Anwender eine einfache Möglichkeit bieten, mit diesen zu interagieren.

Auf dem Weg zum funktionsfähigen Client wird zunächst ein kurzer Einblick in die Werkzeugkiste gegeben, mit der hier gearbeitet wird. Es wird kurz die Programmiersprache Objective-C vorgestellt und einige Eigenheiten aufgezeigt, die man in dieser Form so nicht in der Java- oder C++-Welt findet. Weiterhin wird die von Apple verwendete API Cocoa Touch vorgestellt und die Entwicklungsumgebung Xcode, mit der auf diese zugegriffen wird. Im darauffolgenden Kapitel wird auf die Schnittstelle mit der REST-Welt eingegangen. Es wird erklärt, was im Ge-

---

<sup>3</sup>Der Apple-Marktanteil am Tablet-Markt betrug lt. einer IDC-Pressemitteilung vom 5. November 2012 50,4 %.

nauen unter einem REST-kompatiblen Webservice zu verstehen ist und wie man mit diesen kommuniziert. Den letzten größeren Bereich stellt die Implementierung der fertigen Software, deren Funktionalität anhand genereller Abläufe beschrieben wird. Weiter wird hier auf die Probleme, die sich während des Entwickelns dieser Arbeit ergaben, näher eingegangen. Abgerundet wird diese Arbeit durch einen reflektiven Abriss der vorhergegangenen Kapitel und einen persönlichen Ausblick darauf, ob die herausgearbeiteten Erfahrungen den Ausflug in die Welt der Apple-Systeme beenden oder den Grundstein bilden für weitere Entwicklungen auf den mobilen Systemen der Kalifornier.



## Kapitel 2

### Die Werkzeugkiste

## 2.1 Basissprache: Objective-C

Das Entwickeln der ersten App für Apple-Hardware bringt die Herausforderung mit sich, eine neue Programmiersprache erlernen zu müssen. So wie Microsoft für sein .NET-Framework die C#-Sprache entwickelte, welche Grundkonzepte der objektorientierten Hochsprachen C++, Java und Delphi in sich vereint, so verwendet Apple für seine Systeme die Sprache Objective-C. Anders als C# greift das in den frühen 80ern entwickelte Objective-C auf das prozedurale ANSI-C zurück. Seine objektorientierte Erweiterung ist eine Anlehnung an Smalltalk und wird als solche von der C-Syntax strikt getrennt. Die syntaktisch wichtigsten Neuerungen sind wie bei anderen Erweiterungskonzepten hauptsächlich die Verwendung von Klassen und Methodenaufrufen. Analog kann dieser Ansatz dazu verwendet werden, aus Pascal und Javascript, die beide von Haus aus über kein Klassenkonzept verfügen, die objektorientierten Erweiterungssprachen Objective-Pascal und Objective-J zu erzeugen.

Wie aus der C-Welt bekannt wird in einer Header-Datei mit der Endung `.h` das Interface deklariert, die Implementierungen erfolgt in der dazugehörigen `.m`-Datei, welche Objective-C- oder einfachen C-Code enthalten kann. Aus Kompatibilitätsgründen lassen sich Implementierungen mit der Dateiendung `.mm` anlegen, dieser darf explizit C++-Code enthalten, was die Weiterverwendung von C++-Bibliotheken erlaubt. Statt `#include` sollte man bei Objective-C auf `#import` zurückgreifen, da es sicher stellt, dass jede einzubindende Datei auch nur höchstens einmal eingebunden wird.

In Objective-C werden keine Funktionen mehr mit Parametern aufgerufen. Statt dessen verwendet man Objekte, die sich untereinander Nachrichten senden und so miteinander kommunizieren. Diese Nachrichten veranlassen ein Objekt, eine Methode auszuführen. Dieses Paradigma wird als Message Passings bezeichnet und unterscheidet sich stark von der Art der Methodenaufrufe, die man C++ verwendet. Es darf grundsätzlich jedes Objekt jede beliebige Nachricht an jedes Objekt senden - oder auch an sich selbst, in dem Fall wird der Bezeichner `self` verwendet - ganz gleich ob die Zielklasse oder eine beerbte Oberklasse die passende Methode zu dieser Nachricht implementiert. Da Methoden stets erst zur Laufzeit ermittelt werden und nicht bereits beim Übersetzen, entscheidet sich erst beim Aufruf, wie ein Objekt auf eine Nachricht reagieren wird. Dieses Verfahren bezeichnet man daher auch als dynamisches Binden. Im Gegensatz dazu werden die verwendeten C-Funktionen bereits beim Übersetzen statisch gebunden. Konsequenterweise existiert ebenfalls eine Methode, welche auf Nachrichten reagiert, für die keine eigene Implementierung durch die Klasse bereitgestellt wird. Abstrakte Klassen kennt Objective-C übrigens nicht. Jede Klasse muss vollständig implementiert werden, damit jederzeit Objekte dieser Klasse erzeugt werden können. Um dies zu umgehen lassen sich Protokolle deklarieren, in denen notwendige Methoden unter dem

@required-Tag aufgelistet werden.

Wie in C++ lassen sich aus einer Klasse Objekte erzeugen. Darüber hinaus ist aber jede Klasse auch selbst als Objekt ansprechbar. Diese enthalten jedoch keine Member-Variablen und sind stets Singletons, also Entwurfsmuster, die sicherstellen, dass von einer Klasse genau ein Objekt existiert, welches global verfügbar ist. Da ein solches Klassenobjekt ebenfalls für beliebige Nachrichten empfänglich ist, erfolgt das Binden seiner Methoden auch erst zur Laufzeit. Strings werden in Objective-C mit der Notation @"..." erzeugt. Ein solcher String ist ein Objekt vom Typ *NSString* und als solcher ebenfalls in der Lage, Nachrichten zu empfangen. Das vorangestellte @ ist ein Indikator für ein Objective-C-Objekt. Wird es bei der Angabe des String weggelassen, wird der bekannte, nullterminierte C-String erzeugt. Per Vorgabe besitzen die Datentypen in Objective-C die gleiche Typisierung wie in C: Sie sind *schwach*, *statisch* und *explizit*. Während eine *schwache* Typisierung Auskunft darüber gibt, dass Datentypen ineinander umgewandelt werden können, weist das Merkmal *statisch* darauf hin, dass der Datentyp bereits zur Übersetzungszeit geprüft wird. Im Gegensatz dazu werden dynamische Datentypen erst zur Laufzeit getestet. *Explizit* letztlich bedeutet, dass die Datentypen nicht erst per Typableitung ermittelt werden. Das Konzept der dynamischen Typisierung ist mit dem typlose Datentyp *id* umgesetzt, welcher einen Zeiger auf ein Objekt einer beliebigen Klasse repräsentiert und an den jede beliebige Nachricht gesendet werden kann. Ist für den tatsächlichen Typ dieses Objektes diese Methode jedoch unbekannt oder möchte man eine typlose Variable an eine typgebundene Variable zuweisen, deren Typen nicht übereinstimmen, so kommt es zu einem Laufzeitfehler.

```
id stringObject = @"Objective-C is great."; // OK
NSString *string = stringObject;           // OK
NSNumber *number = stringObject;           // logischer Fehler
char character = [number charValue];       // Laufzeitfehler
```

Instanzobjekte einer Klasse werden durch das Senden einer entsprechenden Nachricht an das zugehörige Klassenobjekt erzeugt. Für Gewöhnlich geschieht dies beim Programmstart, jedoch können im Gegensatz zu C++ bei Objective-C jederzeit auch während der Laufzeit noch neue Objekte erstellt werden. Überladene Methoden, wie man sie in C++/Java kennt, findet man so in Objective-C nicht mehr, denn eine Nachricht an eine Methode besteht aus dem Selektor, der sich aus dem Methodennamen und den Parametern zusammensetzt. Das Laufzeitsystem sucht bei einer eingehenden Nachricht nach einer solchen Methode und führt diese im Erfolgsfall aus. Wird keine passende Methode lokalisiert, wird ein *nil* zurückgeliefert. Möchte man sich absichern, lässt sich das Objekt im Voraus fragen, ob es eine bestimmte Nachricht versteht. Diese Fähigkeit der Selbstkenntnis beinhaltet ebenfalls das Ermitteln der eigenen Klasse und das Wissen, ob seine Klasse eine

Unterklasse einer bestimmten anderen Klasse ist. Der Begriff Reflexion sollte dem ein oder anderen ebenfalls aus der Java-/C++-Welt bekannt sein.

```
if([geometricFigure respondsToSelector:@selector(getShape)]) { ... }
```

Dem Smalltalk-Erweiterungsanteil verdankt Objective-C die verwendete Notation der eckigen Klammern, die ausdrückt, dass an dieser Stelle Nachrichten versendet werden. An erster Stelle wird das Empfängerobjekt genannt, anschließend folgt der Selektor und gegebenenfalls der Parameter. Sind Sender und Empfänger der Nachricht das gleiche Objekt, verwendet man den Bezeichner *self*.

```
// somewhere lost in code:
NSInteger age = [self computeAgeFromYear:1981 withMonth:5 andDay:23]
// method declaration:
- (NSInteger)computeAgeFromYear:(NSInteger)birthyear
    withMonth:(NSInteger)month
    andDay:(NSInteger)day { ... }
```

Diese Schreibweise ist für Neulinge äußerst gewöhnungsbedürftig, da hier jeder Parameter eine explizite Benennung erfordert. Methodennamen können in Objective-C durchaus sehr lang werden. Hat man sich jedoch einmal daran gewöhnt, mag man diese Notation jedoch nur ungern missen, denn sie fördert die Lesbarkeit und Verständlichkeit des Codes ungemein. Die obige Methode bezeichnet man vollständig als: *computeAgeFromYear:withMonth:andDay*, was auch gleichzeitig der Selektor dieser Methode ist.

Das Minus-Präfix zu Beginn der oben stehenden Methode gibt an, dass es sich hierbei um eine Instanzmethode handelt. Das entspricht dem Verständnis einer Methode in C++/Java. Da in Objective-C jedoch auch Klassen Objekte sind, können diese eigene Methoden haben. Solche Klassenmethoden werden durch ein vorangestelltes Plus gekennzeichnet. Ein gutes Beispiel ist der zum Initialisieren gebräuchliche verkettete Befehl `[[NSObject alloc] init]`, der aus den folgenden Methoden besteht: `+(id)alloc` ist die Klassenmethode, die Speicher für das Objekt reserviert und einen Zeiger darauf zurückliefert. Anschließend initialisiert sich das erzeugte Objekt in dem allokierten Speicherbereich durch den Aufruf der Instanzmethode `-(id)init`.

### 2.1.1 Protokolle: Data Sources und Delegates

Ein Protokoll ist einem Interface in Java sehr ähnlich: Es deklariert die Methoden, die in eine Klasse vorhanden sein müssen, welche es implementiert. Eine Protokolldeklaration beginnt mit dem Schlüsselwort `@protocol`, die enthaltenen Methoden befinden sich entweder unter dem Tag `@required` oder unter dem Tag `@optional`. Bei der Deklaration der Klasse wird das Protokoll in spitzen Klammern

hinter der Oberklasse angegeben, mehrere Protokolle werden durch Kommata voneinander getrennt:

```
@interface ViewController : UIViewController <UITextFieldDelegate,
    UITableViewDelegate, UITableViewDataSource> {
    ...
}
```

Ein Konzept, welches so ebenfalls nicht bei Java oder C++ anzutreffen ist, sind die so genannten *Delegates* (vom englischen delegation: Abordnung, Übertragung) und die *DataSources*. Mit ihnen ist es möglich, Nachrichten an ein delegiertes Objekt weiterzuleiten, die in einen bestimmten Funktionalitätsbereich fallen. Ein Textfeld sendet nach dem Betätigen der Eingabetaste die Nachricht *textFieldShouldReturn:(UITextField \*)* an ihre *Delegate*. Diese *Delegate* kann die eigene Instanz sein oder aber ein beliebiges anderes Objekt, welches das Protokoll *UITextFieldDelegate* implementiert. Der Methode *textFieldShouldReturn* spezifiziert, was beim Betätigen der Eingabetaste passieren soll. An dieser Stelle bietet es sich beispielsweise an, mittels `[textField resignFirstResponder]` das On-Screen-Keyboard wieder zurückzuziehen und eine weitere Methode aufzurufen, welche auf den im Textfeld enthaltenen Text zugreift. Die ordnungsgemäße Durchführung wird mit einem zurückgegebenen *YES* bestätigt. Ein Objekt kann nicht nur als Empfänger von Nachrichten dienen, er kann andersrum auch als Datenquelle fungieren. Implementiert es das einer Programmstruktur zugehörige *DataSource*-Protokoll, lässt es sich von dieser nutzen, um bei Laufzeit von dieser benötigte Daten abzufragen.

### 2.1.2 Properties

*Properties* werden in der Headerdatei deklariert und stellen einen einfachen Weg dar, Accessoren zu implementieren. War in früheren Xcode-Versionen noch das manuelle Synthetisieren mittels `@synthesize` notwendig, werden diese durch die Deklaration eines *Property* automatisch generiert. Ein `@property (UITextField*)name` erzeugt per default den Setter *setName* sowie den Getter *name*. Möchte man die Variable *name* nun setzen, werden zwei Möglichkeiten angeboten: den C-Weg oder die nachrichtenbasierte Objective-C-Notation. Ein `_name.text = string` wird intern in ein `[_name setText:NSString]` umgewandelt. In der Umsetzung dieser Arbeit wurde konsequent die Objective-C-Notation verwendet. Wenn man es ganz genau nimmt, führt Objective-C intern den folgenden Befehl aus: `[[self name] setText:NSString]`. Der Unterstrich zeigt, dass es sich um eine Instanzvariable handelt. Da Apple vorangestellte Unterstriche für eigene Zwecke einsetzt, sollte zur Vermeidung von Verwechslungen auf die eigene Verwendung bei der Namensgebung verzichtet werden.

## 2.2 API: Cocoa Touch

Apples mobiles Betriebssystem iOS lässt sich in vier grundlegende Schichten einteilen: Während das Core OS den Mach-Kernel sowie die grundlegenden Funktionen beinhaltet, stellen die Core Services die Frameworks bereit, welche die Grundlage aller Anwendungen und Dienste sind, die auf dem Gerät laufen. Die darüber liegende Media-Schicht bündelt die Multimediafunktionen, schlussendlich folgt an der Spitze die Programmierschnittstelle Cocoa Touch. Diese stellt die Schnittstellen zu sämtlichen darunter liegenden Schichten bereit. Gegenüber Cocoa, der Schnittstelle für OS X-Desktop-Maschinen, greift die Touch-Variante der API auf eine angepasste Benutzerschnittstelle zurück. Hier bie-

ten spezielle Unterstützung für Bewegungssensoren, Multitouch-Gestenerkennung und Animationen, die speziell auf iOS abgestimmt sind, dem Programmierer darauf abgestimmte Eingabeelemente und Events an. Die Basis von Cocoa Touch bildet zum einen das Foundation Framework. Dieses stellt alle Basisklassen zur Verfügung, die für die grundlegende Programmierung mit Objective-C unverzichtbar sind. Weiter werden in diesem Werkzeug wie Collections und Dateihandling bereitgestellt, ohne die eine moderne Programmierung heute nicht mehr auskommt. Seit iOS-Version 2 befindet sich darin bereits XML-Unterstützung, in iOS 5 wurde diese um Unterstützung für das JSON-Format ergänzt. Dies machte es im Rahmen dieser Arbeit sehr einfach, auf externe Frameworks zu verzichten. Das UIKit als weiterer integraler Bestandteil von Cocoa Touch beinhaltet Klassen, die speziell auf die Entwicklung der grafischen Oberfläche ausgerichtet sind. So enthält diese die Infrastruktur für grafische Anwendungen verschiedene Ansichten, so genannte Views, und deren Fenster, Menüs und Schaltflächen, sowie ein passendes Ereignissystem mit Sprachanbindung und Textsystem. Ereignisse durchlaufen eine so genannte "Responder-Chain", die aus unterschiedlichen Klassen besteht. Nacheinander werden dieser eingetroffene Ereignisse entnommen und verarbeitet. Seit OS X 10.4/iOS 3 wurde diese Sammlung um das Core Data-Framework ergänzt, einem schemabasierten Ansatz für die Umsetzung des MVC-Entwurfsmusters. Der Model-View-Controller ist in Cocoa Touch strikt umgesetzt; Klassen lassen sich



Abbildung 2.1: Schichtenmodell von iOS

in der Regel eindeutig zuordnen. Bei den Klassennamen selbst ist auffallend, dass die meisten mit NS beginnen, wie zum Beispiel *NSObject*, *NSString* oder *NSArray*. Dies hat seinen Ursprung darin, dass Apple im Jahre 1996 das OpenStep-Framework NeXTStep aufkaufte und als Basis für Mac OS X verwendete. Durch die Verwendung von “Klassen-Clustern” werden viele sichtbare Klassen nie instanziiert, sondern es werden statt dessen direkt zur Laufzeit Instanzen von passenden Subklassen erzeugt, von denen der Programmierer keine Kenntnis hat. Beispielsweise verlangt ein Anwendungsprogrammierer nach einer Instanz von *NSArray*, erzeugt wird jedoch in Abhängigkeit der Elemente eine Instanz einer Klasse, die er nicht kennt.

## 2.3 IDE: Xcode

Xcode ist Apples integrierte Entwicklungsumgebung und steht aktuell in der Version 4.5.2 zur Verfügung. Seit der Version 4.1 lässt sich Xcode frei zugänglich im App Store herunterladen. In der Version 4.4 sind die SDKs für OS X 10.8 sowie für iOS 5.1 enthalten, Version 4.5 bringt die SDK für das neue iOS 6 mit. Beide benötigen mindestens OS X in der Version 10.7.4 Lion oder 10.8 Mountain Lion. Die Implementierung dieser Arbeit wurde mit der Version 4.4 begonnen und auf 4.5 gewechselt. Jede Funktionalität wurde stets sowohl für iOS 5.1 als auch für iOS 6 getestet und gegebenenfalls angepasst. Mit Xcode lassen sich sowohl OS X- als auch iOS-Anwendungen entwickeln, wobei OS X-Anwendungen auf der Cocoa-API basieren; möchte man eine Anwendung für das mobile iOS entwickeln, so nutzt man hierfür das im vorherigen Kapitel vorgestellte Cocoa Touch-Framework. Xcode kommt mit einer Reihe von Werkzeugen - den Xcode Tools - welche die alltägliche Arbeit des Objective-C-Programmierers erleichtern sollen. Apples LLVM Compiler bietet eine Echtzeitüberprüfung des eingegebenen Codes und meldet auffallende Syntax- oder Schreibfehler umgehend. Oft bietet die eingeblendete Fix-it-Funktion gleich eine adäquate Lösung, die sich mit einem Klick übernehmen lässt. Natürlich darf man hier keine Wunder erwarten, im Rahmen dieser Arbeit hat sich deren Vorhandensein jedoch als sehr angenehm erwiesen. Mit dem Analysetool Instruments lassen sich diverse Messinstrumente mit einem Prozess verbinden und deren Laufzeitverhalten akribisch genau analysieren. Um ein Programmverhalten auf einem bestimmten iOS-Gerät zu testen, lässt sich auf den integrierten iPhone- und iPad-Simulator zurückgreifen, für den verschiedene iOS-Versionen bereitgestellt werden. Diese lassen sich aus Xcode heraus mit einem Klick installieren. Auf dem Simulator lässt sich die App testen, ohne dass man ein physisches Gerät zur Hand haben muss. Es lassen sich verschiedene Hardware-Events simulieren, vom Schwenk des Orientation-Modus über die Betätigung der Home-Taste bis hin zu einer Speicherwarnung, mit welcher man die eigene Software testen kann. Möchte man jedoch

Implementierungen von spezielleren Sensoren überprüfen, wie sie in den aktuellen Geräten eingesetzt werden, kommt man an einem physikalischen Gerät nicht vorbei. Das Interface von Xcode besteht zum einen aus dem Source Editor, welcher Komfortmerkmale wie Code Completion, Syntax Highlighting und Code Folding bereitstellt, sowie die äußerst angenehme Echtzeitsuche nach Fehlern, Warnungen und anderer kontextsensitiven Informationen. Diese werden in Nachrichtenblasen am Fehlerort eingeblendet. Daneben ermöglicht der Assistant Editor über ein zweigeteiltes Fenster den Schnellzugriff auf Dateien, die für gewöhnlich eine hilfreiche Kombination mit der geöffneten Datei darstellen, zum Beispiel wird neben dem Interface gleich deren Implementierung geöffnet. Der Interface Builder, welcher mit Version 4.0 von den externen Xcode Tools in die IDE integriert wurde, ermöglicht es, äußerst komfortabel und ohne eine Zeile Code, die GUI der App zu designen und die einzelnen Interfacekomponenten untereinander zu verknüpfen und in den Programmcode einzubinden. Der Organizer als weiteres Element ist die zentrale Verwaltung für die der Entwicklerlizenz hinzugefügten Geräte, die Versionsverwaltung über lokale sowie entfernte Repositorien, Projekte und deren Snapshots und für ein App Archiv, von dem aus man seine entwickelten Apps verwaltet und zum App Store sendet. Am Rand lassen sich weitere Leisten einblenden: Auf der linken Seite wäre dies der Navigator, der durch Dateien, Fehler und Logs navigiert; rechts lässt uns die Utility Bar einige Einstellungen vornehmen. Darüber hinaus befindet sich dort auch eine File Templates- und eine Code Snippets-Library im unteren Bereich für Dinge, die man immer mal wieder benötigt. Im unteren mittleren Bereich lässt sich selbstverständlich die für das Debugging unentbehrliche obligatorische Konsole einblenden. Da es vor allem auf mobilen Geräten eher zu Ressourcenknappheit kommt als auf Desktop-Systemen, sollte man hier einen entsprechend ökonomischen Programmierstil pflegen und Ressourcen, die nicht mehr benötigt werden nach der Verwendung wieder freigeben. In Xcode 4.2 wurde das *Automatic Reference Counting* eingeführt, um den Entwickler dabei zu unterstützen. Die bislang verwendeten retain und release-Nachrichten, mit denen Speicher freizugeben war oder diese Freigabe verhindert werden sollte, wurden obsolet, wenn das Projekt mit ARC genutzt wird. Statt dessen übernimmt nun der Compiler die Referenzzählung und fügt dem Code beim Übersetzen die entsprechenden retain und release-Nachrichten hinzu.

## 2.4 Interface Builder: Storyboards

Ebenfalls neu hinzugekommen seit Xcode 4.2 bilden Storyboards die Grundlage des Interface Builders und lösen die vorhergehenden NIBs ab, bei denen man darauf beschränkt war, zu jedem Zeitpunkt mit je einer View zu arbeiten. Storyboards zeigen im Gegensatz dazu die komplette Szene an, bestehend aus einzelnen



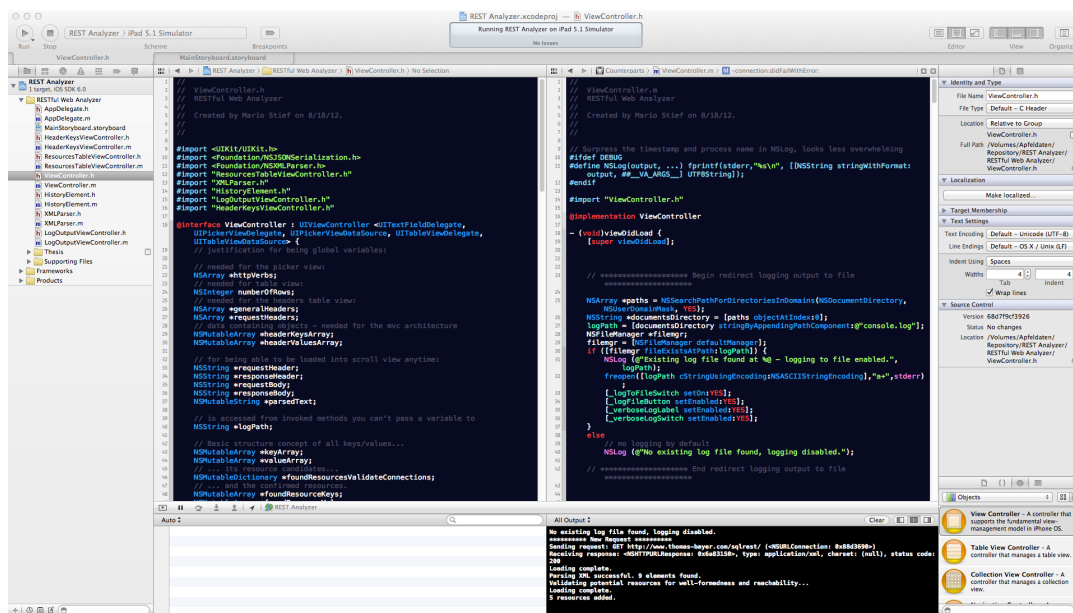


Abbildung 2.2: Xcode

Views und deren Übergänge untereinander. Verbindungen zu anderen Views wie die Möglichkeit, diese als *Delegates* oder als *DataSource* zu nutzen, lassen sich realisieren, indem man ein Objekt bei gedrückter Steuerungs-Taste auf die gewünschte View zieht. Zieht man diese statt dessen in den Programmcode, werden die entsprechenden Codefragmente erstellt, in denen man sofort mit der Implementierung beginnen kann, ohne sich mit programmatischen Details zum Setzen der einzelnen GUI-Elemente beschäftigen zu müssen. Übergänge lassen sich mit ID-Strings versehen, auf die man programmatisch zugreifen kann. So wird im Falle eines Überganges auf eine andere View die Methode *prepareForSegue* mit dem entsprechenden String für diesen Übergang aufgerufen. In dieser lassen sich Vorbereitungen für die neue Ansicht treffen - wie z. B. Daten und Referenzen übergeben - bevor diese dann geladen wird. Im Utility-Bereich kann praktisch jede Eigenschaft bearbeitet werden, mit der das Erscheinungsbild beeinflusst werden kann. So lässt sich ein Objekt inaktiv darstellen bzw. komplett verstecken oder man ändert die Schriftart eines Textfeldes. Um ein Objekt einem Container unterzuordnen lässt sich das unkompliziert per Drag and Drop realisieren. Zieht man beispielsweise eine *TextView* in einen *ScrollView*-Container erhält man Text, welcher sich mit einer Wischbewegung scrollen lässt.



Abbildung 2.3: GUI-Entwurf im Storyboard

## 2.5 Entwicklerlizenz: Testen auf echter Hardware

Grundsätzlich lässt sich eine App auch ohne Lizenz entwickeln und auf dem eingebauten iOS Simulator des seit der Version 4.1 für jedermann verfügbaren Xcode testen. Möchte man jedoch die App auf einem “echten” Gerät zur Ausführung bringen oder sie später einmal in den App Store stellen, so benötigt man eine Entwicklerlizenz von Apple. Diese kostet derzeit 99 USD bzw. 80 EUR und ist für exakt ein Jahr gültig. Mit einem Entwickleraccount hat man Zugriff auf iTunes Connect, mit dem sich der eigene Account, die verfügbaren iOS-Geräte sowie die eingestellten Apps verwalten lassen.

In einem ersten Schritt werden dort die User angelegt, welche an der Entwicklung und Distribution der Software beteiligt sind. Es stehen die vordefinierten Rollen *Admin*, *Technical*, *Finance* sowie *Sales* zur Verfügung. Während der *Technical* ausschließlich Rechte für die Verwaltung der Apps bekommt, stehen dem *Finance* weitergehend verschiedene Rechte im Bereich Finanzwesen, Analyse und Vertrieb bereit, die der Technican in der Regel nicht benötigt. Auf die Verwaltung der Apps hat er hingegen nur lesenden Zugriff, so kommen sich die Beiden nicht in die Quere. Der Rolle *Sales* stehen lediglich Verkauf und Verkaufsanalyse offen.

Abhängig vom Vertriebsmodell müssen in iTunes Connect weitere Angaben gemacht werden. Für unentgeltliche Apps ist es weder notwendig, einen Vertrag zu akzeptieren, noch müssen Bankdaten hinterlegt werden. Für weniger selbstlose Anwendungen stehen die Vertragstypen *iOS Paid Applications* sowie das auf in-App-Werbung spezialisierte *iAd Network* zur Verfügung. Hier sind entsprechend Vertragsbedingungen zu akzeptieren, Kontaktinformationen anzugeben, Landesangaben zur Versteuerung zu machen und die Bankdaten für den erwarteten Zahlungsverkehr zu hinterlegen. Um eine App direkt auf dem Gerät ausführen zu können, wird ein *Development Certificate* benötigt. Das hinzuzufügende Gerät wird an den Mac angeschlossen und im Organizer der Entwicklerlizenz zugeordnet. Beim ersten Mal meldet Xcode, dass kein *iOS Development Certificate* vorhanden ist und bietet an, eine solche von Apple anzufordern. Anschließend fügt er diesen automatisch der Schlüsselverwaltung hinzu und bietet an, das Entwicklerprofil für den Einsatz auf anderen Macs zu exportieren. In den *Build Settings* unserer App stellen wir nun die *Code Signing Identity* auf das neu erzeugte Zertifikat um und sind nun in der Lage, das angeschlossene iOS-Gerät anstelle des Simulators für die Entwicklung zu nutzen.

## 2.6 Distributionsalternative: Der Cydia Store

Zum offiziellen Vertriebsweg über den Apple App Store gibt es eine Alternative, über die jedoch nur freigeschaltete “gejailbreakte” Geräte Zugriff haben: Der

Cydia Store wird in der Regel bei einem Jailbreak gleich mitinstalliert. Mit diesem kann man sich, wie unter Linux-Systemen bekannt, mit APT-kompatiblen Repositorien verbinden und auf deren Angebot zugreifen. Der Cydia Store ist hierbei nicht als Distributionssystem für raubkopierte Software zu verstehen, denn auch er verfügt wie der App Store über ein Bezahlssystem, mit dem Entwickler ihre Anwendungen entgeltlich anbieten können. Interessant wird diese Vertriebsmethode, wenn man eine App entwickeln möchte, welche die von Apple eingeräumten Rechte überschreitet. Ein Beispiel für eine solche Anwendung ist der Fenstermanager Quasar, der eine Alternative zu Apples Einzelanwendungsmodus bietet. Vor allem auf iPads mit Retina-Auflösung kann es durchaus eine wünschenswerte Option sein, zwei oder mehr Fenster nebeneinander betreiben zu können. So könnte man beispielsweise auf der einen Seite ein Textverarbeitungsprogramm geöffnet haben, während auf der anderen Bildschirmhälfte ein Internetbrowser geöffnet ist. Um kompilierte Apps in ein DEB-Paket umzuwandeln – wofür nach wie vor eine Entwicklerlizenz benötigt wird – benötigt man das Paket dpkg. Dieses kann man sich beispielsweise mit dem Programm MacPorts<sup>1</sup> installieren, welches ähnlich Fink eine Paketverwaltung für freie und quelloffene Software für OS X ist.

Da sich dieses Programm nach `/opt/local/bin` installiert, fügen wir dieses Verzeichnis mit `export PATH=/opt/local/bin:/opt/local/sbin:$PATH` unserem Pfad hinzu. Der Befehl `sudo port selfupdate && sudo port install dpkg` führt ein Paketquellen-Update durch und installiert das gewünschte Paket dpkg. Mit diesem sind wir nun in der Lage, aus den fertigen Apps Cydia-Repository-konforme DEB-Pakete zu erstellen.

Zum Konvertieren in ein DEB-Paket besteht dpkg-deb jedoch auf eine Struktur, welche sich wie folgt aufbaut:

```
+-- MyProgram
|   +- Applications
|   |   +- MyProgram.app
|   |   +- Info.plist
|   |   +- MyProgram
|   |   +- icon.png
|   +- DEBIAN
|   |   +- control
|   +- System
|       +- Library
|           +- LaunchDaemons
|               +- com.identifizier.MyProgram.plist
```

Das Verzeichnis System ist hierbei optional.

Die Datei control kann folgende Einträge beinhalten: Package, Name, Version, Architecture, Description, Homepage, Depiction, Maintainer, Author, Sponsor,

---

<sup>1</sup>Projektpage: <http://www.macports.org>

Section. Zwingend vorgeschrieben sind davon jedoch nur Package und Version.

Darüber hinaus erstellt OS X, wenn TAR-Dateien erzeugt werden – die ein Teil der internen Struktur eines Debian-Paketes darstellen – einige Dateien mit der Endung `_*`, welche zusätzliche Informationen enthalten. Diese zusätzlichen Dateien würden zusammen mit dem erzeugten Paket installiert werden. Davon abgesehen, dass diese Dateien sowieso nicht zum Paket gehören, könnten Sie auf dem Zielsystem Konflikte mit anderen Paketen erzeugen. Um dieses Feature zu deaktivieren werden durch `export COPYFILE_DISABLE && export COPY_EXTENDED_ATTRIBUTES_DISABLE` die erforderlichen Umgebungsvariablen exportiert. Abschließend lässt sich im Verzeichnis *MyProgramm* durch `dpkg-deb -b MyProgram` das DEB-Paket erzeugen.

### Vom DEB-Paket aufs Device

Um die nun fertige DEB-Datei auf das iPad zu bekommen, wird ein Cydia-kompatibles Repositorium benötigt. Hierfür kann zum Beispiel auf das kostenlose Angebot von *MyRepoSpace* zurückgegriffen werden. Über ein Webinterface können die Pakete hochgeladen werden, anschließend sind sie in Cydia über das Repositorium <http://cydia.myrepospace.com/Username/> zugreifbar.

## Kapitel 3

### Schnittstelle mit der REST-Welt

## 3.1 RESTful Webservices

### 3.1.1 Begriffseinführung

Wikipedia definiert den Begriff *Webservice* als “eine Methode der Kommunikation zwischen zwei elektronischen Geräten über das Internet”[5]. Man verbindet mit diesem Begriff primär Akronyme wie SOAP (Simple Object Access Protocol) - ein Netzwerkprotokoll für den Transfer von Daten und RPCs (Remote Procedure Calls) - oder den einfacheren XML-RPC. Diese Webservices arbeiten, wie der Name bereits ausdrückt, mit HTTP POST-Requests, um auf einem meist entfernten Computer eine Methode aufzurufen. Die Kommunikationssprache ist die XML-basierte *Webservices Description Language* oder kurz WSDL.

Im Jahr 2000 führte Roy Fielding, einer der Autoren der HTTP-Spezifikationen 1.0 und 1.1 und Mitbegründer des Apache HTTP Projektes, in seiner Dissertation *Architectural Styles and the Design of Network-based Software Architectures* den Begriff *REpresentational State Transfer* ein und definierte ihn als einen “Architekturstil für verteilte hypermediale Systeme, der die Richtlinien für die Softwareentwicklung spezifiziert, welche sowohl REST selbst leiten als auch die Interaktionsvoraussetzungen, die so gewählt werden, dass sie diesen Richtlinien entsprechen”[1]. Die REST-Architektur ist also ein Modell, das spezifiziert, wie das Internet - das im Grunde bereits eine riesige REST-Anwendung darstellt - eigentlich funktionieren sollte. Fielding bezeichnet REST als eine hybride Architektur, die von vielen verschiedenen Netzwerk basierten Architekturstilen abgeleitet ist und bezeichnet Seiten, die mit der Architektur REST kompatibel ist, als *RESTful*.

Den Kern dieses Architekturmodells bilden so genannte Ressourcen, weswegen es auch gerne als Resource-oriented Architecture bezeichnet wird. Alles, was sich eindeutig identifizieren lässt, ist eine Ressource und sollte über eine statische Adresse zugreifbar gemacht werden. Neben Dateien und Verzeichnissen sind auch ausführbare Methoden, die Datenbankabfragen auslösen, gebräuchliche Ressourcen. Hervorragende Kandidaten stellen auch die Entitäten in einem Datenmodell: Für einen Onlineshop bieten sich die Entitäten Artikel, Kunde und Bestellung an, aber auch *Bestellungen des Kunden mit der ID 42 aus Jahr 2010* kann eine valide Ressource sein und sollte ebenfalls eine möglichst langlebige und stabile URL erhalten. Ist man sich bei der Wahl der Ressourcen nicht ganz sicher so gilt: Im Zweifel besser zu viele Ressourcen als zu wenige.

### 3.1.2 Richtlinien

HTTP bringt bereits ressourcenbezogenes Caching mit sich, wodurch man das Gefühl bekommt, mit einer REST-Anwendung im Internet bereits irgendwie “zu Hause zu sein”. Viele Onlineshops, Suchmaschinen oder Buchungssysteme sind,

auch ohne, dass sie gezielt darauf abzielen, bereits RESTful. Damit eine Anwendung das HTTP jedoch auch tatsächlich REST-konform nutzt, hat sie sich an die Richtlinien zu halten, wie sie Fielding in seiner Arbeit beschreibt. Diese Richtlinien lassen sich wie folgt untergliedern, wobei es den einzelnen Diensten obliegt, wie diese implementiert werden.

### **Adressierbarkeit**

SOAP Webservices wie auch viele HTML-Seiten bilden ihre Funktionalität auf eine einzelne URL ab. Das sieht zwar in der Adresszeile sauber aus, bringt jedoch Nachteile mit sich. Weder lassen sich gezielt Inhalte als Lesezeichen ablegen, noch kann man mal eben dem Freund einen Link auf ein Produkt zusenden und auch Suchmaschinen haben es hier schwer, denn auch diese benötigen zum direkten Zugriff auf Unterbereiche einen konkreten Zugriffspfad. RESTful gestaltete Anwendungen bieten die geforderte Adressierbarkeit, ohne dass Einschränkungen in Kauf genommen werden müssen. Durch die Verwendung einer Firewall beispielsweise ist es ohne Weiteres möglich, gezielten Zugriff auf eine Ressource zu unterbinden. Natürlich ließe sich das URL-Prinzip ebenfalls auf SOAP anwenden, aber sobald diese dereferenziert werden, bewegen wir uns wieder in der REST-Welt.

### **Unterschiedliche Repräsentationen**

Von den hinter einer URL bereitgestellten Diensten lassen sich unterschiedliche Darstellungen anfordern. Diese bezeichnet man als die Repräsentation dieser Ressource. Über die im HTTP-Protokoll enthaltenen *Accept*- und *Content-Type*-Header unterstützt das Internet ebenfalls bereits die *Content Negotiation*, mit der sich unterschiedliche Repräsentationen ein- und derselben Ressource anzeigen lassen. Hierbei dürfen sich die Clients das Format anfordern, welches am ehesten ihren Bedürfnissen entspricht: Ein Browser zeigt sich meist mit einer HTML-Repräsentation zufrieden, während andere Client - wie der in dieser Arbeit vorgestellte - ist eher an der XML- oder an der JSON-Repräsentation interessiert. Darüber hinaus lassen sich sogar Versionswünsche über die Content Negotiation realisieren und beispielsweise der gewünschte Content nach Belieben in XML 1.1 oder in XML 1.2 geliefert werden.

### **Zustandslosigkeit**

REST ist konzeptuell ein zustandsloses Protokoll, somit ist auch jeder RESTful Webservice zustandslos. Weder Server noch der Webservice behalten sich den aktuellen Stand ihres Dialoges. Jede Nachricht muss stets alle Informationen beinhalten, die erforderlich sind, diese Nachricht korrekt zu interpretieren und die



Anfrage ordnungsgemäß verarbeiten zu können. Da sich weder Server noch Client einen Zustand behalten, bezeichnet man das Kommunikationsprotokoll einer REST-konformen Architektur als zustandslos. Durch die Abgeschlossenheit der einzelnen Nachrichten lassen sich darüber hinaus Lasten sehr leicht auf mehrere Maschinen verteilen, was sich positiv auf die Skalierbarkeit des Webservice auswirkt. In der Praxis wird jedoch auch gerne auf Cookies und andere Techniken zurückgegriffen, um über den Request hinaus an Zustandsinformationen gelangen.

## Operationen

Damit RESTful gestaltete Webservices miteinander kommunizieren können muss eine gemeinsame Sprache gesprochen werden. Das HTTP-Protokoll kennt unter anderem die Optionen *GET*, *POST*, *PUT*, *DELETE*, *HEAD* und *OPTIONS*. Da auf jede REST-Ressource stets die gleichen Operationen angewandt werden sollen, definiert dieser Methodensatz gleichzeitig die vom REST-Architekturstil geforderten wohldefinierten Operationen. Durch die Verwendung dieser elementaren Webtechniken wird die Integration und Interaktion der beteiligten Softwarekomponenten deutlich vereinfacht.

Die *GET*-Methode hat hierbei sicher zu sein: Es darf nur gelesen werden, so dass der Client jederzeit bedenkenlos ein *GET* auf eine Ressource anwenden darf. Weiterhin diktiert die HTTP-Spezifikation die idempotente Implementierung von *GET*, *PUT* und *DELETE*. Wird eine dieser Operationen mehrfach ausgeführt, wird stets das gleiche Ergebnis erwartet. Ist sich der Client unsicher, ob sein Request erfolgreich abgearbeitet wurde, sollte er ihn wiederholen dürfen, ohne dass Seiteneffekte erwartet werden. *HEAD* und *OPTIONS* finden in REST eher selten Verwendung. Die REST-konforme Verwendung der Methoden in einem kurzen Überblick:

**GET** fordert die Repräsentation der Ressource vom Server an.

**POST** fügt eine neue Unterressource zur angegebenen Ressource ein. Da die Unterressource vorher noch nicht existierte, wird die URL zu dieser vom Server erzeugt und dem Clienten in der Response zurückgeliefert. Über diese Verwendung hinaus kann POST auch für Operationen verwendet werden, die sonst von keiner anderen Methode abgedeckt werden.

**PUT** legt die ihr enthaltenen Ressource an. Existiert sie bereits wird sie wie im Body angegeben abgeändert.

**DELETE** entfernt die angegebene Ressource.

**HEAD** fragt die Metadaten einer Ressource ab.

**OPTIONS** bringt in Erfahrung, welche Methoden auf eine Ressource angewendet werden dürfen.

Clientseitig sollte das folgende Protokoll anwendbar sein:

```
@protocol Resource
- (NSURLResponse)get
- (NSURLResponse)post:(NSURLRequest *)request
- (NSURLResponse)put:(NSURLRequest *)request
- (NSURLResponse)delete
- (NSURLResponse)head
@end
```

Die (vereinfachte) Implementierung der Anlage eines Kundenkontos im Server kann in etwa wie folgt aussehen:

```
@interface Customers : NSObject <Resource> {
    NSURL *resource;
}
@implementation Customers {
    ...
    (NSURLResponse)post:(NSURLRequest *)request {
        NSInteger customerId = [self createCustomer:request];
        ...
        return [initWithURL:newResource
                    statusCode:201
                    HTTPVersion:@"HTTP/1.1"
                    headerFields:headerFields];
    }
    ...
}
```

## Hypermedia

Das Kofferwort *Hypermedia* ist eine Zusammensetzung aus den Begriffen *Hypertext* und *Multimedia* und weist auf die Verwendung von Hypertext mit starkem Akzent auf den multimedialen Gesichtspunkt hin. Repräsentationen enthalten in der Regel neben Informationen selbst wieder Links zu anderen Ressourcen. Das macht das Internet zu dem, was es ist: Eine stark vernetzte Verbindungsstruktur von identifizierbaren Ressourcen über hypermediale Links auf andere identifizierbare Ressourcen. Die standardisierte Adressierung sorgt dafür, dass sich die verlinkte Ressource in einem anderen Prozess im gleichen System oder auf einem anderen Rechner befinden kann, ganz gleich ob sich dieser im lokalen Netzwerk befindet, an einem beliebigen anderen Ort dieses Planeten oder auf einer Raumstation. Hierfür wird auch gerne der Ausdruck Verbindungshaftigkeit verwendet. Im Optimalfall werden dem Clienten in der angeforderten Repräsentation gleich

die nächsten möglichen Übergänge als hypermediale Links mitgeteilt. Der Wert einer Anwendung ist proportional zur Anzahl der Ressourcen, auf die sie verlinkt. Mit einem Link auf einen REST-konform konzipierten Webservice eines Telekommunikationsunternehmens würden mit wenigen Klicks Millionen von Kundendaten zugreifbar gemacht werden. Diese könnten von der Anwendung selbst ausgewertet oder dem Endanwender zur Weiterverarbeitung überlassen werden.

### 3.1.3 Beispiel

Wie so oft lässt sich das Prinzip von REST am Besten an einem Beispiel zeigen. Wir betrachten die Schnittstelle für den fiktiven Onlineshop `smartphonesep-pel.de`. Ein Neukunde möchte sich ein Smartphone bestellen und legt sich zu diesem Zweck ein Kundenkonto an. Er wechselt auf die Seite des Onlineshops mit `GET http://www.smartphonesep-pel.de` und teilt dem Server des Onlineshops mit einem `POST /customer` seine Registrierungsabsicht mit. Im Requestbody stehen seine Daten, die er für die Registrierung angegeben hat. Der Server bestätigt mit der Response

```
HTTP/1.1 200 OK
Content-Type: text/xml;
Content-Length: 44
```

```
http://www.smartphonesep-pel.de/customer/1337
```

die erfolgreiche Anlage des Kundenkontos und teilt die vergebene Kundennummer mit. Die Bestandteile einer HTTP-Response sind stets der Statuscode - an dieser Stelle weist der Statuscode *201 CREATED* darauf hin, dass eine neue Resource angelegt wurde -, die Art des zurückgelieferten Contents sowie der Content selbst. Nun kann sich der Kunde mit `GET /customer/1337` sein Kundenkonto anzeigen lassen. Eine Änderung ist mit einem `PUT /customer/1337` möglich. Möchte er sich den Artikel Samsung Galaxy S3 anzeigen lassen, kann er das per `GET /article/galaxy_s3`. Angenommen es gefällt ihm, so packt er es sich mit einem `PUT /shoppingcart/1337&article=galaxy_s3` in seinen Warenkorb. Zur Sicherheit überprüft er nun dessen Inhalt mit `GET /shoppingcart/1337`. Die Antwort des Onlineshops könnte wie folgt aussehen:

```
HTTP/1.1 200 OK
Content-Type: text/xml

<?xml version="1.0"?>
<shoppingcart xmlns:xlink="http://www.w3.org/1999/xlink">
  <customer xlink:href="http://www.smartphonesep-pel.de/customer/1337">
    1337
  </customer>
```

```

<position pos="1" amount="1">
  <article xlink:href="http://www.smartphoneseppele.de/article="galaxy_s3"
    article="galaxy_s3">
    <description>Samsung Galaxy S3</description>
  </article>
</position>
</shoppingcart>

```

Kunde 1337 ist mit dem Ergebnis einverstanden und sendet mit `POST /shoppingcart/1337` seine Bestellung ab. Der Server quittiert ihm dies mit einem `/shoppingcart/1337/open`, was auf den offenen Bestellstatus hinweist. Nun erscheint jedoch unerwartet das iPhone 5 und der Kunde überlegt es sich anders. Er überprüft mit einem `GET /customer/1337/orders` seinen Bestellstatus und er hat Glück, denn die Bestellung wurde noch nicht versendet. Er storniert seine noch offene Bestellung mit `DELETE /customer/1337/orders/open/2012-09-11_001` und legt sich statt dessen mit `PUT /shoppingcart/1337&article=iphone_5` den neuen Artikel in seinen Korb.

Der Onlineshop seinerseits entfernt nach dem Release des iPhone 5 mit `DELETE /article/iphone_4s` das iPhone 4S aus seinem System und legt mit `PUT /article` das neue Modell an. Der Body könnte wie folgt aussehen:

```

<articles>
  <name>iPhone 5</name>
  <description>Das beste iPhone, das es je gab.</description>
  <size>64</size>
  <weight>140</weight>
  <price>849</price>
</articles>

```

Er sendet Kunde 1337 das gewünschte iPhone 5 und verschiebt seine Bestellung vom Status offen nach versendet mit `DELETE /customer/1337/orders/open/2012-09-12_001` und `POST /customer/1337/orders/shipped/2012-09-12_001`.

## 3.2 XML

Die Codebeispiele im vorhergehenden Kapitel zeigen bereits, dass sich hier zur Kommunikation eines Austauschformates bedient wird, welches sowohl von Menschen gelesen als auch von Maschinen unabhängig von Plattform und Implementierung geparkt werden kann. XML steht für Extensible Markup Language (zu Deutsch: erweiterbare Auszeichnungssprache) und verwendet das - im einfachsten Fall im ASCII-Format kodierte - Textformat, um über eine Metasprache strukturiert Informationen verfügbar zu machen. Diese vom World Wide Web Consortium (W3C) herausgegebene Spezifikation ist aktuell in der 5ten Ausgabe verfügbar und ein Derivat des älteren SGML. In vielen Punkten ist XML sehr verwandt mit

HTML, jedoch folgt XML einer konsequenteren Syntax. Das führt dazu, dass valide XML-Dokumente zuverlässig von verschiedenen Anwendungen gelesen werden können. Um die zur Formatierung verwendeten Zeichen `<`, `>`, `&`, `"` und `'` in Texten einzubetten, greift man auf die Entitäten `&lt;`, `&gt;`, `&amp;`, `&quot;` sowie `&apos;` zurück. Weitere feste Entitäten gibt es nicht, jedoch lassen sich beliebig weitere Entitäten definieren.

Die Struktur von XML bietet einige Vorteile gegenüber anderen Austauschformaten: Die redundante vollständige Wiederholung des Tagnamens beim Schließen erschwert Fehler bei der Verschachtelung und die vergleichsweise einfache Syntax von XML mit seinen Element- und Attributbezeichnungen macht es Einsteigern leicht, gleich zu Beginn ein gutes Gefühl zu bekommen, wie man mit XML umzugehen hat. Da die volle XML-Spezifikation allerdings über einen Umfang von gut 30 Seiten verfügt, soll an dieser Stelle nur ein kleiner Überblick über das Basiswissen im Umgang mit XML vermittelt werden.

Ein einfaches, wohlgeformtes XML-Dokument könnte wie folgt aussehen:

```
<?xml version="1.0" encoding="UTF-8" ?>
<content>
  <chapter no="1">
    <description>Einleitung</description>
  </chapter>
  <chapter no="2">
    <description>Die Entwicklungsumgebung</description>
  </chapter>
  <chapter no="3">
    <description>REST</description>
  </chapter>
  <!-- at least chapter IMPLEMENTATION still missing -->
</content>
```

*Wohlgeformt* bedeutet im Kontext eines XML-Dokumentes, dass es sich an die offiziellen Regeln des W3C zur Erstellung von XML-Dokumenten hält. Es muss exakt ein Wurzelement vorhanden sein, welches das gesamte Dokument umschließt. Die einzelnen Elemente sind Informationsträger, ganz gleich welcher Art, sie können Texte oder weitere Elemente enthalten oder eine Kombination aus beiden. Die Elementbezeichnung ist frei wählbar. Verfügt ein Element über einen Inhalt, so wird dieser mit einem Start- sowie mit einem Endelement versehen: `<tag>content</tag>`. Steht ein Element ohne Inhalt, so kann das Element den Ein- und den Auszeichner in sich vereinen: `<no_content/>`. In einem Einzeichner oder einem leeren Tag können Attribute enthalten sein, die aus einem Schlüssel/Werte-Paar bestehen. Mehrere gleichlautende Elemente nebeneinander sind kein Problem, mehrere gleichlautende Attribute in einem Element sind jedoch nicht zulässig. Die Verschachtelung erfolgt ebenentreu, ein Element muss also geschlossen werden, bevor ein Geschwisterelement beginnt oder ein Elternelement geschlossen wird. Über die Wohldefiniertheit

hinaus besteht die Möglichkeit, weitere Anforderungen an unser Dokument zu stellen, welches sich mit einer Grammatik wie einer DTD (Document Type Definition) oder einem XML-Schema realisieren lässt. Ein Dokument, welches einen Verweis auf eine solche Grammatik enthält, diese auch einhält und darüber hinaus noch wohlgeformt ist, bezeichnet man als valid.

### 3.3 JSON

XML ist nicht immer der ideale Weg seine Daten zu strukturieren. Das Tag-System bläht kleine Datenbestände schnell auf und das Ansprechen einzelner XML-Nodes ist nicht immer leicht. JSON - die Kurzform für Java Script Object Notation - ist XML sehr ähnlich und bietet eine leichtgewichtige Alternative. JSON ist ein auch vom Menschen gut lesbares Austauschformat, welches so konzipiert ist, dass es sich leicht erstellen und leicht parsen lässt. Für ein sprachenübergreifendes Austauschformat ist es natürlich sinnvoll, dass man für den Aufbau auf Strukturen zurückgreift, die gängigen Sprachen geläufig sind. Somit kommen in JSON Key/Value-Records und Arrays zum Tragen - Strukturen, von denen es undenkbar ist, dass man sie in einer halbwegs modernen Programmiersprache nicht in irgendeiner Form wiederfindet. Ein Objekt folgt in JSON der Syntax *{ String : Value }*. Mehrere solcher Paare werden durch Kommata voneinander getrennt: *{ String1 : Value1, String2 : Value2, ... }*. Leerzeichen können zwischen den einzelnen JSON-Elementen der Übersichtlichkeit wegen beliebig gesetzt werden. Wie aus den meisten Programmiersprachen bekannt, kennzeichnet die eckige Klammer ein Array: *[ Value1, Value2, Value3, ... ]*. Ein Wert kann hierbei ein oben genanntes String/Value-Objekt sein, aber auch Strings, Zahlen oder die boolschen Werte true und false sind möglich. Darüber hinaus sind auch null und weitere Arrays zulässige Werte. Zeichenketten, bestehend aus 0 bis n Unicode kompatiblen Zeichen, beginnen und enden je mit einem Anführungszeichen und können, ebenfalls wie aus C und Java bekannt, Escape-Sequenzen beinhalten, beginnend mit einem Backslash. Diese sind im Folgenden: `\`, `\\` sowie `\/` geben jeweils den Wert hinter dem `\` aus, also `"`, `\` und `/`. Hingegen steht `\b` für ein Backspace, `\f` für einen Seitenumbruch, `\n` kennzeichnet eine neue Zeile, `\r` den Zeilenrücklauf und `\u` einen horizontalen Tabulator. Schließlich kann man noch, beginnend mit `\u`, einen vierstelligen Zahlencode eingeben, an dessen Stelle wird dann der Wert an der entsprechenden Stelle der Unicodetabelle eingefügt. Die Zahlensyntax ist ebenfalls aus C und Java bekannt mit der Ausnahme, dass JSON weder die Oktal- noch die Hexadezimaldarstellung beherrscht. 42 ist ebenso zulässig wie 1.35e-4 oder auch -3.7589E+20.

Ein Beispiel für eine JSON-Datei:

```
{
  {
    "customer-id": 1337,
    "name": "John Doe",
    "contact": [
      3456,
      "123-4567"
    ], "john.doe@example.com"
  ],
  "vip": true
},
{
  "customer-id": 1338,
  "name": "Jane Doe",
  "contact": null,
  "vip": false
}
}
```

## Kapitel 4

# Implementierung



## 4.1 Ein erster Überblick

Erstellt man ein neues Projekt, hat man verschiedene vorgefertigte Templates zur Auswahl. Die App sollte nicht mit Optionsvielfalt erschlagen und alles intuitiv aus einer View heraus erreichbar sein. Die Wahl fiel auf ein einfaches *Single View Application*-Template und gegen einen *NavigationController*, der einen zentralen Navigator bereitstellt, von dem aus man die einzelnen Views anwählen kann. Erzeugt man ein Projekt, liefert Xcode bereits einige Template-Dateien, in denen man sofort losprogrammieren darf, dazu gehören der *ViewController* und das *MainStoryboard*.

Der *ViewController* ist hierbei die Initialisierungs-View und das zentrale Bedieninstrument dieser App. Jede View gehört in eine eigene Klasse, was somit den Klassen *HeaderKeysViewController*, *ResourcesTableViewController* sowie *LogOutputViewController* ihre Existenzberechtigung verleiht. Keine eigene View besitzt der *XMLParser*, dessen Funktionalität sich jedoch hervorragend als autonome Klasse ausgliedern lässt. *HistoryElement* ist die Implementierung einer URL-History in Form einer doppelt verketteten linearen Liste.

## 4.2 Hauptklasse: ViewController

Der *ViewController* als unsere Basisklasse erbt vom Typ *UIViewController*, welche das fundamentale View-Management-Modell für alle iOS Applikationen stellt. Seine Oberklasse *UIResponder* definiert das Interface zum Umgang mit Events und erbt selbst direkt von *NSObject*, der Wurzelklasse aller Objective-C-Klassen. Er wird uns von Xcode zu Projektbeginn bereitgestellt und fügt die Methoden *viewDidLoad*, *viewDidUnload* mit je einer kurzen Beschreibung ein sowie die Methode *shouldAutorotateToInterfaceOrientation*, mit der sich der Landscape Orientation-Modus ein- und ausschalten lässt. Letztere sorgt dafür, dass sich der Bildschirminhalt mitdreht, wenn die Lagesensoren des iPads ein Kippen des Gerätes erkennen. Aufgrund des fehlenden *NavigationControllers* muss jede weitere View ihr aus erreichbar sein. Die View beinhaltet mehrere Textfelder und implementiert demnach das *UITextFieldDelegate*-Protokoll. Dieses Protokoll definiert die darin möglichen Nachrichten, die ein Textfeld als Teil der Textverarbeitung an seine *Delegate*-Klasse sendet. In der *ViewController* wird beispielsweise die im Protokoll enthaltene Methode *textFieldShouldReturn* so implementiert, dass nach dem Druck der Eingabetaste auf der On Screen-Tastatur das Keyboard wieder vom Bildschirm ausgeblendet wird. Sofern das URL-Feld das gerade aktive Textfeld war wird anschließend die Methode *go* aufgerufen, was den gleichen Effekt hat wie das Berühren des Go-Buttons oben rechts am Bildschirm. Für das Textfeld, welches diese Methode aufrufen möchte, muss zwingend eine Delegate spezifiziert werden.



Abbildung 4.1: Klassen als UML-Diagramm

Dazu wird im Storyboard bei gedrückter Steuerungstaste eine Verbindung vom URL-Textfeld zum Controller gezogen und das Outlet *delegate* ausgewählt. Das Interface des entsprechenden Controllers wird automatisch um die entsprechende Deklaration des Protokolls erweitert. Der programmatische Weg wäre das manuelle Setzen der Delegate mittels `[_url setDelegate:self]`. Noch einmal zur Erinnerung: `url` wird als *property* im Interface deklariert und deshalb werden standardmäßig die Accessoren generiert. Der Zugriff auf `url` über den Getter erfolgt über die Instanzvariable `_url`.

Beispielhafte Implementierung der Methode `textFieldShouldReturn` im View-Controller:

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    [textField resignFirstResponder];
    if (textField == _url)
        [self go:nil];
    else if (textField == _username)
        [_password becomeFirstResponder];
    else if (textField == _keyTextField)
        [_valueTextField becomeFirstResponder];
    else if (textField == _valueTextField)
        [self addKeyValue:nil];
    return YES;
}
```

Weitere Elemente in dieser View, die eine *Delegate* benötigen, sind zum einen die *PickerView* links oben, in der die zu verwendende HTTP-Methode ausgewählt wird sowie die *TableView* auf der rechten Seite, in der sich Schlüssel/Werte-Tupel aufnehmen lassen. Diese werden bei einem PUT oder einem POST automatisch als Header hinzugefügt. Diese beiden Komponenten benötigen nicht nur ein Objekt, dem sie Nachrichten übermitteln können, die im weiteren Programmverlauf etwas anstoßen. Beide benötigen darüber hinaus ebenfalls ein Objekt, bei dem sie Daten abfragen dürfen, die sie für ihre eigene Funktion benötigen. Das *DataSource*-Objekt liefert als Antwort auf die im Protokoll deklarierten Nachrichten, die es von der *PickerView* oder der *TableView* empfängt, die benötigten Daten zurück. Oft handelt es sich dabei bei der *Delegate* und der *DataSource* um ein- und dasselbe Objekt, daher implementiert der *ViewController* sowohl die *Delegate*-Protokolle *UIPickerViewDelegate* und *UITableViewDelegate* als auch die *DataSource*-Protokolle *UIPickerViewDataSource* und *UITableViewDataSource*.

Auf die Anfrage der *PickerView* nach der Anzahl der Spalten reagiert das *ViewController*-Objekt mit der folgenden Methode:

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView {
    return 1;
}
```

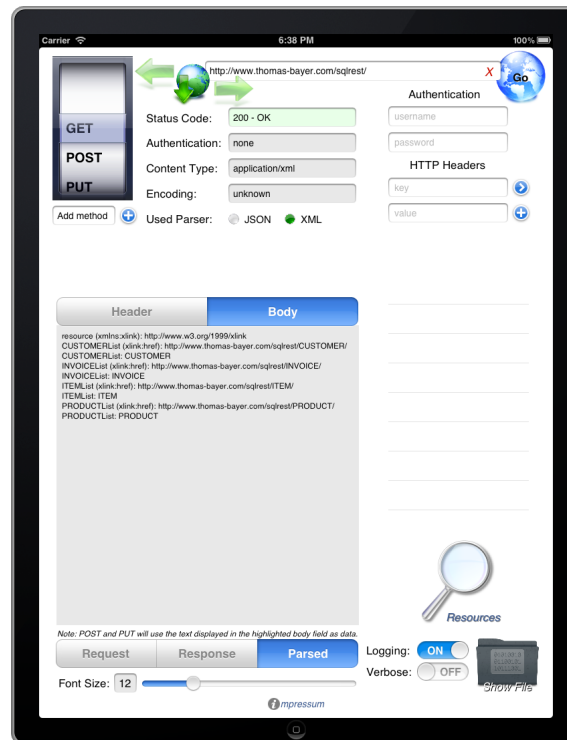


Abbildung 4.2: ViewController nach geparster XML-Response

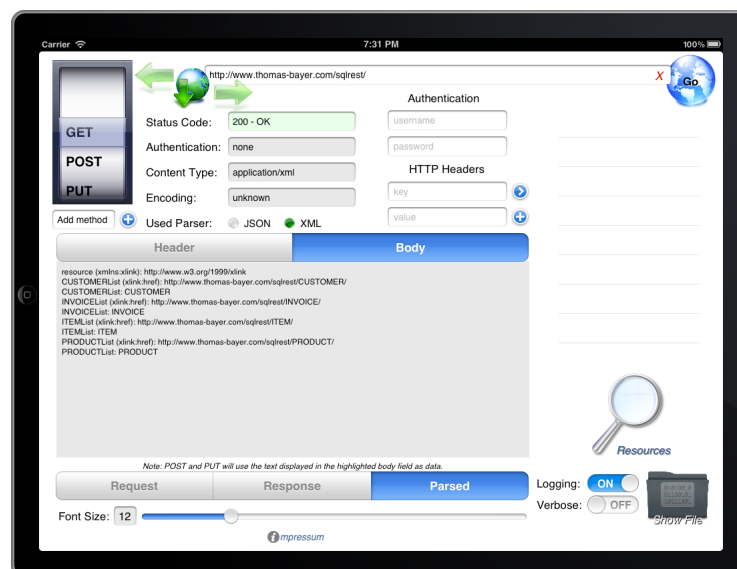


Abbildung 4.3: ViewController im Landscape-Modus

### 4.2.1 Senden eines Requests

Beim Senden eines Requests passiert Folgendes: Im URL-Textfeld ist die Adresse einzugeben, zu welcher der Request gesendet werden soll; im linken oberen Picker wird die HTTP-Methode ausgewählt. Das Textfeld direkt unter dem Picker bietet die Möglichkeit, zusätzliche Request-Methoden aufzunehmen wie beispielsweise OPTIONS, das standardmäßig nicht enthalten ist. Soll ein PUT- oder ein POST-Request versendet werden, so wird der im Body-Bereich angezeigte Text des aktuell aktiven Tabs im Output-Fenster (Request/Response/Parsed) dem Request als Body hinzugefügt. Dieser Textbereich ist aus diesem Grund vom Anwender bearbeitbar. Es sollte natürlich niemals der Parsed-Tab aktiv sein, da dieser aufbereitete Daten enthält und keinen gültigen JSON- oder XML-Code. Der angefragte Server wird diesen Text wahrscheinlich nicht interpretieren können. Nach einem PUT- oder einem POST-Request befindet sich eine Kopie des gesendeten Bodys im Body/Request-Tab zur eventuellen weiteren Bearbeitung und auch die gesendeten Request-Headers sind im Bereich Headers/Request ersichtlich. Der Header *Content-Length* wurde automatisch mit der Länge der angehangenen Body-Nachricht gesetzt. Wurde ein beliebiger Request durchgeführt und die Response erfolgreich auf JSON oder XML getestet und ist das Request-Fenster bislang noch leer, so wird dem Anwender an dieser Stelle ein einfaches JSON- bzw. ein XML-Template generiert, welches er anpassen kann. Der Anwender hat darüber hinaus im rechten Bereich die Möglichkeit, diesen um weitere Headers zu erweitern. An dieser Stelle kann man aus der angebotenen Liste einen gebräuchlichen General- oder einen Request-Header auswählen, man darf jedoch auch einen beliebiges Header/Werte-Paar eingeben. Zwei gleichnamige Headers sind hier verständlicherweise ebenso wenig erlaubt, wie ein leeres Schlüssel- oder Wertefeld. Wird eine Headerzeile markiert, so wird der neue Header direkt darüber eingefügt und die Markierung entfernt. Ist keine Headerzeile markiert, so wird das neue Tupel am Ende der Liste eingefügt. Die an dieser Stelle verwendete Datenstruktur sind die beiden *NSArray*s *headerKeysArray* sowie *headerValuesArray*, an deren Indexwert *x* sich der Eintrag der *x*-ten Reihe der einspaltigen Headertabelle befindet. Zwar wären die Zellen der *TableView* theoretisch ebenfalls in der Lage Daten zu speichern, jedoch kommt es dann zu folgendem unerwarteten Programmverhalten: Als Container dient eine *ScrollView*, darin befindet sich eine *TableView*. Das hat den Vorteil, dass es möglich ist, mehr Daten darzustellen, als sichtbar sind, da sich die Tabelle einfach mit dem Finger aus dem sichtbaren Bereich wischen lässt. Bewegt man nun jedoch eine Zelle aus dem sichtbaren Bereich, existiert aktuell keine sichtbare Referenz mehr darauf und das System setzt den Inhalt auf seine *Reusable*-Liste. Die Daten, die in der Zelle enthalten sind, werden entfernt. Kommt die Zelle wieder in den sichtbaren Bereich, so werden nicht die alten Daten wiederhergestellt, sondern an die *DataSource* der *TableViews* - ein instanziiertes Objekt der Klasse

HeaderKeysViewController - wird die Nachricht *tableView:cellForRowAtIndexPath* gesendet, woraufhin dieser eine neue Zelle erzeugt. Diese würde sich den Zelleninhalt jedoch aus den aktuellen Inhalten der Textfelder für den HTTP Header-Input besorgen, was nicht das ist, was an dieser Stelle passieren sollte. Da gemäß MVC allerdings unterhalb der GUI die Daten in *NSArray*s abgelegt sind, deren Index mit der Zellennummer identisch ist, kann die Wiederherstellung des Zelleninhaltes auf die entsprechenden Einträge der *NSArray*s zugreifen.

Implementierung der *tableView:cellForRowAtIndexPath*:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell =
    [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil)
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleValue1
                reuseIdentifier:CellIdentifier];
    NSString *cellText; if ([indexPath section] == 0) cellText =
    [[NSString alloc] initWithFormat:@"%@",
    [_generalHeaders objectAtIndex:indexPath row]];
    else cellText = [[NSString alloc] initWithFormat:@"%@",
    [_requestHeaders objectAtIndex:indexPath row]];
    [[cell.textLabel] setText:cellText];
    return cell;
}
```

Befindet sich der hinzugefügte Header in der Liste der bekannten General Headers, so wird er mit grüner Schrift angezeigt, ist er ein bekannter Request Header, erscheint seine Schrift in Blaugrün. Befindet sich der hinzugefügte Header weder in der General- noch in der Request-Header-Liste, so wird dessen Name rot gefärbt. Dies soll der Vermeidung von Tippfehlern dienen.

Benötigt die Seite eine *HTTP Basic*- oder eine *HTTP Digest*-Authentifizierung, erhält das View-Objekt eine *Authentication Challenge*. Die API stellt hierfür den Methodenaufruf *connection:didReceiveAuthenticationChallenge* bereit, in welcher sich dieser Aufforderung angenommen werden sollte. Sind Authentifizierungsinformationen gegeben, so werden diese nun ausgelesen und dem angefragten Server übermittelt. Die Implementierung dieser Methode sieht wie folgt aus:

```
- (void)connection:(NSURLConnection *)connection
didReceiveAuthenticationChallenge:
(NSURLAuthenticationChallenge *)challenge {
    if ([challenge.previousFailureCount] == 0) {
        NSURLConnectionCredential *credential =
        [NSURLCredential credentialWithUser:[_username text]
```

```
                password:[_password text]
                persistence:NSURLCredentialPersistenceNone];
        [[challenge sender] useCredential:credential
        forAuthenticationChallenge:challenge];
    } else {
        [[challenge sender] cancelAuthenticationChallenge:challenge];
        UIAlertView *alert =
        [[UIAlertView alloc] initWithTitle:@"Error"
            message:@"Authentication incorrect."
            delegate:self
            cancelButtonTitle:@"Close"
            otherButtonTitles:nil];
        [alert show];
        [challenge description]);
    }
}
```

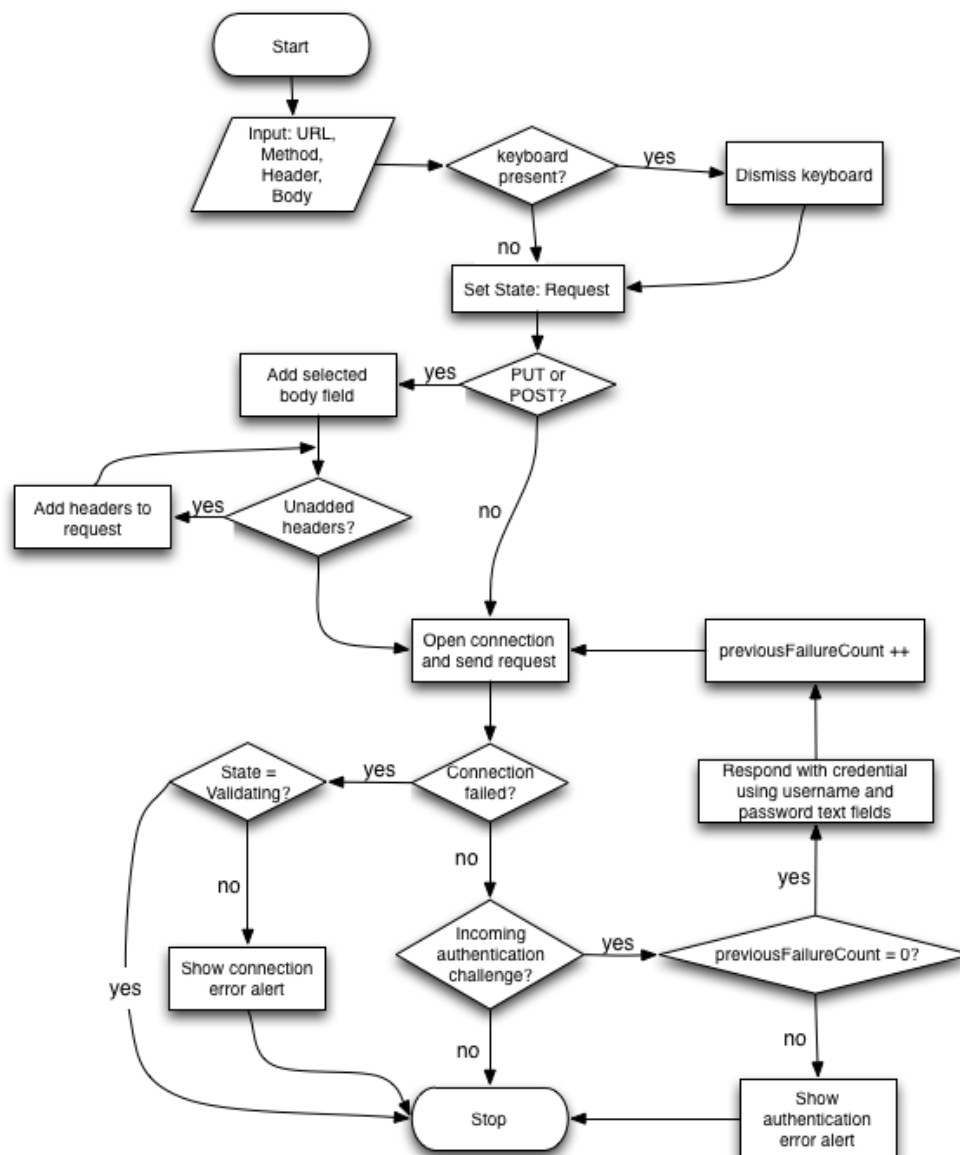


Abbildung 4.4: Senden eines Request

### 4.2.2 Empfangen einer Response

Jede eingehende Response ruft die Methode *connection:didReceiveResponse* auf. Hier werden die Informationstextfelder im oberen Bereich der View gesetzt, die Headers in den entsprechenden Bereich im Output-Feld übertragen und überprüft, ob der Inhaltstyp als JSON oder XML angegeben wird. Liegt der vom Server



zurückgelieferte Statuscode unter 400 deutet das darauf hin, dass es auf unseren Request hin keinen Fehler gab. Es wird nun ein neues Historyelement erzeugt, mit der erfolgreich bearbeiteten URL gefüllt und der History-Queue angehangen.

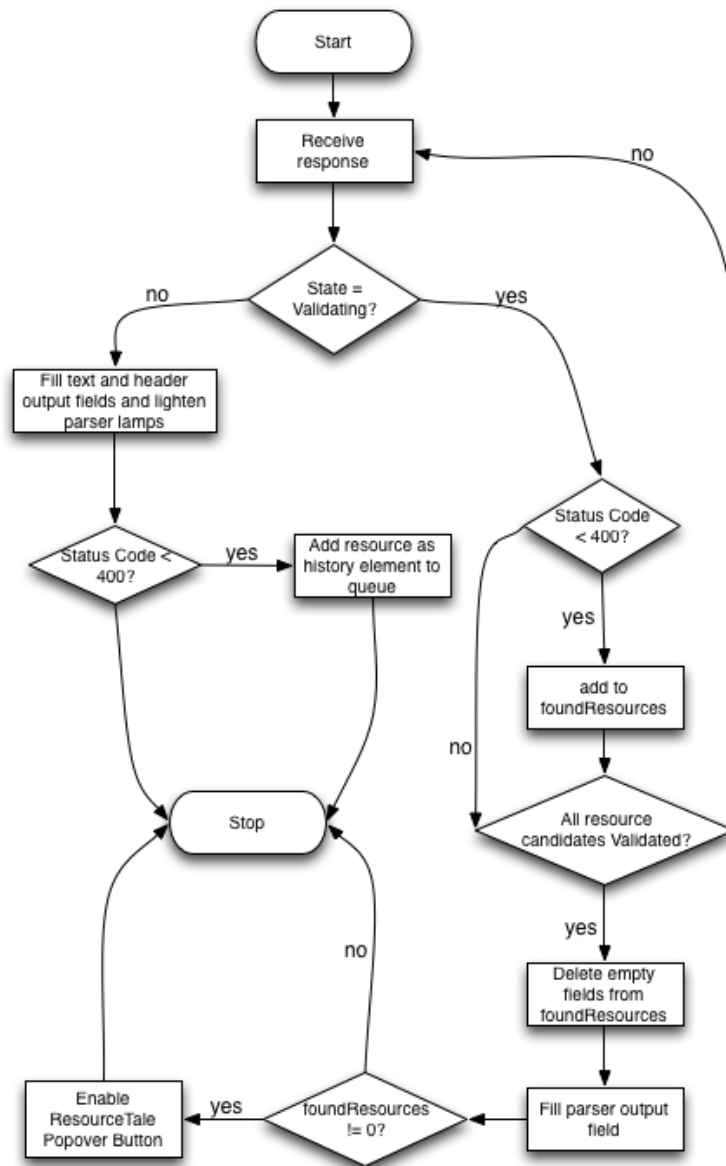


Abbildung 4.5: Eintreffen einer Response

Trifft eine Response ein, die neben dem Header einen Body enthält, wird mit diesem die Methode *connection:didReceiveData* aufgerufen. Für jede Teilresponse werden die Daten mittels `[responseBodyData appendData:_bodyData]` der statischen Variable *responseBodyData* hinzugefügt.

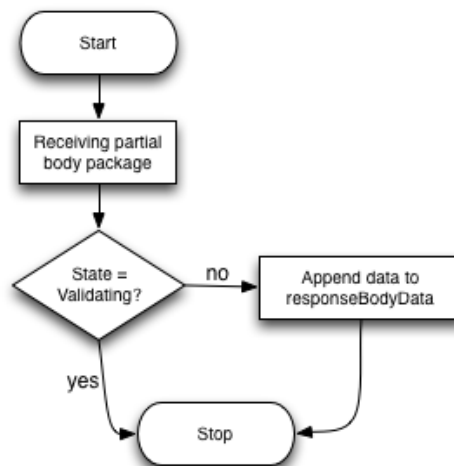


Abbildung 4.6: Eingehende Packages

Die eintreffenden Pakete brachten anfangs leider ein paar Probleme mit sich, denn jedes Datenpaket wurde stets gleich an den Parser übergeben. Als erster Ansatz wurde die *Content Length* mit der Länge des eingetroffenen Bodys verglichen und die Daten in einem statischen *NSString* angesammelt, bis die Nachricht vollständig empfangen wurde. Leider ist nicht bei jeder Response der *ContentLength*-Header gesetzt, wodurch die vollständige Größe nicht immer im Voraus bekannt ist. Als Workaround wurde mit einer Response ein *NSTimer* mit einer provisorischen Dauer von 3 Sekunden gestartet, welcher sich mit jeder eingehenden Nachricht wieder zurücksetzte. Lief dieser ab, so wurde das vollständige Empfangen der Nachricht angenommen und die Analyse der Bodydaten begonnen. Hier zeigte sich, dass ein gründlicherer Blick ins Referenzhandbuch einiges an Arbeit erspart hätte, denn wie für so vieles stellt Apple durchdachte automatische Methodenaufrufe bereit und einige Zeit später stieß ich auf die Methode *connectionDidFinishLoading*. Da dies natürlich ein wesentlich feinerer Ansatz ist als der Timer, wurde die Implementierung entsprechend abgeändert.

Anhand des ersten Bytes des Bodys wird überprüft, ob es sich um ein Bild handelt. Ist das der Fall, so wird in die *ScrollView* programmatisch eine *ImageView* eingebettet und das Bild darin angezeigt. Beim Absenden eines neuen Requests werden alle *ImageViews* wieder aus der *ScrollView* entfernt. Weist das erste Byte

nicht auf ein Bild hin und wurde bereits vorher anhand des *Content Type* erkannt, dass es sich um JSON- oder um XML-Code handelt, wird der Body an den entsprechenden Parser übergeben. Im Apple Foundation-Framework findet sich bereits seit OS X-Version 10.3 der *NSXMLParser*, seit iOS 5 wurde dieses um die *NSJSONSerialization* erweitert, welche ein *NSData*-Objekt parst und ein *NSDictionary* zurückliefert. In beiden Fällen werden sowohl die Schlüssel als auch die Werte in je ein veränderliches *NSMutableArray* geschrieben, da es nötig sein wird, gezielt Felder zu löschen und neue hinzuzufügen. Wurde das Parsen erfolgreich abgeschlossen, wird die rekursive Methode *processKeys* aufgerufen. *Erfolgreich* bedeutet im Fall des XML-Parsers, dass er nach erfolgter Durchführung ein *YES* zurückliefert. *NSJSONSerialization* definiert einen Durchlauf als *erfolgreich*, wenn das zu füllende Array nach Durchlauf des Parsers nicht leer ist. Die einzelnen Schlüssel/Werte-Paare werden daraufhin überprüft, ob es sich bei den Werten um eine Ressource handelt. Da es durchaus sein kann, dass ein Wert wiederum ein komplettes Array mit weiteren Schlüssel/Werte-Paaren beinhaltet, ist *processKeys* so gestaltet, dass ein rekursiver Aufruf möglich ist. Da hierbei Schlüssel/Werte-Paare eingefügt und gelöscht werden, ist das Mitführen des statischen Indexes *staticIndex* erforderlich. Eingefügt werden Schlüssel/Werte-Paare, wenn ein Unter-Array gefunden wird und deren Inhalt dem eigenen Array am aktuellen Index eingefügt wird. Das Schlüssel/Werte-Paar, welches das Array enthielt, hat nach dem rekursiven Abarbeiten keine weitere Funktion mehr und es wird entfernt, denn die darin enthaltenen Schlüssel/Werte-Paare wurden bereits an dessen Stelle eingefügt. Das Array wächst und die enthaltenen Schlüssel/Werte-Paare sind noch in der richtigen Reihenfolge.

Zum Vorbereiten des Validitätstests auf eine valide URL wird der String in einem ersten Schritt in den Typ *NSURL* umgewandelt. Hierbei wird bereits intern getestet, ob die Zeichen für eine URL valid sind. Verläuft dieser Test negativ, wird ein *nil* zurückliefert. Wird kein *nil* zurückgegeben, wäre folglich der Einsatz als URL denkbar und der Anfang wird überprüft. Beginnt dieser mit *http*, wird er unverändert zum Validieren weitergeleitet. Beginnt er statt dessen mit einem Schrägstrich, wird er an die Basis-URL des zuletzt aufgerufenen Requests angefügt und dann ebenfalls zur Validitätsüberprüfung weitergereicht. Treffen beide Fälle nicht zu, wird ein Element auf der gleichen Verzeichnisebene vermutet, auf der diese Response basiert. Aus der Request, die hierher führte, wird die höchste Verzeichnisebene (der Teil bis inklusive dem letzten Schrägstrich) abgefragt, um diesen String erweitert und wie zuvor zur Validierung weitergegeben.

Um den Test durchzuführen, wechselt die App mit `[self setValidatingState:YES]` in einen Zustand, welcher ausschließlich der Validierung von Ressourcen dient. Es wird für jeden Ressourcenkandidaten einen asynchronen HEAD-Request abgesendet, deren Responses unter einem anderen Gesichtspunkt betrachtet werden. Ein

nebenbei laufender Counter führt Buch über den aktuellen Fortschritt und wird inkrementiert, sobald eine Ressource als valid oder als invalid befunden wird. Die endgültige Entscheidung über die Validität kann jedoch erst in der Methode *didReceiveResponse* getroffen werden, welche die Antwort des Servers auswerten darf. Da die asynchronen Requests die Responses in willkürlicher Reihenfolge eintreffen lassen, kann leider nicht ohne Weiteres zugeordnet werden, für welchen Wert gerade eine Antwort empfangen wurde. Die ID der abgesendeten *NSURLConnection* wird daher zusammen mit dem Index des zu überprüfenden Wertes in ein *NSDictionary* gepackt. In der *didReceiveResponse* kann anhand dieser ID der Indexwert aus dem Wörterbuch abgefragt werden und es ist wieder klar, welchem Schlüssel/Werte-Paar der empfangene Statuscode zuzuordnen ist. Ein Statuscode kleiner 400 ist auch hier positiv zu interpretieren und weist auf die Existenz der angefragten Resource hin. Der angefragte Wert wird in die *NSMutableArray*s *foundResourceKeys* sowie *foundResourceValues* geschrieben. Da ein asynchroner Request den Nachteil hat, dass die überprüften Ressourcen in beliebiger Reihenfolge im Zielarray landen würden, eine synchrone Überprüfung jedoch zu viel Zeit in Anspruch nehmen würde, wird wie folgt vorgegangen: Die Arrays *foundResourceKeys* und *foundResourceValues* werden mit der Größe der Ressourcenkandidaten angelegt und die Felder mit einem leeren *NSString* initialisiert. Eine überprüfte Ressource, die sich im Quellarray an einem Feld mit dem Index y befindet, wird in das Zielarray ebenfalls an das Feld mit dem Index y geschrieben. Alle Felder der beiden Arrays, an die keine Einträge geschrieben wurden, enthalten weiterhin leere Strings.

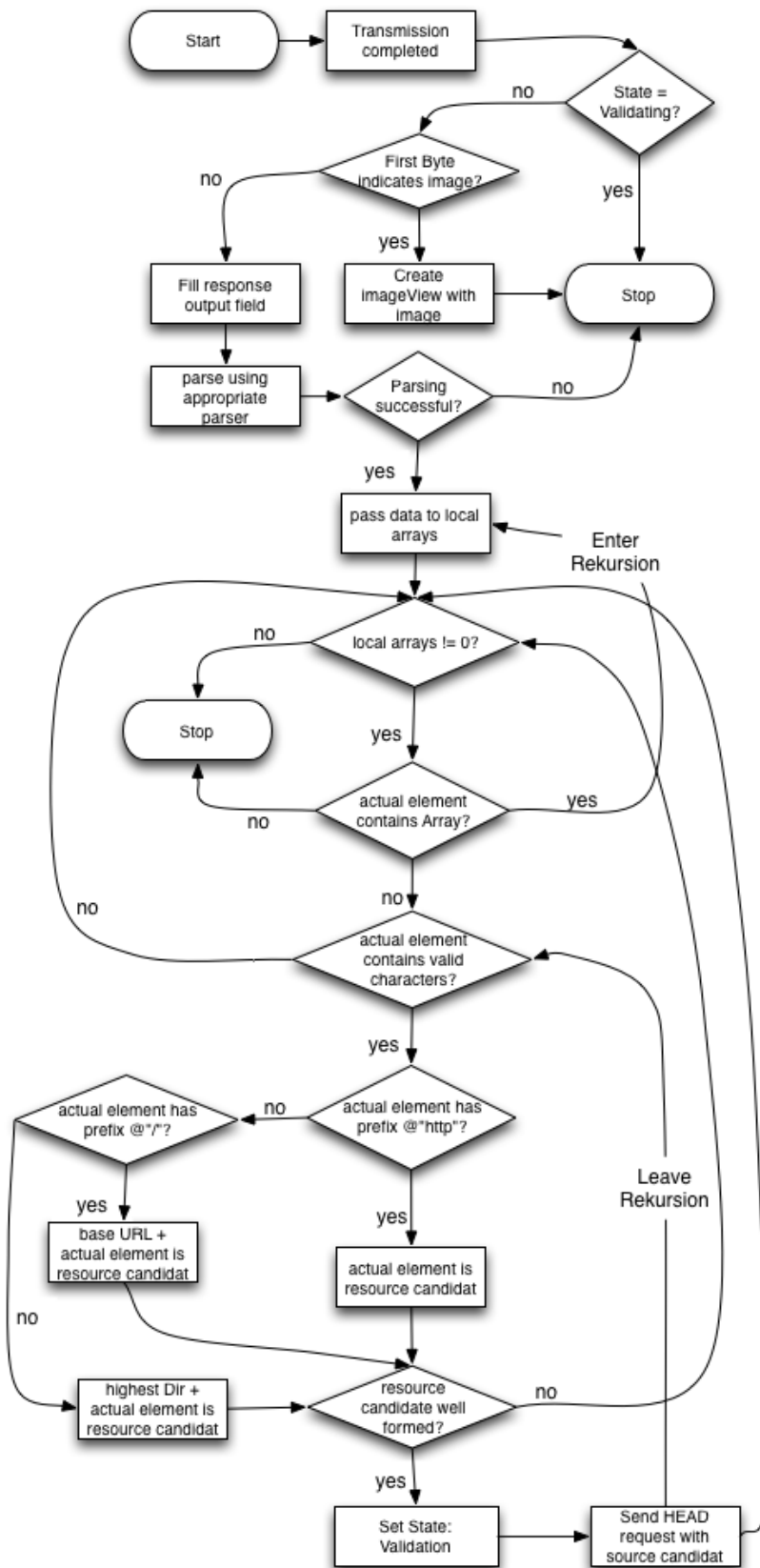


Abbildung 4.7: Response wurde vollständig empfangen

Nachdem keine Ressourcenkandidaten mehr auf ihre Validierung warten, lassen sich die *NSMutableArray*s, welche über die validen Ressourcen verfügen, bereinigen, in dem die Felder mit den leeren Strings kurzerhand aus dem Array entfernt werden. Zurück bleiben je ein Array mit Schlüsseln und eines mit den dazugehörigen Werten. Beide werden beim Berühren der Ressourcenschaltfläche - der Lupe - an die gerade erzeugte Instanz der Klasse *ResourceTableViewController* übergeben. Was hier fast beiläufig klingt war, zumindest zeittechnisch - alles andere als beiläufig, da die sich öffnende TableView nach ihrer Instanziierung und Überreichung der Daten dennoch stets jungfräulich präsentierte. Die *NSMutableArray*s schienen dort nie anzukommen. Wie sich herausstellte, erzeugt bereits das Storyboard nach dem Berühren der Übergangs-Schaltfläche eine Instanz der gewünschten View. Die programmatisch erzeugte Instanz war eine weitere, sich an einem anderen Speicherbereich befindliche Instanz, die unsichtbar im Hintergrund abläuft. Hier soll jedoch auf die Storyboard-Instanz zugegriffen werden. Um das zu bewerkstelligen, müssen die Daten an diese übergeben werden, nachdem der Übergang in die neue View eingeleitet wurde. Dies muss allerdings auch geschehen, bevor die View auf dem Bildschirm erscheint, was ja scheinbar zeitgleich ist. Um auf dieses Zeitfenster zuzugreifen, lässt sich auf diese Übergänge, die Xcode als *Segues* bezeichnet, programmatisch zugreifen. Bei einem Übergang erhält die Ursprungs-View die Nachricht *prepareForSegue:sender*. Um gezielt Übergänge ansprechen zu können, lassen sich im Storyboard *Identifier* vergeben. Der Identifier des gerade aktiven Überganges kann in der Implementierung der Methode *prepareForSegue:sender* mit `[segue identifier]` abgefragt werden.

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"resourcesTableViewPopover"])
    {
        ResourcesTableViewController *resourceTableViewController =
            [segue destinationViewController];
        [resourceTableViewController setKeys:foundResourceKeys];
        [resourceTableViewController setValues:foundResourceValues];
        [resourceTableViewController setReferenceToUrl:_url];
        UIStoryboardPopoverSegue* popoverSegue =
            (UIStoryboardPopoverSegue*)segue;
        [resourceTableViewController setReferenceToPopoverController:
            [popoverSegue popoverController]];
        [resourceTableViewController setReferenceToBaseUrl:
            [self urlPart:_url text] definePart:@"baseUrl"]];
        [resourceTableViewController setReferenceToHighestDir:
            [self urlPart:_url text] definePart:@"highestDir"]];
    }
    else if ([[segue identifier] isEqualToString:@"logOutputViewPopover"])
    {
        LogOutputViewController *logOutputViewController =
```

```
[segue destinationViewController];
...
```

Ein weiteres Problem war die Tatsache, dass offensichtlich zwischen iOS 5.1 und 6 das Laden der View an unterschiedlichen Zeitpunkten erfolgt. Unter iOS 6 ist es ausreichend, wenn die Initialisierung des übergebenen Arrays in der Startmethode *viewDidLoad* erfolgt. Doch während dort bei einem *prepareForSegue* der Aufbau der View verzögert wird, bis die Methode abgearbeitet wurde, ist dies bei iOS 5.1 nicht der Fall. Die Initialisierung des übergebenen Arrays darf erst in der Initialisierungsmethode *viewDidAppear* erfolgen, sonst werden alle Zellen mit *nil* gefüllt. Da dieses Programm zu iOS 5.1 vollständig abwärtskompatibel sein soll, erfolgt die Initialisierung in der Methode *viewDidAppear*.

## 4.3 Hilfsklassen

### 4.3.1 ResourceTableViewController

Die erste Hilfsklasse ist eine View vom Typ *UITableViewController*, einer Unterklasse von *UIViewController*. Der *TableViewController* bietet eine View, bestehend aus einer einzigen Tabelle, deren *Delegate* und *DataSource* er selbst ist. Ihm stehen die *NSArray*s *keys* und *values* zur Verfügung, welche die validierten Ressourcen darstellen, die er von der Hauptview unmittelbar vor dem Aufbau erhalten hat und die er in die Zellen seiner Tabelle einfüllt. Da eine Berührung einer Zelle die Rückkehr zur Hauptview zur Folge haben soll, nachdem die Ressource in das dortige URL-Feld eingefügt wurde, benötigt er für seine Funktion einige Referenzen. Verständlicherweise wird die Referenz des URL-Feldes benötigt, auf deren Text er schreibend zugreift. Referenzen zu den *NSString*s *highestDir* und *baseUrl* benötigt er, um die URLs zu vervollständigen. Eine Ressource 1337, welche auf <http://www.smartphoneseppe.de/customer/> gefunden wurde, sollte auf <http://www.smartphoneseppe.de/customer/1337> verweisen. Die Ressource wird an den höchsten Verzeichnispfad der Abfrage gehangen, welche die Ressource 1337 enthielt. Eine mit einem Schrägstrich beginnende Ressource */article/galaxy\_s3* wird hingegen als absoluter Pfad interpretiert und direkt an die Base-URL angehängen: [http://www.smartphoneseppe.de/article/galaxy\\_s3](http://www.smartphoneseppe.de/article/galaxy_s3).

Eine View kann sich nicht so einfach selbst wieder schließen. Um das dennoch zu realisieren wurde darüber hinaus die Referenz des *PopoverControllers* hierher übergeben. Der Zugriff auf diesen ermöglicht es, die View mit dem Befehl `[_referenceToPopoverController dismissPopoverAnimated:YES]` wieder sozusagen “von unten” zu verwerfen.

### 4.3.2 HeaderKeysViewController

Wie auch bei der *ResourceTableViewController* handelt es sich hier um eine Klasse des Typs *UITableViewController*. Er beinhaltet eine Auflistung der für einen Request gebräuchlichen Request-Headers und der General-Headers und soll dem Anwender eine einfache Möglichkeit bieten, seine Anfrage um valide Headers zu erweitern. Wie zuvor werden hier die Daten der durch Berührung selektierten Zelle in das Header-Feld der Hauptview geschrieben und die View mittels Referenzzugriff auf den darunter liegenden *PopoverController* beendet. Als kleines “Extra” wird die Referenz für das dem Header zugehörige Wertefeld übergeben, mit dessen Hilfe dieses zum *FirstResponder* wird - nach dem Verwerfen der View hat es somit den Tastaturfokus und es kann sofort nach dem Selektieren des Headers mit der Eingabe des Wertes begonnen werden.

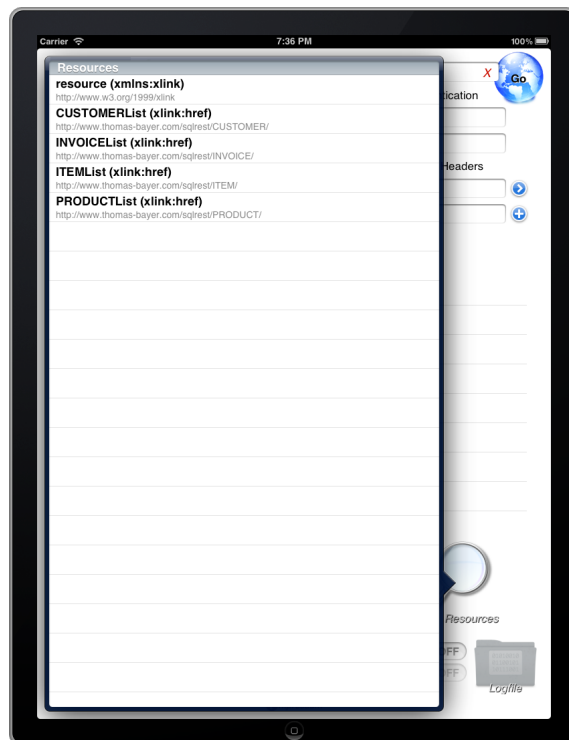


Abbildung 4.8: Erkannte Ressourcen

### 4.3.3 XMLParser

Der Inhalt der Klasse *XMLParser* ist nicht umfangreich, eignete sich jedoch hervorragend, um als abgeschlossene Funktionalität in einer eigenen Klasse ausgliedert zu werden. Er wird initialisiert mit zwei leeren *NSMutableArray*s, an die sowohl Attribute als auch Elemente der geparsen XML angefügt werden. Attributnamen werden in Klammern gesetzt und erhalten den Elementnamen, dem sie zugehörig sind, vorangestellt. Die Bezeichnung der Keys setzt sich als Pseudocode vereinfacht wie folgt zusammen:

```
for (int i = 0; i < elementsToParse; i++) do
    if actualObjectType = attribut then
        keyArray[i] ← elementName + “(“ + attributName + “)”;
        valueArray[i] ← AttributValue;
    else if actualObjectType = element then
        keyArray[i] ← elementName;
        valueArray[i] ← elementValue;
```



```

        fi;
    od;

```

#### 4.3.4 LogOutputViewController

Da es sich um ein Entwicklertool handelt, wäre es für den ein oder anderen sicherlich interessant zu sehen, was das Programm in der Kommandozeile an Log-Ausgabe erzeugen würde. Diese *PopoverView* stellt die Ausgabe von *NSLog* dar, welche der normale App-Anwender eigentlich gar nicht zu sehen bekäme und leitet sie in eine Datei um. Um nicht in jeder Zeile das Datum, die Uhrzeit und den Programmnamen aufzuführen, wurden diese zu Beginn der Hauptklasse aus der Ausgabe von *NSLog* herausgeschnitten, indem der reine Ausgabeteil als String in UTF8-Codierung mittels *fprintf* ausgegeben wird:

```

#ifdef DEBUG
#define NSLog(FORMAT, ...) fprintf(stderr,"%s\n",
[[NSString stringWithFormat:FORMAT, ##__VA_ARGS__] UTF8String]);
#endif

```

Weiterhin wird gleich zum Programmstart in der Methode *viewDidLoad* die Ausgabe in eine Datei umgeleitet. Hierbei wird zuerst geschaut, ob bereits eine Ausgabedatei existiert. Ist das der Fall, so wird die Logging-Funktion automatisch eingeschaltet; ist sie nicht vorhanden, bleibt sie deaktiviert. Deaktiviert man sie während der Laufzeit, so wird die Datei ebenfalls entfernt. Somit findet man den Logging-Schalter bei einem App-Restart stets so vor, wie man ihn verlassen hat.

```

NSArray *paths = NSSearchPathForDirectoriesInDomains
(NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
logPath = [documentsDirectory stringByAppendingPathComponent:@"console.log"];
NSFileManager *filemgr;
filemgr = [NSFileManager defaultManager];
if ([filemgr fileExistsAtPath:logPath]) {
    freopen([logPath cStringUsingEncoding:NSUTF8StringEncoding], "a+", stderr);
    [_logToFileSwitch setOn:YES];
    [_logFileButton setEnabled:YES];
    [_verboseLogLabel setEnabled:YES];
    [_verboseLogSwitch setEnabled:YES];
} else
    NSLog(@"No existing log file found, logging disabled.");

```

Mit dem Papierkorb unter dem Ausgabe-Fenster lässt sich die Datei nach dem Sichten aufräumen und das Logging dabei eingeschaltet lassen. An diesem Icon wurde keine Bildunterschrift angebracht, denn es sollte selbsterklärend sein, was hier passiert. Werden informationsreichere Mitschnitte gewünscht, lässt sich dieser

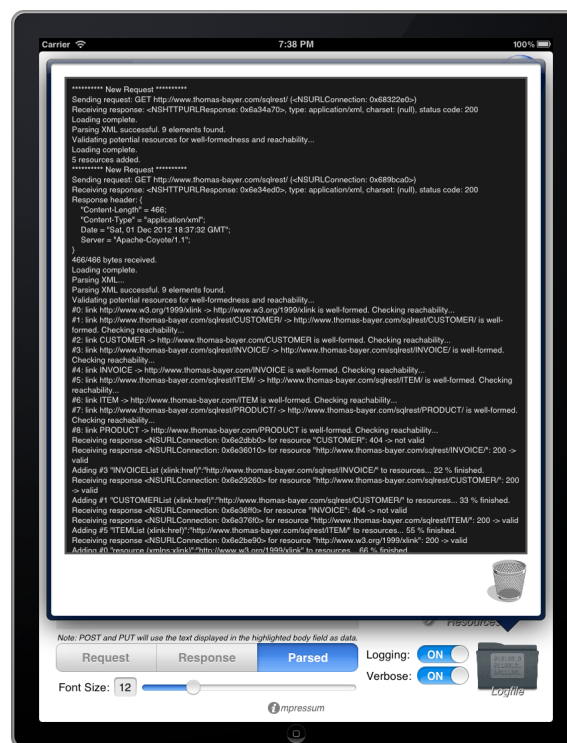


Abbildung 4.9: Logging-Output im Verbose-Modus

Wunsch über den Verbose-Schalter mitteilen. Es werden dann mehr Informationen mitgeschritten. Auch hier ist ein unerwartetes Problem aufgetreten: Da die View nichts beinhaltet, was eine *Delegate* oder eine *DataSource* benötigt, wollte ich diese recht codezeilenarme Funktionalität in der Hauptview unterbringen. Jedoch war nach dem Öffnen der View das Ausgabefeld stets leer, so dass ein Refresh-Button betätigt werden musste, der die Datei erneut in das Textfeld mappte. Erst dann wurde der Inhalt sichtbar. Abhilfe brachte das Auslagern in eine eigene Klasse mit der Initialisierungsmethode *viewDidAppear*, welche nun das Laden der Datei übernimmt.

### 4.3.5 HistoryElement

*HistoryElement* ist die Implementierung der URL-History in Form einer doppelt verketteten Liste. Es kennt seinen Vorgänger, seinen Nachfolger als auch die an ihn gebundene Ressource. *HistoryElement* verfügt über eine Methode *getPart*, die auf Wunsch die Basis-URL - bestehend aus Protokoll und Domain -, den höchsten Verzeichnispfad oder den vorherigen zurückliefert. Der Unterschied zwischen vorherigem und höchstem Verzeichnis ist folgender: Endet eine URL mit einem

Schrägstrich, so wird er als Verzeichnis betrachtet und ist gleichzeitig das höchste Verzeichnis. Endet er nicht mit einem Schrägstrich, wird eine Datei vermutet und statt dessen das letzte Verzeichnis zurückgegeben, das mit einem Schrägstrich endet.

```
Beispiel 1: @"http://www.smartphoneseppe.de/customer/1337"
highestDir = previousDir = @"http://www.smartphoneseppe.de/customer/"
Beispiel 2: @"http://www.smartphoneseppe.de/shoppingcart/1337/"
highestDir = @"http://www.smartphoneseppe.de/shoppingcart/1337/"
previousDir = @"http://www.smartphoneseppe.de/shoppingcart/"
```

#### 4.3.6 ... und die AppDelegate?

Erstellt man ein neues Projekt, befindet sich im Projektordner neben den Controllern eine weitere Klasse: die *AppDelegate*. Konzeptuell sollte der *ViewController* alles beinhalten, womit sich die aktuelle View beschäftigt. Die *AppDelegate* bleibt dabei erschreckend leer und hinterlässt das Gefühl, scheinbar etwas Wichtiges vergessen zu haben. Die einzige Methode, die überhaupt etwas macht ist die application *didFinishLaunchingWithOptions:launchOptions*. Sie gibt ein *YES* zurück. Immerhin.

Auch hier läuft wieder alles über das Storyboard. Wirft man einen Blick auf die Datei Projektname-Info.plist findet sich ein Schlüssel mit der Bezeichnung *Main storyboard file base name* und dem Wert *MainStoryboard*. Die Bedeutung hinter diesem Eintrag ist die Folgende: Ist diese Einstellung gesetzt, lädt *UIApplication* die Datei mit der Bezeichnung *MainStoryboard.storyboard*, instanziiert den ersten *ViewController* und packt dessen View in ein neues Objekt des Typs *UIWindow*. Es ist also völlig in Ordnung, wenn diese Datei bei Benutzung von Storyboards leer bleibt.

## Kapitel 5

### Schlusswort und Ausblick

Es ist vollbracht - die erste eigene Anwendung, geschrieben in der auf den ersten Blick exotisch anmutenden Programmiersprache mit den vielen eckigen Klammern, läuft auf dem marktbeherrschenden Tablet-Computer der Firma Apple Inc. In einem abschließenden Fazit werde ich die einzelnen Schritte auf dem Weg hierher als auch den persönlichen Nutzen dieser Arbeit für mich reflektiv umreißen und auf die sich für mich ergebende Bereitschaft eingehen, weiterhin in dieser Umgebung zu entwickeln.

Objective-C ist anders. Allerdings nicht so anders, dass man sich nicht mit ein wenig Zeit daran gewöhnen kann. Und möchte. Die Namenskonvention bei Objective-C ist gewöhnungsbedürftig, die Verwendung von scheinbar möglichst langen Variablennamen oder einer redundanten zusätzlichen Beschreibung für jeden Parameter in einer Methode ist zu Beginn äußerst gewöhnungsbedürftig, steigert aber die Lesbarkeit des Codes ungemein. Auch an die Notation mit den eckigen Klammern gewöhnt man sich ziemlich schnell - wenngleich das auch nicht wirklich nötig ist, denn die Klammern lassen sich in den meisten Fällen mit der alternativ angebotenen Punkt-Schreibweise umgehen, die für die meisten Umsteiger sehr vertraut wirkt. Ich selbst habe mich nach einiger Eingewöhnungszeit recht schnell schon während meines ersten Programmes auf diese Notation umgestellt; sie führte mir automatisch das Konzept der Nachrichtenkommunikation vor Augen. Die Cocoa-API erleichtert viele programmatische Handgriffe und es lohnt sich stets, einen Blick in die Referenz zu werfen. Beispielsweise habe ich zuerst anhand der Content-Länge überprüft, wann eine Response vollständig eingegangen ist. Ein sehr einfacher Weg, leider setzt er voraus, dass die Content-Länge in der Response angegeben ist. Da dies jedoch nicht immer der Fall ist, implementierte ich einen Workaround, der eine eingehende Response einen *NSTimer* starten lies, welcher sich bei jedem eingehenden Package wieder zurücksetzte. So sollte der Client herausfinden, ab wann eine Nachricht unbekannter Länge als *vollständig empfangen* anzusehen ist. Ein sorgfältiger Blick in die Referenz hätte mir diesen Umstand abgenommen, denn dort wäre ich auf die Methode *connectionDidFinishLoading* gestoßen, welche eine elegantere Lösung zu genau dieser Problematik bietet, denn deren Implementierung wird nach dem Erhalt des finalisierenden Packages aufgerufen.

Auch meine Erfahrungen mit Xcode waren fast durchgängig von sehr angenehmer Natur. Unter Apples Entwicklungsumgebung lässt es sich sehr angenehm arbeiten. Die Live-Prüfung, die eine Autokorrektur für syntaktische Fehler und Tippfehler anbietet und auch der Assistenzcontroller, der zur geöffneten Datei immer gleich passende Sekundärdateien anbietet, sind Dinge, die ich in künftigen Entwicklungsumgebungen nur ungern missen werde. Ein kleiner Wermutstropfen ist, dass Xcode nach einiger Zeit kontinuierlich immer langsamer wird. Das geht so weit, dass es eine Qual ist, den aktiven Quellcode-Tab zu wechseln, da man im

schlimmsten Fall lange auf das drehende Rad schaut, in welches sich der Mauszeiger verwandelt, um den ausgelasteten Zustand der Anwendung zu signalisieren. Ein wenig Foren-Recherche ergibt schnell, dass der Schuldige die sich ansammelnden Workspace-Einstellungen sind. Lässt man sich die Inhalte des Packages *Projectname.xcodeproj* anzeigen, so findet man dort eine Datei mit der Bezeichnung *project.xcworkspace*. In dieser Datei speichert Xcode die Einstellungen ab. Nach einem Löschen dieser Datei und Neustart von Xcode sind zwar alle Workspace-Einstellungen verschwunden, Xcode im Gegenzug aber um ein Vielfaches performanter. Ein Opfer, dass man an dieser Stelle gerne zu zahlen bereit ist. Es könnte natürlich sein, dass es sich um einen versionsspezifischen Bug handelt. In der aktuellen Version 4.5.2 habe ich dieses Verhalten noch nicht festgestellt, allerdings war beim Update auf diese Version die Implementierung des REST Analyzer nahezu vollständig.

Das Storyboard ist eine ungemeine Arbeitserleichterung, wenn es um die Gestaltung der GUI geht. Hier findet der erste Kontakt mit Xcode statt, die ersten Zeilen Code, die bereits ein vollständiges GUI-Grundgerüst bilden, werden hier generiert. Es fühlt sich sehr einfach an und schafft es, Berührungsängste schnell zu nehmen. Mit wenig Zeitaufwand sind Views und Übergänge erstellt, GUI-Elemente eingefügt, gegebenenfalls deren *Delegates* und *DataSources* zugewiesen und in der Seitenleiste die Eigenschaften gesetzt, welche die Elemente bei Anwendungsstart besitzen sollten. Automatisch werden die Bereiche im Code eingefügt und mit den Storyboard-Elementen verbunden und man beginnt kurze Zeit später bereits mit der eigentlichen Implementierung der Funktionalitäten, ohne das Gefühl zu haben, sich vorher an notwendigen Nebensächlichkeiten aufhalten zu müssen. Selbstverständlich lässt sich auch alles ohne Storyboards entwerfen. Durch das Storyboard bereit gestellte Funktionalität sollte man jedoch nach Möglichkeit nicht mit dem programmatischen Weg vermischen. Es hat mich viel Zeit und Recherche gekostet, festzustellen, dass eine im Storyboard erstellte View nicht ohne Weiteres programmatisch zugreifbar ist. Der Weg, einem Übergang eine eindeutige Bezeichnung zu verpassen und auf diese über die in einem solchen Fall automatisch aufgerufene Methode *prepareForSegue* zuzugreifen, wirkte auf mich ein wenig wie ein Workaround, ist aber wohl der Weg, der in einem solchen Fall zu gehen ist. Um der Übersichtlichkeit zu genügen bietet das Storyboard 4 Zoomstufen an: 12,5 %, 25 %, 50 % und 100 %. Warum an dieser Stelle keine manuelle Eingabe der Zoomstufe gegeben ist oder zumindest ein Schieberegler angeboten wird, der sich etwas feingranularer einstellen lässt, ist mir nicht ganz ersichtlich. Bei meiner nicht ungewöhnlichen Auflösung von 1920x1200 wäre oft 75 % die Zoomstufe meiner Wahl gewesen.

In der Wahl der Implementierungsthematik fand ich mit RESTful Webservices ein interessantes Umsetzungsgebiet. Die Umsetzung eines generischen Clients für

verschiedene Webservices, deren Schnittstelle die Methoden bilden, die das HTTP-Protokoll schon selbst mitbringt, war interessant und erweiterte meine Kenntnisse der Webservices. Die diesem Standard zugrunde liegende Idee ist einfach und leicht zu implementieren und das Arbeiten mit HTTP-Verben fühlt sich “schon irgendwie nahe am Internet” an. In einem ersten Implementierungsansatz verwendete ich das externes Framework *RestKit*, welches ich dafür andachte, mir die Arbeit mit REST-konformen Seiten zu erleichtern. Es stellte sich jedoch sehr schnell heraus, dass einige der Funktionen meine Erwartungen für einen generischen Ansatz nicht erfüllen konnten. So bot mir *RestKit* das Mappen einer JSON-Ressource in ein *NSDictionary* an, welches jedoch bereits über die Schlüssel verfügen musste, zu welchen anschließend die Werte hinzugefügt werden. Für einen generischen Clienten, der jedoch noch nicht weiß, welche Schlüssel die Zielressource liefern wird, ist das leider nur von eingeschränktem Nutzen und die Zuhilfenahme eines Parsers ist unvermeidlich. Apple bietet in seinem hauseigenen Framework sowohl einen Parser für XML als auch einen Parser für JSON an, welcher bei vergleichbarer Funktionalität bereits wegen der offiziellen Unterstützung durch Apple bevorzugt eingesetzt werden sollte. Auch andere Teile, die das externe Framework bereitstellen würde, waren ohne größere Schwierigkeiten durch ein breites Arsenal hilfreicher Funktionen zu realisieren, die Cocoa Touch ohnehin bereitstellt. Letztendlich erlaubte dies mir die Umsetzung dieser Arbeit, ohne dass ich auf externe Frameworks zurückgreifen musste.

Abschließend kann ich über die vergangenen Wochen, in denen ich mich mit dieser Arbeit beschäftigt habe, Folgendes sagen: Mein “Blick über den Tellerrand” in Richtung Objective-C fühlt sich befriedigend an. Der Exkurs wird hier noch nicht enden, denn ich möchte noch einige weitere Anwendungen für Apple-Hardware entwickeln. Die Arbeit auf und mit Apples Plattform hat sich für mich als sehr angenehm erwiesen und meinen Programmierstil positiv beeinflusst. Kurze und unverständliche Variablennamen werde ich zugunsten der Lesbarkeit künftig vermeiden, ebenso werde ich beim Entwickeln fernab Objective-C die zusätzlichen Parameter-Bezeichner in den Methoden vermissen. Einen ersten Kundenauftrag für eine iPhone-App habe ich bereits schon vor dem Abschluss dieser Arbeit angenommen. Xcode wird also nicht lange geschlossen bleiben und schon bald werde ich meinen Ausflug in die Objective-C-Welt an der Stelle weiterführen, wo ich ihn mit dem Abschluss dieser Arbeit unterbrochen habe.

# Literaturverzeichnis

- [1] Fielding, Roy (2000) Dissertation: Architectural Styles and the Design of Network-based Software Architectures
- [2] Bayer, Thomas (2002): REST Webservices: Eine Einführung, URL: <http://www.oio.de/public/xml/rest-webservices.htm> (letzter Aufruf am 6.9.2012)
- [3] Tilkov, Stefan (2009): REST - Der bessere Webservice? In: JAXenter, URL: <http://it-republik.de/jaxenter/artikel/REST—Der-bessere-Web-Service-2158.html> (letzter Aufruf am 6.9.2012)
- [4] Freeman, Jay: How to Host a Cydia™ Repository, URL: <http://www.saurik.com/id/7> (letzter Aufruf am 16.11.2012)
- [5] Wikipedia: Web service, URL: [http://en.wikipedia.org/wiki/Web\\_service](http://en.wikipedia.org/wiki/Web_service) (letzter Aufruf am 5.9.2012)
- [6] [http://de.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://de.wikipedia.org/wiki/Representational_State_Transfer) (letzter Aufruf am 5.9.2012)
- [7] Introducing JSON, URL: <http://json.org/index.html> (letzter Aufruf am 15.09.2012)
- [8] XML ESSENTIALS, URL: <http://www.w3.org/standards/xml/core> (letzter Aufruf am 16.09.2012)
- [9] <https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html> (letzter Aufruf am 10.10.2012)
- [10] [http://developer.apple.com/library/ios/#referencelibrary/GettingStarted/Learning\\_Objective-C\\_A\\_Primer/\\_index.html](http://developer.apple.com/library/ios/#referencelibrary/GettingStarted/Learning_Objective-C_A_Primer/_index.html) (letzter Aufruf am 10.10.2012)



- [11] Rodewig, Klaus M. / Wagner, Clemens: Apps entwickeln für iPhone und iPad - Das Praxisbuch, Galileo Computing, 2012, URL: [http://openbook.galileocomputing.de/apps\\_entwickeln\\_fuer\\_iphone\\_und\\_ipad/index.html](http://openbook.galileocomputing.de/apps_entwickeln_fuer_iphone_und_ipad/index.html) (letzter Aufruf am 12.11.2012)