

Bachelorthesis

# **Verlässliche mobile Anwendungen**

## **Untersuchungen am Beispiel einer Fitness-App**

Am IT-Center Dortmund GmbH  
Studiengang IT- und Softwaresysteme  
erstellte Bachelorthesis  
zur Erlangung des akademischen Grades  
Bachelor in Information Technology

von

Kevin Schie / Stefan Suermann  
geb. am 04.07.1993 / 13.12.1987  
Matr.-Nr. 2012013 / 2012027

Betreuer:

Prof. Dr. Johannes Ecke-Schüth  
Prof. Dr. Klaus-Dieter Krägeloh

Dortmund, 15. September 2015



# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>1</b>
<b>Eidesstattliche Erklärung</b>	<b>2</b>
<b>Aufgabenstellung</b>	<b>5</b>
<b>1. Einleitung</b>	<b>7</b>
1.1. Problemstellung . . . . .	7
1.2. Zielsetzung . . . . .	8
1.3. Vorgehensweise . . . . .	8
1.4. Arbeitsaufteilung . . . . .	9
<b>2. Problemanalyse</b>	<b>11</b>
2.1. Problembeschreibung . . . . .	11
2.2. Soll-Konzept . . . . .	12
2.2.1. Kommunikation zwischen Client und Server . . . . .	12
2.2.2. Evaluation zur Client-Entwicklung . . . . .	13
2.2.3. Weiterentwicklung eines Clients . . . . .	14
2.3. Phasenplan . . . . .	14
2.3.1. Phase 1: Umsetzung der Clients als <i>Proof-of-Concept</i> -Prototypen	14
2.3.2. Phase 2: Umsetzung einer mobilen Anwendung als Messepro- totyp . . . . .	15
<b>3. Grundlagen</b>	<b>17</b>
3.1. Definition eines Caches . . . . .	17
3.1.1. Cache-Arten . . . . .	17

3.1.2. Funktionsweisen von Caches . . . . .	19
3.1.3. Definition eines Caches innerhalb dieser Arbeit . . . . .	19
3.2. Allgemeine Umsetzung des Caches . . . . .	20
3.2.1. Aufbau eines Caches innerhalb des Projekts . . . . .	20
3.2.2. Funktionsumfang eines Caches innerhalb des Projekts . . . . .	21
<b>4. Architektur</b>	<b>23</b>
4.1. Programmarchitektur . . . . .	23
4.2. Anwendungsfälle . . . . .	24
4.2.1. Anwendungsfälle für Phase 1 (Proof-of-Concept-Phase) . . . . .	24
4.2.2. Anwendungsfälle für Phase 2 (Messeprototyp-Phase) . . . . .	25
4.3. Datenbank-Entwurf . . . . .	27
4.4. Rollen-Konzept . . . . .	28
<b>5. Übergreifende Aspekte der Realisierung</b>	<b>29</b>
5.1. Datenbank-System . . . . .	30
5.2. Hosting-Plattform . . . . .	30
5.3. Entwicklungsumgebung . . . . .	31
5.4. Dokumentationslösung . . . . .	32
5.5. Versionsverwaltung . . . . .	32
<b>6. Realisierung der serverseitigen Implementierung</b>	<b>35</b>
6.1. Was ist ein Web Service? . . . . .	35
6.1.1. Besonderheiten eines RESTful Web Services . . . . .	35
6.2. Umsetzung der Komponenten . . . . .	38
6.2.1. Einbindung und Zugriff auf die Datenbank . . . . .	39
6.2.2. Web API . . . . .	41
6.3. Einbindung von Authentifizierung & Autorisierung . . . . .	44
6.3.1. Das Protokoll <i>OAuth2</i> . . . . .	45
6.3.2. JWT and Bearer Token . . . . .	47
6.3.3. Zugriffsbeschränkung per CORS . . . . .	48
6.4. Testen der Server-Funktionalität . . . . .	49
<b>7. Realisierung des Clients als native App</b>	<b>53</b>
7.1. Allgemeine Funktionsweise einer Android-App . . . . .	53
7.2. Komponenten einer Android-App . . . . .	54

7.2.1. User Interfaces . . . . .	54
7.2.2. Activities . . . . .	55
7.2.3. Services . . . . .	57
7.2.4. Prozesse und Threads . . . . .	59
7.2.5. SQLite . . . . .	60
7.3. Was ist <i>Xamarin Platform</i> ? . . . . .	61
7.3.1. Multiplattform-Unterstützung . . . . .	62
7.3.2. Besonderheiten der Android-Umsetzung . . . . .	62
7.3.3. Android Emulator . . . . .	63
7.4. Entwicklung der App mit Xamarin Platform . . . . .	63
7.4.1. Anlegen der Layouts . . . . .	63
7.4.2. Umgang mit Online-/Offline-Situation . . . . .	68
7.4.3. Lokale Datenbank . . . . .	70
7.4.4. Verbindungsaufbau über den ManagementService . . . . .	72
7.4.5. Verbindungsprüfung zum Server . . . . .	73
7.4.6. Umsetzung des Caches . . . . .	74
<b>8. Realisierung des Clients als Web Applikation</b>	<b>77</b>
8.1. Definition einer Single Page Application . . . . .	77
8.2. AngularJS . . . . .	78
8.2.1. Abgrenzung des Begriffs: Komponente . . . . .	78
8.2.2. Dependency Injection . . . . .	78
8.2.3. Services . . . . .	79
8.2.4. Promises . . . . .	79
8.2.5. MVC . . . . .	80
8.2.6. Routing . . . . .	83
8.3. Umsetzung . . . . .	84
8.3.1. Layout mit Twitter Bootstrap . . . . .	84
8.3.2. Herausforderungen durch das statusloses Protokoll HTTP . . . . .	84
8.3.3. Online-Check . . . . .	85
8.4. Erweiterung um Offline-Nutzung . . . . .	85
8.4.1. Implementierung des CachedHttpServices . . . . .	86
8.4.2. Das AppCache-Manifest . . . . .	89
8.5. Fazit . . . . .	90
<b>9. Gegenüberstellung der clientseitigen Implementierungen</b>	<b>91</b>

9.1. Umsetzung als Web App . . . . .	91
9.1.1. Vorteile . . . . .	91
9.1.2. Nachteile . . . . .	92
9.2. Umsetzung als native App . . . . .	92
9.2.1. Vorteile . . . . .	93
9.2.2. Nachteile . . . . .	93
9.3. Fazit aus Meilenstein 1 . . . . .	94
<b>10. Weiterentwicklung eines Clients zu einem Messeprototyp</b>	<b>95</b>
10.1. Anpassungen der Ablauflogik . . . . .	95
10.2. Anpassungen der Oberfläche . . . . .	97
10.3. Implementierung der Statistik . . . . .	98
10.4. User Story der App . . . . .	99
10.5. Fazit aus Meilenstein 2 . . . . .	102
<b>11. Fazit</b>	<b>103</b>
11.1. Ziele / Ergebnisse . . . . .	103
11.2. Erkenntnisse . . . . .	103
11.3. Ausblick . . . . .	104
<b>A. Anhang</b>	<b>105</b>
A.1. Pflichtenheft . . . . .	106
A.2. Cache Post . . . . .	109
A.3. User-Story in der nativen App . . . . .	109
<b>Glossar</b>	<b>117</b>
<b>Abbildungsverzeichnis</b>	<b>123</b>
<b>Tabellenverzeichnis</b>	<b>125</b>
<b>Quelltextverzeichnis</b>	<b>127</b>
<b>Literatur</b>	<b>129</b>

# Abkürzungsverzeichnis

<b>.apk</b>	Android Package
<b>ACL</b>	Access Control Lists
<b>AES</b>	Advanced Encryption Standard
<b>AJAX</b>	Asynchronous JavaScript and XML
<b>ANR</b>	Application Not Responding
<b>API</b>	Application Programming Interface
<b>CORS</b>	Cross-origin resource sharing
<b>CRUD</b>	Create Read Update Delete
<b>CSS</b>	Cascading Style Sheet
<b>DLL</b>	Dynamic Link Library
<b>DRAM</b>	Dynamischer Random Access Memory (RAM)
<b>HTML</b>	Hypertext Markup Language
<b>HTTP</b>	Hypertext Transfer Protocol
<b>HTTPS</b>	Hyper Text Transfer Protocol Secure
<b>IDE</b>	Integrated Development Environment
<b>JSON</b>	JavaScript Object Notation
<b>JWT</b>	JavaScript Object Notation (JSON) Web Token
<b>MSSQL</b>	Microsoft SQL Server
<b>MVC</b>	Model-View-Controller
<b>MVVM</b>	Model-View-ViewModel
<b>OR-Mapper</b>	objekt-relationaler Mapper
<b>PCL</b>	Portable Class Library
<b>RAM</b>	Random Access Memory
<b>RBAC</b>	Role Based Access Control
<b>REST</b>	Representational State Transfer

<b>SPA</b>	Single Page Application
<b>SQL</b>	Structured Query Language
<b>SRAM</b>	Statischer RAM
<b>TCP</b>	Transmission Control Protocol
<b>URI</b>	Uniform Resource Identifier
<b>URL</b>	Uniform Resource Locator
<b>Visual Studio</b>	Microsoft Visual Studio 2015 Community Edition
<b>VM</b>	Virtuelle Maschine
<b>Web-App</b>	mobil-optimierte Webseite
<b>XML</b>	Extensible Markup Language



# Eidesstattliche Erklärung

Gemäß § 17,(5) der BPO erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Ich habe mich keiner fremden Hilfe bedient und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, 15. September 2015

Kevin Schie / Stefan Suermann

## Erklärung

Mir ist bekannt, dass nach § 156 StGB bzw. § 163 StGB eine falsche Versicherung an Eides Statt bzw. eine fahrlässige falsche Versicherung an Eides Statt mit Freiheitsstrafe bis zu drei Jahren bzw. bis zu einem Jahr oder mit Geldstrafe bestraft werden kann.

Dortmund, 15. September 2015

0



# Aufgabenstellung

Mobile Applikationen sind im täglichen Leben allgegenwärtig.

Eine Herausforderung bei diesen Anwendungen ist es, dass sie verlässlich funktionieren müssen, da ansonsten ein Schaden auftreten kann. Da dieses Problem in unterschiedlichen Anwendungen immer wieder auftaucht, ist es sinnvoll, hierfür einen generischen Ansatz anzubieten.

Für mobile Endgeräte können zwei unterschiedliche Lösungsansätze verfolgt werden:

- die Entwicklung nativer Apps und
- die Entwicklung mobiler Webseiten.

Diese beiden Lösungsansätze sollen unter dem Aspekt der Verlässlichkeit gegenübergestellt und verglichen werden.

Der aus der Evaluation hervorgegangene günstigere Lösungsweg soll in einem konkreten Messeprototypen implementiert werden.

Als Beispiel soll eine Applikation für mobile Endgeräte erstellt werden, in der ein Nutzer die Fortschritte seines Trainings festhalten kann. Die dabei entstandenen Daten sollen zentral auf einem Server verwaltet werden. Dieses Szenario ist zwar kein klassisches Beispiel für eine verlässliche Anwendung, allerdings lassen sich an diesem Beispiel alle Konzepte aufzeigen.



# 1. Einleitung

In diesem Kapitel wird das grundlegende Problem und die daraus resultierende Zielsetzung erläutert. Anschließend wird grob auf das Vorgehen zur Lösung dieser Ziele eingegangen.

## 1.1. Problemstellung

Momentan besitzen 57% der Deutschen ein Smartphone. Somit hat sich die Zahl der Smartphone-Nutzer seit Ende 2011 mehr als verdoppelt.<sup>1</sup>

Durch die verstärkte Nutzung geraten Apps immer mehr in den Fokus. Applikationen haben sich im Laufe der Zeit im Alltag ausgebreitet und sind somit für den Endnutzer immer wichtiger. Sei es beim Online-Shopping, Chatten oder der Navigation. Überall finden diese kleinen Programme ihre Verwendung.

Dabei ist es besonders wichtig, dass eine konstante Internetverbindung besteht, um den kompletten Funktionsumfang nutzen zu können. Bis die Umsetzung eines flächendeckenden freien **WLAN!** (**WLAN!**)s in Deutschland abgeschlossen ist, wird eine gute Verbindung über seinen Netzbetreiber benötigt. Diese ist aber noch nicht vollständig und ausreichend im ganzen Land verfügbar.

Deshalb ist es notwendig, dass Apps versuchen Verbindungsabbrüche für den Benutzer zu überbrücken. Dabei besteht die Möglichkeit einer kurzzeitigen Zwischenspeicherung von Daten, solange die Internetverbindung nicht bereitsteht. Änderungen, die in dieser Zeit gemacht werden, sollen aufgenommen und später zur Verfügung gestellt werden, damit auf allen Endgeräten einen einheitlichen Stand der Daten sicherstellen werden kann.

---

<sup>1</sup> SCHMIDT: Anzahl der Smartphone-Nutzer in Deutschland in den Jahren 2009 bis 2015 (in Millionen).

### 1.2. Zielsetzung

Das Hauptziel dieser Arbeit ist der Wissenserwerb der Projektdurchführenden im Bereich der verlässlichen mobilen Applikationen.

Hierzu sollen zwei verlässliche Applikationen unter Nutzung verschiedener Techniken entworfen, umgesetzt und getestet werden. Als Beispiel soll eine Applikation entwickelt werden, die es ermöglicht den Trainingsfortschritt beim Krafttraining darzustellen, aufzunehmen und dauerhaft zu speichern. Zum Speichern der Benutzerdaten, wie Trainingspläne, Übungen und Trainings, wird ein Server benötigt, der die Anfragen der mobilen Geräte annimmt und verarbeitet.

Bei den Applikationen wird während der Entwicklungsphase entschieden, welche der beiden Apps zu einem lauffähigen Messeprototypen weiterentwickelt wird. Diese Einschätzung geschieht aufgrund der Erkenntnisse, welche durch die verschiedenen Umsetzungen gewonnen wurden.

Der Prototyp soll es dem Benutzer ermöglichen durch seine Trainingspläne mit den zugehörigen Übungen zu navigieren und die Daten eines Trainings eingeben zu können. Zudem soll es möglich sein die letzten Trainingseinheiten einzusehen. Das soll unabhängig davon funktionieren, ob das Endgerät eine Verbindung zum Server aufbauen kann oder nicht.

### 1.3. Vorgehensweise

Zum Erreichen der Ziele muss als Erstes eine genauere Betrachtung der entstandenen Problematik durchgeführt werden. Anschließend werden benötigte Kenntnisse für das Erreichen der Ziele gesammelt. Diese gehen in die Planung der allgemeinen Architektur ein, welche Grundlage der späteren Implementierung für Server und Clients ist. Nachdem die Architektur für Server und Clients definiert wurde, können diese umgesetzt werden. Die gewonnen Erkenntnisse aus den Implementierungen werden anschließend gegenübergestellt. Dies bildet die Grundlage für die Entscheidung, welche der beiden zu einem Messeprototyp weiterentwickelt wird. Abschließend wird die Erweiterung zum Messeprototyp umgesetzt.

Projektbegleitend wird eine Dokumentation erstellt, welche jeweils die durchgeführten Maßnahmen und gewonnen Kenntnisse widerspiegelt.

## **1.4. Arbeitsaufteilung**

Da diese Arbeit von zwei Personen durchgeführt wird, muss an dieser Stelle noch aufgeschlüsselt werden, welche Aufgaben von wem durchgeführt werden. Hierzu dient die nachfolgende Tabelle 1.1.

Des Weiteren wird auf Grundlage dieser Tabelle die interne Zeitplanung durchgeführt. Hierdurch kann ermittelt werden, welche Teilaufgaben parallel bearbeitet werden können. Zur Wahrung der größtmöglichen Transparenz wird zwischen der Dokumentation und der Implementierung der benötigten Komponenten unterschieden.

**Tabelle 1.1.:** Arbeitsaufteilung

<b>Aufgaben</b>	realisiert von	
	Kevin Schie	Stefan Suermann
<b>Implementierung</b>		
Server		X
native App	X	
mobile Web Applikation		X
Messeprototyp	X	
<b>Dokumentation</b>		
Einleitung		X
Problemanalyse	X	
Grundlagen	X	
Architektur		X
Aspekte der Realisierung		X
Realisierung der serverseitigen Implementierung		X
Relisierung der clientseitigen Implementierung als native App	X	
Relisierung der clientseitigen Implementierung als Webapplikation		X
Gegenüberstellung der clientseitigen Implementierung	X	X
Weiterentwicklung eines Clients zu einem Messeprototyp	X	
Fazit		X
Pflichtenheft		X
Cache Post	X	
User-Story in der nativen App	X	
Funktionsumfang	X	



## 2. Problemanalyse

Im letzten Kapitel wurde die Aufgabenstellung grob beschrieben. Nun sollen die angesprochenen Probleme feiner analysiert werden, sodass sich konkrete Ziele ergeben. Diese Ziele bilden die Grundlage für die Entscheidungen zum weiteren Vorgehen während der Umsetzung, welche in einem Soll-Konzept dargelegt wird.

### 2.1. Problembeschreibung

Aus der groben Problembeschreibung lassen sich folgende technische Herausforderungen ablesen:

Der Server dient als zentrale Datenhaltung für verschiedene Clients. Darum ist es nötig, dass Client und Server dafür ausgelegt werden, über eine standardisierte Schnittstelle zu kommunizieren, um die anfallenden Daten auszutauschen. Diese muss einen Authentifizierungs- und Autorisierungsmechanismus bereitstellen, so dass jeder Nutzer nur an seine eigenen Daten gelangt. Darüber hinaus ist die Schnittstelle so zu implementieren, dass diese Kommunikation optional ist. Dadurch entsteht eine Ausfallsicherheit, da die Funktionalität des Endgeräts autonom von Server genutzt werden kann. Um diese zu gewährleisten, muss ein Client die Möglichkeit haben, die Verbindung zum Server zu prüfen. Schlägt diese Prüfung fehl, muss der Client mit entsprechenden Maßnahmen, wie beispielsweise einer lokalen Zwischenspeicherung, reagieren. Hierfür muss zum einen der Zugriff auf Funktionen, welche zwingend eine Verbindung benötigen, reguliert werden. Zum Anderen müssen lokal anfallende Daten bei fehlender Verbindung zwischengespeichert werden. Letzteres hat zwei Vorteile:

- Neu angelegte Daten gehen dem Nutzer nicht verloren, obwohl sie nicht zum Server geschickt werden.
- Dem Nutzer bleiben Funktionalitäten und bereits vorhandene Daten erhalten, obwohl keine Verbindung zum Server besteht.

Fallen neue lokale Daten an, ergibt sich aus der Problembeschreibung, dass diese bei späterer Verbindung zum Server persistiert werden. Dies muss die Kommunikationschnittstelle durch einen geeigneten Synchronisationsmechanismus unterstützen.

## 2.2. Soll-Konzept

Aus der vorangehenden konkreten Problembeschreibung ergibt sich ein Soll-Konzept für die Umsetzung des Projekts, welches in den folgenden Abschnitten erläutert wird.

### 2.2.1. Kommunikation zwischen Client und Server

Ziel soll die Umsetzung zweier mobiler Applikationen sein, welche mit einem selbst entwickelten Server kommunizieren. Während der Kommunikation muss festgestellt werden, ob- bzw. wann diese abbricht. Abhängig davon müssen die Applikationen das Verhalten zwischen Online- und Offline-Modus umstellen.

Wenn der Server erreichbar ist, können die benötigten Daten dort direkt abgefragt und lokal angezeigt werden. Zum Entgegenwirken von Datenverlust für den Benutzer, werden die bei dieser Abfrage erhaltenen Informationen lokal gespeichert werden. Daten, die im Online-Status angelegt werden, können direkt zum Server übertragen werden. Dort werden sie dann persistent gespeichert und sind für diesen Benutzer von überall erreichbar.

Wenn die Verbindung abgebrochen ist, können die Applikationen nur auf die abgespeicherten Daten zurückgreifen und diese anzeigen. Die Applikationen sollen die Möglichkeit bieten, auch im Offline-Zustand Daten anzulegen. Geschieht dies, werden die Daten ebenfalls lokal gespeichert. Hierbei werden sie als Offline-Daten mittels Flag erkennbar, in dem lokalen Speicher, abgelegt.

Wenn die Verbindung zwischen Server und Client gerade wieder hergestellt werden kann, müssen lokal angelegte Daten zum Server übertragen werden. Zur Erkennung, welche Daten an den Server übertragen werden müssen, dient das Offline-Flag aus

den Daten des lokalen Speichers. Bei dieser Übertragung muss eine Synchronisation der Daten erfolgen.

### 2.2.2. Evaluation zur Client-Entwicklung

Während der Umsetzung sollen zwei mobile Applikationen entwickelt werden. Hierzu beschreibt die Aufgabenstellung die Implementierung in zwei unterschiedlichen Technologien. Dabei soll evaluiert werden, welche Technik für die Umsetzung der Anforderungen am Besten geeignet ist. Deshalb sollen folgende Applikationen entwickelt werden:

- eine mobil-optimierte Webseite (Web-App)
- eine native App

#### Single Page Application

Die Web-App soll als Webseite im Browser umgesetzt werden. Damit die clientseitige Logik einfacher umgesetzt werden kann, soll die Web-App als Single Page Application (SPA) umgesetzt werden. Hierbei soll konsequent auf aktuelle Web-Techniken aus HTML5, CSS3 und Javascript gesetzt werden.

Da eine mobile Nutzung im Vordergrund steht, soll die SPA sich responsiv verhalten. Dadurch wird eine Nutzung auf kleinen Displays unterstützt. Dies erhöht die Vergleichbarkeit der Applikationen, da beide Varianten problemlos auf dem gleichen Gerät getestet werden können.

#### Native App

Die native App soll für Android entwickelt werden. Android wurde als Plattform ausgewählt, um die Vorteile des offenen Systems nutzen zu können. So ist es beispielsweise möglich die entwickelte App auf einem Testsystem zu installieren, ohne - wie bei *Apples* Web-App nötig - einen Entwickler-Account anlegen zu müssen.

Zudem ist es bei einer iOS-App notwendig, das Aufspielen einer Testapplikation über ein spezielles Entwickler-Tool in XCode durchzuführen. Diese Hürde fällt bei einer

Android-App weg. Des Weiteren ist das Android-Betriebssystem weiter verbreitet.<sup>1</sup> Dadurch kann bei einer möglichen späteren Weiterentwicklung eine größere Akzeptanz der App erzielt werden.

### 2.2.3. Weiterentwicklung eines Clients

Auf Grundlage der Evaluation soll eine der beiden Applikationen ausgewählt und anschließend zu einem rudimentären Messeprototyp weiterentwickelt werden. Diese soll eine komplette User Story implementieren. Als Beispiel soll eine Fitness-App dienen. Hierbei kann ein Nutzer Trainingsdaten verwalten.

## 2.3. Phasenplan

Aus dem nun vorliegenden Soll-Konzept kann ein Phasenplan erzeugt werden. Dieser zeigt einen groben Projektablauf auf und spiegelt parallel ablaufende Entwicklungen wieder. Das gesamte Projekt kann in zwei Phasen unterteilt werden, welche nachfolgend genauer beschrieben werden (siehe Tabelle 2.1).

**Tabelle 2.1.:** Phasenplan

Phase	Titel
1	Umsetzung der Clients als <i>Proof-of-Concept</i> -Prototypen
2	Umsetzung einer mobilen Anwendung als Messeprototyp

### 2.3.1. Phase 1: Umsetzung der Clients als *Proof-of-Concept*-Prototypen

In dieser Phase werden Erkenntnisse zur Implementierung einer verlässlichen mobilen Anwendung gesammelt.

Hierbei müssen folgende Teilschritte durchgeführt werden:

---

<sup>1</sup>IDC: Prognose zu den Marktanteilen der Betriebssysteme am Absatz von Smartphones weltweit in den Jahren 2015 und 2019.

1. Erwerb grundsätzlicher Kenntnisse eines Caches und dessen Implementierung
2. Implementierung des Servers
3. Erstellung einer Web-App
4. Erstellung einer nativen App
5. Evaluierung der Erkenntnisse

Da beide Clients während der Implementierung den Server benötigen, müssen die Teilschritte bis einschließlich Schritt 2 nacheinander abgearbeitet werden. Die Umsetzung der beiden Clients kann anschließend parallel erfolgen. Diese Phase endet mit der Gegenüberstellung der gewonnen Erkenntnisse und der Auswahl einer Technik für Phase 2.

### **2.3.2. Phase 2: Umsetzung einer mobilen Anwendung als Messeprototyp**

In dieser Phase wird die in Phase 1 gewählte Technik benutzt, um einen Messeprototyp zu entwickeln. Hierbei sollen alle Funktionalitäten implementiert werden, um einen Anwendungsfall vollständig durchzuführen. Als Anwendungsfall soll ein Nutzer ein neues Trainingsdatum anlegen. Dabei soll es irrelevant sein, ob eine Verbindung zum Server besteht, oder nicht. Dieser Anwendungsfall wird als grafisch aufbereitete User Story im Umfeld des fertig implementierten Messeprototyps vorgestellt.



## 3. Grundlagen

Das folgende Kapitel gibt eine Übersicht über die verschiedenen Funktionsweisen eines *Caches*. Darauf aufbauend wird anschließend herausgestellt, welche Art von Speicherung in diesem Projekt umgesetzt wird. Dabei wird ebenfalls auf die verschiedenen Auffassungen des Begriffes *Cache* eingegangen und deren Unterschiede erläutert.

### 3.1. Definition eines Caches

Ein *Cache* wird im Allgemeinen als eine Speicherregion oder Puffer verstanden, die besonders schnell erreichbar ist, da oft verwendete Daten zwischengespeichert werden. Somit erkaufte man sich durch einen erhöhten Speicherbedarf einen Performancegewinn.

#### 3.1.1. Cache-Arten

*Caches* werden in verschiedenen Umgebungen eingesetzt. Hierbei kann zwischen folgenden Arten unterschieden werden:

- Memory Cache
- Internet Browser Cache
- Disk Caching
- Server Caching

#### ***Memory Cache***

Der *Memory Cache* wird in Computern verwendet, um den sehr schnellen SRAM des Rechners auszunutzen. Diese Art macht es sich zunutze, dass Programme immer dieselben Daten oder Befehle ausführen. Bereits errechnete Ergebnisse werden vom Betriebssystem in diesem *Cache* gespeichert, um zum Beispiel darauf aufbauend schnellere Berechnungen vollziehen zu können. Als Alternative würde der DRAM zur Verfügung stehen, welcher jedoch langsamer ist.<sup>1</sup>

#### ***Internet Browser Cache***

Der *Internet Browser Cache* wird ähnlich, aber in einem anderen Einsatz verwendet. Dieser speichert beliebte Seiten des Benutzers zwischen, um den Seitenaufruf zu beschleunigen. Dabei werden Dateien und Requests zu der besuchten Seite gespeichert. Wenn der Benutzer auf die vorherige Seite zurück navigiert, kann der Browser viele Dateien wiederverwenden und muss nicht die gesamte Seite nachladen.<sup>2</sup>

#### ***Disk Caching***

*Disk Caching* wird besonders beim Lesen von Festplatten verwendet. Dabei werden Daten im *Memory Buffer* gespeichert. Dieser liegt heutzutage in einem gesonderten Bereich auf der Festplatte. Dieser Bereich kann sich jedoch auch im RAM des Computers befinden.<sup>3</sup>

#### ***Server Caching***

Beim *Server Caching* geht es darum, den *Traffic* in einem Netzwerk zu minimieren, indem die meistbesuchten Seiten auf einem *Caching Server* gespeichert werden. Wenn ein Benutzer aus dem Netzwerk eine dieser Seiten aufruft, wird diese Seite aus dem *Cache* zurückgegeben und der Request muss nicht wieder über das Internet geleitet werden, sondern wird direkt im internen Netzwerk beantwortet.<sup>4</sup>

---

<sup>1</sup>Vgl. GUMMER/SOMMER: Einführung in die Informatik, S.48f.

<sup>2</sup>ROUSE: Cache (Computing) Definition.

<sup>3</sup>DERS.: Disk Cache Definition.

<sup>4</sup>THE APACHE SOFTWARE FOUNDATION: HTTP Proxy Caching.



### 3.1.2. Funktionsweisen von Caches

*Caches* lassen sich in zwei unterschiedliche Funktionsweisen unterteilen. Diese werden nachfolgend genauer vorgestellt.

#### Store and Forward

Das *Store and Forward*-Prinzip ist eine spezielle Art des *Cachings*, das Netze überbrücken soll, welche Verzögerungen tolerieren. Demgegenüber gibt es die Techniken *Streaming* und Internettelefonie, die keine Verzögerungen in der Verbindung tolerieren.

*Store and Forward* ist mit einer TCP-Übertragung vergleichbar. Beide Techniken speichern kleinere Datenteile zwischen, um so das gesamte Datum in Gänze zu erhalten. Der Vorteil besteht darin, dass eine Zwischenstation die übertragenen Daten speichert, auf Integrität prüft und, wenn gewünscht, weiterleitet.<sup>5</sup>

Soll ein Datum auf der Festplatte gespeichert werden, wird es dazu jedoch sicherheitshalber im Puffer zwischengespeichert. Wird dieses Datum direkt danach abgerufen, ist aber noch nicht auf der Festplatte gespeichert, dann kann das Datum aus dem Puffer geladen werden und die geringe Geschwindigkeit der Festplatte wird dem Anwender nicht bewusst.<sup>6</sup>

#### Function Cache

Beim *Function Cache* oder auch *Memoization* handelt es sich um einen *Cache*, der die Funktionsaufrufe eines Programmes samt Ergebnissen speichert. Diese werden dann in den meisten Fällen für darauf aufbauende Berechnungen oder Aktionen verwendet und sorgen somit für einen enormen Geschwindigkeitsanstieg.<sup>7</sup>

### 3.1.3. Definition eines Caches innerhalb dieser Arbeit

Innerhalb dieser Arbeit wird der Begriff des *Caches* als eine abgewandelte Technik zur lokalen Zwischenspeicherung von Daten auf einem mobilen Gerät verstanden.

---

<sup>5</sup>DATAKOM BUCHVERLAG GMBH: Store-and-Forward-Verfahren.

<sup>6</sup>ARCITURA EDUCATION INC.: Service Data Forward Cache.

<sup>7</sup>SAUMONT: Do it in Java 8: Automatic memoization.

Die allgemeine Definition geht bei einem *Cache* von einer Performancesteigerung aus (siehe Kapitel 3.1). In diesem Projekt wird das Speichern von Daten jedoch dazu verwendet, um Daten auch offline zur Verfügung zu stellen und damit eine Autonomie vom Server zu erreichen. Die bestehenden Eigenschaften zur Ersetzung von Daten und den verschiedenen Arten der Datenspeicherung werden beibehalten, jedoch im ersten Meilenstein nur rudimentär umgesetzt. Dabei wird eine Art *Store and Forward* umgesetzt. Dagegen wird bevorzugt auf den Server zugegriffen, da dieser als primärer Persistenzspeicher fungiert.

Ein *Function Cache* wird nicht umgesetzt, da dieser in diesem Anwendungsfall nicht optimal wäre. Es werden die Daten benötigt, die untereinander auch Beziehungen besitzen. Dabei reicht es nicht aus, die Funktionsaufrufe mit den entsprechenden Daten zu speichern, da man die gespeicherten Daten in einigen Fällen über mehrere Abfragen erhalten würde und diese dann mehrfach gespeichert werden würden. Dieses Problem würde dann die Effizienz des Zwischenspeicherns umgehen.

## 3.2. Allgemeine Umsetzung des Caches

Der *Cache* wird in beiden Applikationen als eine lokale Datenbank umgesetzt und spiegelt die Datentypen des Servers in einem möglichst großen Umfang wider (siehe Kapitel 4.3 zum Aufbau der Server-Datenbank). Die genaue Umsetzung und Auswahl der Entitäten und Attribute muss entsprechend der Umsetzung und der technischen Möglichkeiten geschehen. Dabei wird jedoch weiterhin auf eine möglichst große Übereinstimmung zwischen den verschiedenen Applikationen geachtet. Somit sind die Entwicklungen besser zu vergleichen und bieten den Autoren damit ein besseres Maß zur Entscheidung, welche Applikation zu einem Messeprototypen weiterentwickelt wird.

### 3.2.1. Aufbau eines Caches innerhalb des Projekts

Der *Cache* als solches ist die Kombination aus der Logik, die in der Applikation zur Datenhaltung mit umgesetzt wird, und einer lokalen Datenbank zur Speicherung der Daten. Die Schicht der *Business-Logik* muss dabei Methoden zur Verfügung stellen, um die Daten lokal zu speichern und diese bei Bedarf wieder auslesen zu können. Des Weiteren muss die Logik zur Synchronisierung von Daten zwischen der lokalen

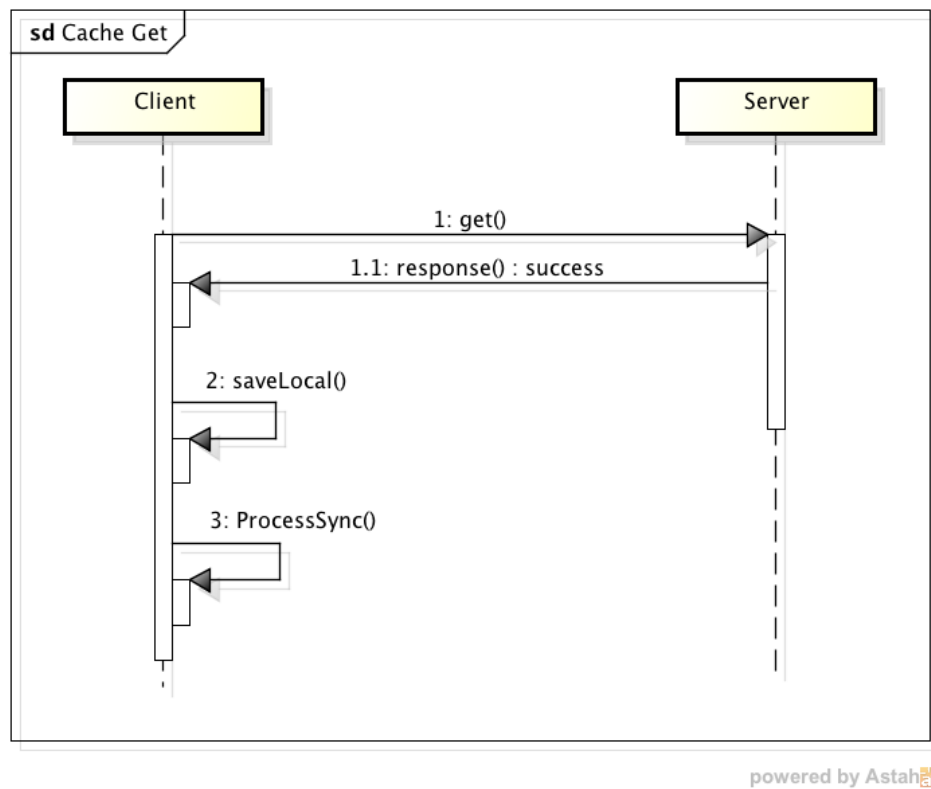
und der Server-Datenbank umgesetzt werden. Diese wird im Folgenden allgemein beschrieben.

#### 3.2.2. Funktionsumfang eines Caches innerhalb des Projekts

Der *Cache* muss die Daten auf demselben Stand halten, wie sie auf dem Server vorliegen. Deshalb bietet es sich an, Daten, die zum Server geschickt werden, auch redundant lokal zu speichern. Genauso verfährt man mit Daten, welche vom Server abgerufen werden. Somit hat man keine unnötigen Abfragen zum Erhalt der Datenkonsistenz zwischen den beiden Ebenen. Diese Strategie hat somit einen positiven Einfluss auf die Performance der Applikationen und entlastet den Server von übermäßigen *Requests*.

In dem nachstehenden Sequenzdiagramm (siehe Abbildung 3.1) ist der Ablauf für die Logik des *Caches* bei einem *Get*-Aufruf an den Server zu sehen. Ein Sequenzdiagramm zum Ablauf eines *POST*-Requests liegt dem Anhang bei (siehe Abbildung A.1).

Die Synchronisation soll demnach im Anschluss einer Serververbindung geschehen, da man in diesem Fall sicher sein kann, dass eine Verbindung besteht, die dafür verwendet werden kann. Dementsprechend funktioniert auch der *Post* oder das Hochladen von Daten (siehe Anhang A.2).



**Abbildung 3.1.:** Abrufen vom Server

## 4. Architektur

In diesem Kapitel werden die architektonischen Randbedingungen für die Entwicklung der Applikation beschrieben. Hierzu zählt die allgemeine Systemarchitektur. Anschließend wird beschrieben, welche Anwendungsfälle in die späteren Prototyp und im Messeprototyp gegeben sein sollen. Daraus resultiert der Aufbau der Datenbank und die schlussendliche Systemarchitektur. Als Grundlage dient das Pflichtenheft. Dieses liegt dieser Arbeit gesondert im Anhang bei (siehe Kapitel A.1). Der Arbeitstitel dieses Projekts wird, in Anlehnung an ein Fitness-IT-Projekt, *fIT* lauten.

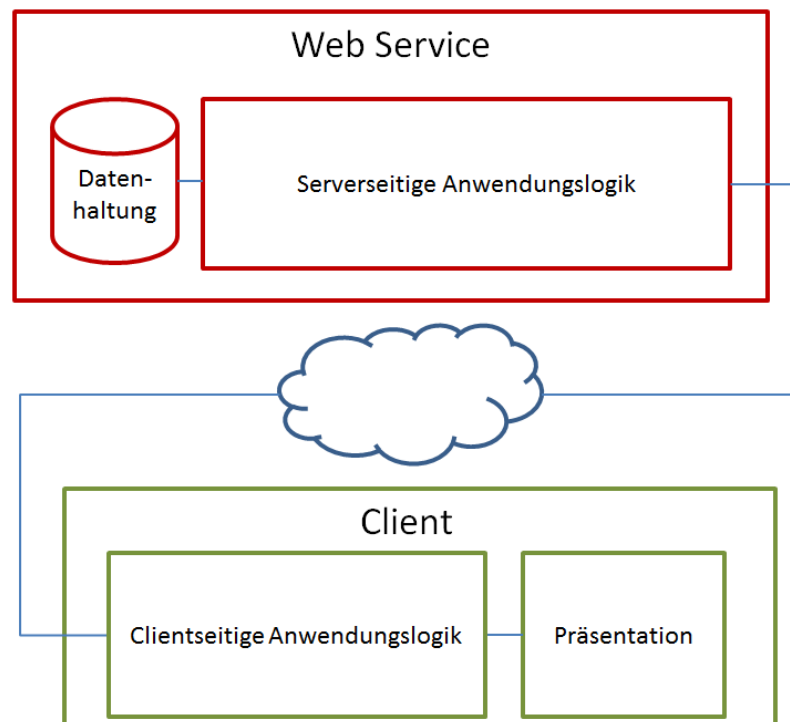
### 4.1. Programmarchitektur

Da verschiedene Clients implementiert werden sollen, ist es sinnvoll das Projekt als Mehrschichtenarchitektur für eine verteilte Anwendung zu implementieren.

Der Server stellt dabei die Funktionen zur Nutzung durch die Clients bereit. Konkret greift der Server per OR-Mapper auf die Datenbank zu, bereitet die Daten in der Applikationsschicht auf und reicht sie über eine Representational State Transfer (REST)-Schnittstelle an den anfragenden Client weiter. Eine detaillierte Beschreibung zu REST wird in Kapitel 6.1.1 gegeben. Bei dieser Kommunikation muss gewährleistet sein, dass ein Nutzer nur die Daten abrufen darf, für die er autorisiert wurde.

Clientseitig werden Daten in einem lokalen Speicher zum Schutz vor eventuellen Verbindungsabbrüchen zwischengespeichert. Anschließend werden die erhaltenen Daten auf dem Endgerät für die Anzeige aufbereitet und angezeigt.

Abbildung 4.1 bildet diesen Aufbau mit den zugehörigen Verantwortlichkeiten grafisch ab:



**Abbildung 4.1.:** Aufbau der Anwendung

## 4.2. Anwendungsfälle

Das Pflichtenheft sieht eine Unterteilung des Projekts in zwei aufeinanderfolgende Phasen vor (siehe Kapitel 2.3). Hierbei werden erst Proof-of-Concept-Prototyp entwickelt. Anschließend wird ein Prototyp zum Messeprototyp weiterentwickelt.

Für diese beiden Prototypen müssen andere bzw. erweiterte Anwendungsfälle implementiert werden. Aus diesem Grund werden nachfolgend für die beiden Implementierungsschritte die Anwendungsfälle einzeln aufgeschlüsselt.

### 4.2.1. Anwendungsfälle für Phase 1 (Proof-of-Concept-Phase)

Aus dem Pflichtenheft ergeben sich folgende Anwendungsfälle für die erste Phase des Projekts (siehe Abbildung 4.2):

- Es soll möglich sein, sich an der Anwendung anzumelden.

- Es soll möglich sein, eine Entität mit Daten (Trainingsplan, Training oder Übung), unabhängig von der Verbindung zum Web Service, persistent anzulegen, zu ändern und zu speichern.
- Optional soll sich ein Nutzer an der Anwendung registrieren können.

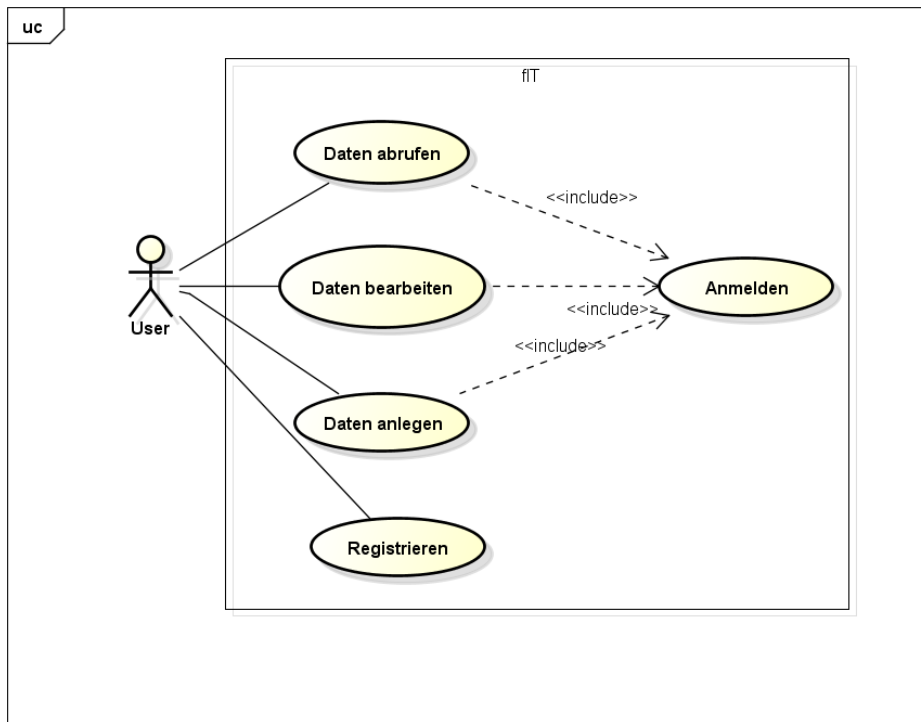


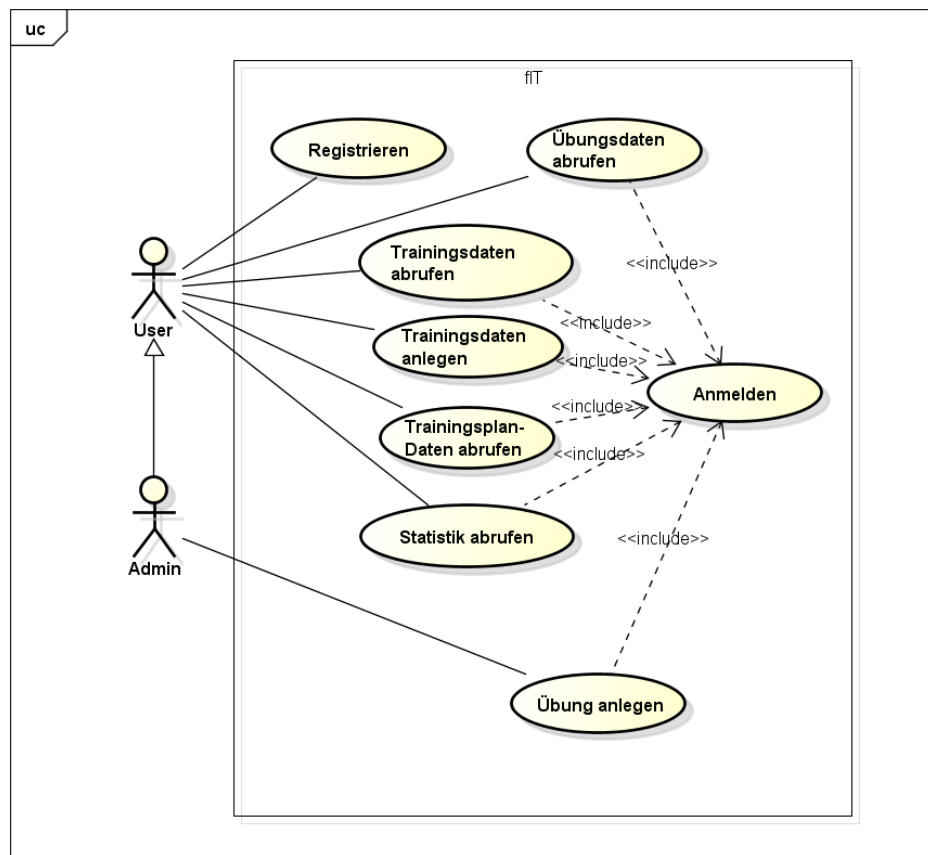
Abbildung 4.2.: Anwendungsfälle des *Proof-of-Concept*-Prototyp

#### 4.2.2. Anwendungsfälle für Phase 2 (Messeprototyp-Phase)

Für Phase 2 werden die bereits vorgestellten Anwendungsfälle weiter verfeinert. Daraus ergeben sich folgende Anwendungsfälle (siehe Abbildung 4.3):

- Ein Nutzer soll sich an der Anwendung anmelden können.
- Ein Nutzer soll seine eigenen Trainingsplan-Daten abrufen können.
- Ein Nutzer soll zu einem seiner Trainingspläne alle zugehörigen Übungen abrufen können.
- Ein Nutzer soll zu einer dieser Übungen seine bisherigen Trainingsdaten abrufen können.

- Ein Nutzer soll zu einer Übung ein neues Training anlegen können.
- Alle nicht-optionalen Anwendungsfälle müssen unabhängig von einer Serververbindung funktionieren und eventuell anfallende Daten dauerhaft speichern.
- Optional: Ein Nutzer soll eine Statistik der letzten Trainingsdaten zu einer Übung abrufen können. Neu erstellte Trainingsdaten aktualisieren diese Statistik.
- Optional: Ein Nutzer soll sich an der Applikation registrieren können.
- Optional: Ein Nutzer mit der Rolle *Administrator* soll neue Übungen anlegen können.

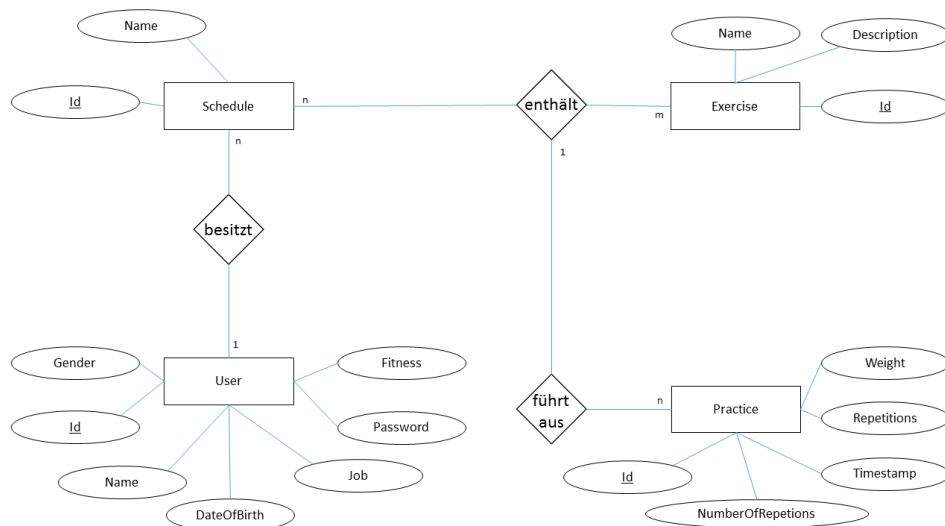


**Abbildung 4.3.:** Anwendungsfälle des Messeprototyp



## 4.3. Datenbank-Entwurf

Aus den definierten Anwendungsfällen ergibt sich die Struktur für die Datenbank. Diese wird in Abbildung 4.4 dargestellt. Als Grundlage werden die Anwendungsfälle der zweiten Phase genutzt, um spätere Anpassungen nach Beendigung der ersten Phase zu vermeiden.



**Abbildung 4.4.:** Datenbank-Entwurf

## 4.4. Rollen-Konzept

Im Pflichtenheft wird für die zweite Phase eine Unterscheidung im Bereich der Berechtigungen vorgenommen. So dürfen beispielsweise nur Administrationen neue Übungen anlegen. Aus diesem Grund ist es notwendig, ein Rollenkonzept zu entwickeln, welches den Zugriff auf bestimmte Ressourcen reguliert.

Aus den Aussagen, die im Pflichtenheft getroffen wurden, geht hervor, dass sich der Nutzer in einer der nachfolgenden Status befindet, wenn er auf den Web Service zugreifen will:

- unautorisiert

Der Nutzer hat sich noch nicht gegenüber des Web Servers authentifiziert. In diesem Status kann der Nutzer sich mit seinen Anmeldedaten einloggen oder als neuer Nutzer an der Web Application Programming Interface (API) registrieren.

- Rolle **Nutzer**

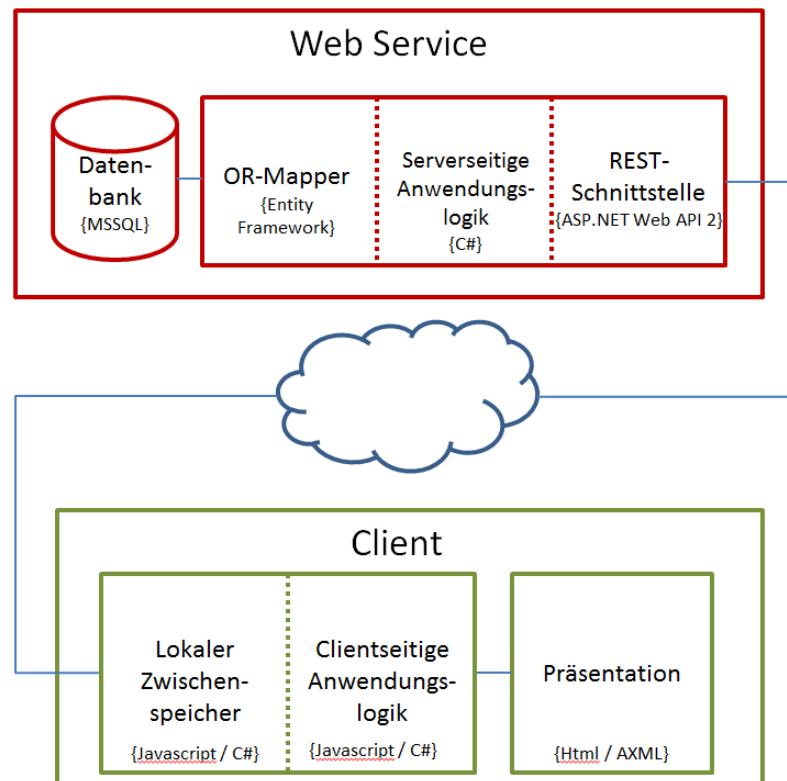
Jeder angemeldete Nutzer besitzt die Rolle *Nutzer*. Ein normaler Nutzer kann seine Daten einsehen. Dies beinhaltet seine Trainingspläne, deren Übungen und die dazu angelegten Trainingseinheiten.

- Rolle **Administrator**

Ein *Administrator* ist ebenfalls ein Nutzer. Er kann zusätzlich neue Übungen anlegen.

## 5. Übergreifende Aspekte der Realisierung

In diesem Kapitel werden allgemeine Komponenten für die Umsetzung dieses Projektes beschrieben. Dabei werden jeweils nur die Techniken vorgestellt, welche von mehreren Komponenten genutzt werden, sodass sie in den jeweils dafür vorgesehenen Kapiteln nicht mehrfach genannt werden müssten. Abbildung 5.1 zeigt hier bei die genutzten Produkte zu den definierten Komponenten aus Kapitel 4.1. Begriffe, welche nur in einem Teilbereich des Systems (Server oder einer der Clients) genutzt werden, werden in den jeweiligen Kapiteln gesondert besprochen). Als Leitfaden wird versucht, möglichst viele Entwicklungswerkzeuge eines Unternehmens zu benutzen, um positive Synergieeffekte, beispielsweise in Form von leichter Kommunikation, zwischen den gewählten Komponenten zu gewährleisten. Zur Umsetzung dieses Projekts werden weitestgehend Produkte der Softwarefirma *Microsoft* genutzt.



**Abbildung 5.1.:** Zuordnung von Produkten und Techniken zu Verantwortlichkeiten

### 5.1. Datenbank-System

Als Datenbanksystem wird die Lösung von *Microsoft* verwendet. Hierbei handelt es sich um *Microsoft SQL Server (MSSQL)*. Durch die einheitliche Nutzung von *Microsoft*-Produkten kann für den Zugriff auf die Datenbank der objekt-relationaler Mapper (OR-Mapper) genutzt werden. Dieser erlaubt es, direkt aus Modell-Klassen Datenbank-Entitäten zu entwickeln. Dieser Vorgang wird in Kapitel 6.2.1 näher erläutert.<sup>1</sup>

### 5.2. Hosting-Plattform

Sowohl der Web Service als auch der SPA-Client müssen im Internet verfügbar gemacht werden, damit sie so von überall erreichbar sind. Hierzu bietet *Microsoft* die Hosting-Plattform *Azure* an. Diese ermöglicht es, direkt aus *Microsoft Visual Studio*

---

<sup>1</sup>MICROSOFT: SQL Server 2014.

2015 Community Edition (*Visual Studio*) heraus, seine Webanwendung zu veröffentlichen. Gleichzeitig lässt sich eine Datenbank hosten, welchen der Web Service direkt benutzen kann. Zusätzlich ist *Azure* sehr gut skalierbar, was den Einsatz als Hosting-Plattform für kleine Prototyp-Projekte optimal unterstützt.<sup>2</sup>

## 5.3. Entwicklungsumgebung

Für die Entwicklung sämtlicher Komponenten wird *Visual Studio* benutzt. Hierbei handelt es sich um die Standard-Entwicklungsplattform von *Microsoft*. Diese Entscheidung wurde aus folgenden Gründen getroffen:

- Die Clients sollen durch zwei Drittanbieter-Frameworks umgesetzt werden, um den Implementierungsaufwand zu verringern. Eine genauere Erläuterung der Funktionsweise, der Vorteile und der Einbindung werden in den Kapiteln 7 und 8 gegeben. Beide Frameworks sind entweder bereits in die Entwicklungsumgebung integriert oder können problemlos nachträglich zum Projekt hinzugefügt werden. Die hierfür benutzten Programmiersprachen *C#* und *Javascript* werden vollständig mit gängigen Features einer Integrated Development Environment (IDE), wie Autovervollständigung, Syntax-Highlighting und Debugger, unterstützt.
- Die Entwicklung von Web Anwendungen wird stark erleichtert, da *Visual Studio* mit einem integrierten Web Server ausgeliefert wird. Dadurch können entwickelte Applikationen direkt lokal getestet werden, ohne, dass ein zusätzlicher Web Server installiert oder die Anwendung auf einen Web Server veröffentlicht werden muss.
- Es besteht eine starke Integration zu anderen *Microsoft* Produkten. Hierzu zählen die Hosting-Plattform *Microsoft Azure* und das Datenbank-System *MSSQL*.
- Die Entwicklungsumgebung kann benötigte Komponenten und deren Abhängigkeiten durch den integrierten Paketmanager *NuGet* nachladen. Dadurch entfällt das nachträgliche Herunterladen von *Dynamic Link Library (DLL)*-Dateien, wodurch Kompatibilitätsprobleme verringert werden.<sup>3</sup>

---

<sup>2</sup>DERS.: Was ist Azure?

<sup>3</sup>DERS.: Visual Studio.

## 5.4. Dokumentationslösung

Zur Umsetzung der Projektdokumentation wird das Framework *LaTeX* benutzt. Dieses erweitert das Textsatzsystem *Tex* mit Makros. Die Makros können als Markup benutzt werden, um einen Text zu strukturieren. Dadurch kann, wie bei Nutzung von Markup üblich, sehr einfach Inhalt und Aussehen getrennt werden. Somit hat man den Vorteil, dass man nur einmal das Aussehen pflegen muss und anschließend ausschließlich den Inhalt mit dem gewünschten Markup definieren muss. Diese Dateien können nachfolgend zu einer *PDF*-Datei kompiliert werden.<sup>4</sup>

Ein weiterer Vorteil ist, dass die Inhaltsdateien aus einfachen Textdateien bestehen. Bei dieser Art von Dateien lassen sich Änderungen (z.B. Mittels Versionsverwaltung) leichter nachvollziehen als beispielsweise bei *DOCX*-Dateien, welche das Textverarbeitungsprogramm *Microsoft Word* benutzt. Dabei handelt es sich um Binärdateien, welche die Nachvollziehbarkeit erschweren.

## 5.5. Versionsverwaltung

Dieses Projekt wird von mehreren Personen durchgeführt. Dabei ist es unerlässlich, dass die Projektstände zwischen den Teilnehmern nachvollziehbar verwaltet werden, da sonst schnell die Übersicht verloren gehen kann. Um dem entgegenzuwirken wird *Git* zur Versionsverwaltung eingesetzt.

Diese Entscheidung wurde aus folgenden Gründen getroffen:

- *Git* ist ein verteiltes Versionsverwaltungssystem. Das bedeutet, dass sowohl ein lokales- als auch ein globales Repository entsteht. Dies macht die Entwicklung leichter: Im vorliegenden Fall wird beispielsweise solange lokal entwickelt, bis eine Funktionalität vollständig implementiert und getestet ist. Anschließend werden alle lokalen Stände in das globale Repository eingchecked, sodass immer ein funktionierender Stand im globalen Repository vorliegt.<sup>5</sup>
- Für die Nutzung von *Git* benötigt man keinen eigenen Server, in dem das globale Repository aufgespielt wird. Hierfür kann die Webseite <https://github.com>

---

<sup>4</sup>DEUTSCHSPRACHIGE ANWENDERVEREINIGUNG TEX E.V.: Was ist LaTeX?

<sup>5</sup>GIT: Los geht's - Git Grundlagen.

genutzt werden. Diese stellt diese Funktionalität kostenlos bereit und ist weltweit über das Internet verfügbar.

- Für *Git* gibt es eine direkte Integration in die Visual Studio.





## 6. Realisierung der serverseitigen Implementierung

In diesem Kapitel wird näher auf die Implementierung des in Kapitel 4.1 besprochenen Web Services eingegangen. Es enthält eine Übersicht über die genutzten Komponenten und die konkreten Techniken, welche für die Implementierung genutzt wurden. Anschließend wird gesondert auf Sicherheitsaspekte in Verbindung mit RESTful-Architekturen eingegangen. Der hier beschriebene Web Service kann über *<http://fit-bachelor.azurewebsites.net/>* erreicht werden.

### 6.1. Was ist ein Web Service?

Um verteilte Systeme aufzubauen ist es nötig, eine Struktur zu implementieren, mit der Maschinen untereinander kommunizieren können. Diese Aufgabe übernehmen Web Services. Sie stellen innerhalb eines Netzwerkes Schnittstellen bereit, damit Maschinen plattformübergreifend Daten austauschen können. Hierbei wird meistens Hypertext Transfer Protocol (HTTP) als Träger-Protokoll genutzt, um eine einfache Interoperabilität zu gewährleisten.<sup>1</sup> Die dabei angeforderten Daten werden in der Regel als Extensible Markup Language (XML) oder JSON übermittelt.

#### 6.1.1. Besonderheiten eines RESTful Web Services

Da Web Services in der Regel HTTP als Protokoll verwenden, wurde die Idee zur Implementierung eines Web Services erweitert, um die Möglichkeiten des Protokolls

---

<sup>1</sup>BOOTH u. a.: Web Services Architecture.

besser zu benutzen. Daraus entstand das Programmierparadigma REST. Als REST-Server bzw. RESTful Web Service bezeichnet man einen Web Services, welcher die strikte Nutzung von HTTP als Programmierparadigma umsetzt. Dies meint, dass, wie im Internet üblich, Uniform Resource Identifier (URI) bzw. Uniform Resource Locator (URL) zur eindeutigen Identifikation von Ressourcen genutzt werden.<sup>2</sup> Diese Ressourcen lassen sich nachfolgend werden einige Prinzipien von REST näher beleuchtet.

### Adressierbarkeit

Im Gegensatz zu anderen Web Service-Implementierungen stellen RESTful Web Services keine Methoden oder aufrufbare Funktionalitäten, sondern ausschließlich Ressourcen, zu Verfügung. Dies hat den Vorteil, dass die Schnittstelle leicht und eindeutig beschrieben werden kann, da ein Aufruf einer URL an den REST-Service immer eindeutig auf eine Ressource zeigt, ohne, dass Abhängigkeiten oder ein Kontext berücksichtigt werden müssen.

In den meisten Fällen, wie auch in den Anwendungsfällen dieser Arbeit, soll der Web Service die Funktionalitäten Create Read Update Delete (CRUD) bereitstellen. Damit die Schnittstelle nicht durch unnötig viele unterschiedliche URLs überladen wird, sieht der RESTful-Ansatz die Verwendung der verschiedenen HTTP-Verben vor, um mit den Ressourcen zu interagieren. Auf die Nutzung von Http-Verben wird im nächsten Abschnitt näher eingegangen. Zur Interaktion mit Ressourcen werden zwei Arten von URLs unterschieden, welche in Kombination mit HTTP-Verben verschiedene Aufgaben erfüllen. Zur Veranschaulichung sollen folgende zwei URLs dienen:

- <http://myRestService.de/Schedule>
- <http://myRestService.de/Schedule/123>

Es fällt auf, dass die beiden URLs sich bis auf das letzte Segment gleichen. Im ersten Fall wird die URI als *Collection URI* bezeichnet, da hiermit die Gesamtheit aller Trainingspläne angesprochen wird. Im zweiten Fall wird die ID eines Trainingsplans benutzt, um mit einer konkreten Trainingsplan-Ressource zu interagieren. Man spricht

---

<sup>2</sup>TILKOV u. a.: REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web, S. 26ff.

dabei von einer *Element URI*.<sup>3</sup> Beide können mit verschiedenen HTTP-Verben kombiniert werden.<sup>4</sup>

### Nutzung von HTTP-Verben

Um den Rahmen der Arbeit nicht zu übersteigen, wird hier nur auf vier meist verwendeten HTTP-Verben beschränkt:

Das Verb GET ruft eine Ressource vom Server ab, wobei diese nicht verändert wird. Bei Nutzung einer *Collection URI*, werden alle Einträge dieser Entität als Verbundstruktur abgerufen. Jedes Element der Struktur beinhaltet die *Element URI* auf das konkrete Element. Wird GET auf eine *Element URI* aufgerufen, wird das konkrete Objekt aufgerufen. Hierbei antwortet der Server, dem HTTP-Standard folgend, mit dem Status-Code *200 (OK)* bei erfolgreicher Suche oder *404 (Not Found)*, wenn keine Ressource gefunden wurde.

Das POST-Verb wird zur Erstellung neuer Inhalte verwendet. Bei Nutzung von *Element URIs* wird versucht, die ID für das neue Element zu benutzen. In der Regel wird das ID-Management aber auf dem Server implementiert, sodass eine *Collection URI* zur Erstellung von Elementen zum Einsatz kommt.

Mit dem HTTP-Verb PUT wird eine vorhandene Ressource geändert oder hinzugefügt. Obwohl es REST-konform wäre, eine Collection URI per PUT aufzurufen, wird dies selten implementiert, da der normale Anwendungsfall darin besteht, dass ein einzelnes Objekt geändert werden soll. Deshalb wird sich stattdessen auf *Element URIs* beschränkt. Ist eine Ressource mit der übergebenen ID nicht vorhanden, wird je nach Implementierung entweder ein neues Objekt mit der ID erstellt (*Statuscode 201 (Created)*) oder die Verarbeitung verweigert. Der Server gibt dann den Statuscode *400 (Bad Request)* oder *404 (Not Found)* zurück.

Das letzte HTTP-Verb, welches an dieser Stelle vorgestellt werden soll, ist DELETE. Wie der Name vermuten lässt, wird damit eine Ressource vom Server entfernt. Wie auch bei PUT wird in der Regel auf eine Implementierung von DELETE als *Collection URI* verzichtet, da sonst alle Einträge einer Entität gelöscht werden können. Im Er-

---

<sup>3</sup>KURTZ/WORTMAN: ASP.NET Web API 2: Building a REST Service from Start to Finish, S. 12ff.

<sup>4</sup>TILKOV u. a.: REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web, S. 26ff.

fallsfall wird mit dem Statuscode *200 (OK)* geantwortet und bei Fehlern mit *400 (Bad Request)* oder *404 (Not Found)*.<sup>5</sup>

### **Zustandslosigkeit**

Das zum Datenaustausch genutzte Protokoll HTTP ist statuslos. Das bedeutet, dass kein Kontext bei der Kommunikation besteht bzw. jede Kommunikation unabhängig von vor- oder nachherigen Verbindungen ist. Deshalb muss ein RESTful Web Service so implementieren werden, dass alle Informationen, welche für die Kommunikation notwendig sind, bei jeder Anfrage mitgesendet werden. Was vordergründig als Nachteil erscheint, ist ein wesentlicher Vorteil. Dadurch, dass jeder Request alle nötigen Informationen mitliefert, ist es nicht nötig, den Kontext der Kommunikation über mehrere Requests auf dem Server zu verwalten. Dadurch kann ein RESTful Web Service sehr leicht skaliert werden.<sup>6</sup>

### **Daten sind unabhängig von der Darstellung**

Das RESTful-Paradigma besagt, dass Daten losgelöst von einer Darstellung bereitgestellt werden. Darum ist ein RESTful Web Service so zu implementieren, dass der Client das gewünschte Datenformat anfragen kann. Bei Nutzung des Protokolls HTTP wird dies in der Regel über die Header-Eigenschaft *accept* realisiert, welche gewünschten Datenformate angibt. Werden dieses nicht vom Server unterstützt, werden die angeforderten Daten in einem Standard-Format zurückgegeben.<sup>7</sup>

## **6.2. Umsetzung der Komponenten**

In diesem Abschnitt wird beschrieben, wie der zuvor theoretisch beschriebene REST-Ansatz für das Projekt umgesetzt wurde. Der aus der Umsetzung entstandene Server besteht aus zwei Teilen: Der Datenbank und der Web API, welche jeweils gesondert vorgestellt werden.

---

<sup>5</sup>TILKOV u. a.: REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web, S. 26ff.

<sup>6</sup>Ebd., S. 26ff.

<sup>7</sup>Ebd., S. 26ff.

Die Web API bietet eine Schnittstelle zum Abrufen der Daten mittels HTTP. Diese wurde nach dem Design-Pattern *Model-View-ViewModel (MVVM)* aufgebaut. Die erzeugten Objekte, welche Tupel einer Datenbank-Relation darstellen, werden aus speziell dafür präparierten Model-Klassen erzeugt. Bevor diese Daten dann über die Web API bereitgestellt werden, werden sie vom Model in ein ViewModel übertragen. Hierbei wird, nach dem Grundgedanken des Separation of Concerns, klar zwischen den Models für die Datenbank und den ViewModels, welche die Web API benutzt, unterschieden.

### 6.2.1. Einbindung und Zugriff auf die Datenbank

Wie bereits in Kapitel 5.1 vorgestellt, wurde die Datenbank-Lösung *MSSQL* von *Microsoft* zur Datenhaltung gewählt. Dies hat den Vorteil, dass das *Microsoft Entity Framework*, welches sehr gut für die Nutzung mit einer Web API optimiert ist, als OR-Mapper genutzt werden kann. Dieser bietet das Design-Pattern *Code First*. Das bedeutet, dass anhand präparierter Model-Klassen die benötigten Relationen in der Datenbank automatisch erzeugt werden.<sup>8</sup>

An den folgenden Beispielen wird exemplarisch beschrieben, wie die Model-Klassen aufgebaut wurden und wie sich daraus die Struktur der Datenbank ergibt. Grundlage für die Model-Klassen ist das Interface *IEntity* (siehe Quellcode-Beispiel 6.1):

```
1 public interface IEntity<T>
2 {
3     T Id { get; set; }
4 }
```

**Quelltext 6.1:** Basisinterface für DB-Repräsentationen

Das Interface gewährleistet, dass jede Datenbank-Entität einen eindeutigen Schlüssel besitzt. Eine konkrete Implementierung für eine Model-Klasse sieht man im Quellcode-Beispiel 6.2, in der die Trainingspläne implementiert sind:

```
1 // Definiert einen Trainingsplan
2 public class Schedule: IEntity<int>
```

<sup>8</sup>DYKSTRA: Getting Started with Entity Framework 6 Code First using MVC 5.

```
3  {
4      public Schedule(int id, string name = "", string userId = "",
        ICollection<Exercise> exercises = null)
5      {
6          this.Id = id;
7          this.Name = name;
8          this.UserID = userId;
9          this.Exercises = exercises;
10     }
11
12     public Schedule(): this(-1){}
13
14     // DB ID
15     public int Id { get; set; }
16     // DisplayName des Trainingsplans
17     [Required]
18     public string Name { get; set; }
19     // Fremdschlüssel zum Nutzer (per Namenskonvention)
20     public string UserID { get; set; }
21     // Uebungen (per Namenskonvention)
22     public virtual ICollection<Exercise> Exercises { get; set; }
23 }
24 }
```

**Quelltext 6.2:** Modelklasse für Trainingspläne

Hierbei zeigt sich gut, was mit einer präparierten Model-Klasse gemeint ist. Über die Annotation *Required* wird definiert, dass die Eigenschaft *Name* zwingend bei Insert- und Update-Operationen gesetzt werden muss.

Gleichzeitig sieht man an diesem Beispiel, wie das Entity Framework über Namenskonventionen Verbindungen zwischen Entitäten auflöst. Auf Grund des Aufbaus der Klasse *Schedule* wird eine einwertige Fremdschlüssel-Beziehung zu der Model-Klasse *User* erzeugt, da folgende Bedingungen erfüllt sind:

- Die Klasse *User* besitzt eine Eigenschaft *ID* vom Datentyp *string*
- Die Klasse *Schedule* besitzt eine Eigenschaft *UserID* vom Datentyp *string*

Auch die Erstellung einer mehrwertigen Beziehung lässt sich aus dem Code-Beispiel 6.2 ablesen:

Da es eine Entität gibt, welche *Exercise* heißt und die Model-Klasse *Schedule* eine Verbundstruktur besitzt, welche *Exercises* heißt, wird implizit eine Verbindung zwischen den Relationen in der Datenbank angelegt.<sup>9</sup>

## 6.2.2. Web API

Die Umsetzung der REST-Schnittstelle wurde mit Hilfe des *Microsoft*-Frameworks *ASP.NET Web API 2* (kurz *Web API-Framework*) realisiert. Dieses ermöglicht es, Controller-Methoden zu erstellen, welche über definierte Routen per HTTP aufgerufen werden können. Hierbei wird die Umsetzung im Sinne des REST-Paradigmas durch vorhandene Funktionen unterstützt.<sup>10</sup>

Dies wird beispielhaft an der Methode aus Quellcode-Beispiel 6.3 gezeigt:

```

1 // Grants access to schedule data
2 [SwaggerResponse(HttpStatusCode.Unauthorized, "You are not allowed to receive
   this resource")]
3 [SwaggerResponse(HttpStatusCode.InternalServerError, "An internal Server error
   has occurred")]
4 [Authorize]
5 [RoutePrefix("api/schedule")]
6 public class SchedulesController : BaseApiController
7 {
8     // Create new Schedule for the logged in user
9     [SwaggerResponse(HttpStatusCode.Created, Type = typeof(ScheduleModel))]
10    [SwaggerResponse(HttpStatusCode.BadRequest)]
11    [Route("")]
12    [HttpPost]
13    public async Task<IHttpActionResult> CreateSchedule(ScheduleModel schedule)
14    {
15        if (ModelState.IsValid && !schedule.UserId.Equals(this.CurrentUserId))
16        {

```

<sup>9</sup>DYKSTRA: Getting Started with Entity Framework 6 Code First using MVC 5.

<sup>10</sup>KURTZ/WORTMAN: ASP.NET Web API 2: Building a REST Service from Start to Finish, S. 2ff.

```
17     ModelState.AddModelError("UserId", "You can only create schedules for  
        yourself");  
18 }  
19 if (!ModelState.IsValid)  
20 {  
21     return BadRequest(ModelState);  
22 }  
23  
24 var datamodel = this.TheModelFactory.CreateModel(schedule);  
25 await this.AppRepository.Schedules.AddAsync(datamodel);  
26 var result = this.TheModelFactory.CreateViewModel(datamodel);  
27 return CreatedAtRoute("GetScheduleById", new { id = schedule.Id },  
        result);  
28 }  
29 }
```

**Quelltext 6.3:** POST-Methode zur Erstellung eines Trainingsplans

Im Beispiel fällt auf, dass das *Web API-Framework* die Nutzung von Annotationen fördert: Das Routing kann durch die Annotationen *Route* (Zeile 11) an der Methode und *RoutePrefix* (Zeile 5) am gesamten Controller konfiguriert werden. Neben der Konfiguration der Route muss dem Framework noch mitgeteilt werden, welche HTTP-Verben in dieser Methode zulässig sind. Das *Web API-Framework* bietet hierfür pro Verb eine eigene Annotation. Im Codebeispiel 6.3 wird über die Annotation *HttpPost* (Zeile 12) festgelegt, dass nur POST-Requests durch diese Methode verarbeitet werden.<sup>11</sup>

Das Framework versucht die empfangenen Daten in einem ViewModel-Objekt zu kapseln und anschließend zu validieren. Die dafür genutzten Validatoren werden direkt im View-Model als Annotationen angegeben.<sup>12</sup> Die Klasse *EntryModel* (siehe Quellcode-Beispiel 6.4) zeigt die Nutzung von Validator-Annotationen in den Zeilen 7 und 11.

```
1 namespace fIT.WebApi.Models  
2 {
```

---

<sup>11</sup>WASSON: Attribute Routing in ASP.NET Web API 2.

<sup>12</sup>DERS.: Model Validation in ASP.NET Web API.



```

3  // Defines one entry from the server
4  public class EntryModel<T>
5  {
6      // Id of an entity
7      [Required(ErrorMessageResourceName = "Error_Required",
8          ErrorMessageResourceType = typeof(Resources))]
9      public T Id { get; set; }
10
11     // Name of an Entity
12     [Required(ErrorMessageResourceName = "Error_Required",
13         ErrorMessageResourceType = typeof(Resources))]
14     public string Name { get; set; }
15
16     // Url to receive this entity
17     public string Url { get; set; }
18 }

```

**Quelltext 6.4:** Basis-Model-Klasse

Schlägt die Validierung fehl, werden die Fehler mit dem passenden Statuscode zurückgegeben. Andernfalls werden die Daten per Factory-Klasse in ein Model konvertiert und per Repository-Klasse in der Datenbank persistiert. Anschließend wird dem ViewModel, im Sinne des REST-Gedankens, eine URL zur GET-Methode mit der ID des neu erstellten Objekts übergeben.

**Swagger als Entwurfshilfsmittel**

Da die Web API zur Entwicklung der Clients benötigt wurde, entstand schnell die Notwendigkeit einer Dokumentation des aktuellen Stands des Web Services.

Aus diesem Grund wurde *Swagger* in die Web API integriert. *Swagger* ist ein quelloffenes Framework zur Dokumentation von RESTful Web APIs, welche von vielen großen Konzernen genutzt wird.<sup>13</sup> Durch Nutzung des NuGet-Packages *Swashbuckle* konnte durch hinzufügen von Kommentaren und Annotationen eine vollständige und übersichtliche Dokumentation erstellt werden.<sup>14</sup> Die Abbildung 6.1 zeigt diese.

<sup>13</sup>SWAGGER: Swagger.

<sup>14</sup>JOUDEH: ASP.NET Web API Documentation using Swagger.

Da das Autorisierungsprotokoll *OAuth* in Version 2 (kurz: *OAuth2* siehe Kapitel 6.3.1) zum Durchführungszeitpunkt des Projekts noch nicht von *Swagger* unterstützt wurde, kann das Ausführen von API-Requests aus *Swagger* heraus nur für Methoden durchgeführt werden, für die keine Autorisierung des Nutzers benötigt wird. Die Dokumentation ist unter <http://fit-bachelor.azurewebsites.net/swagger> aufrufbar.

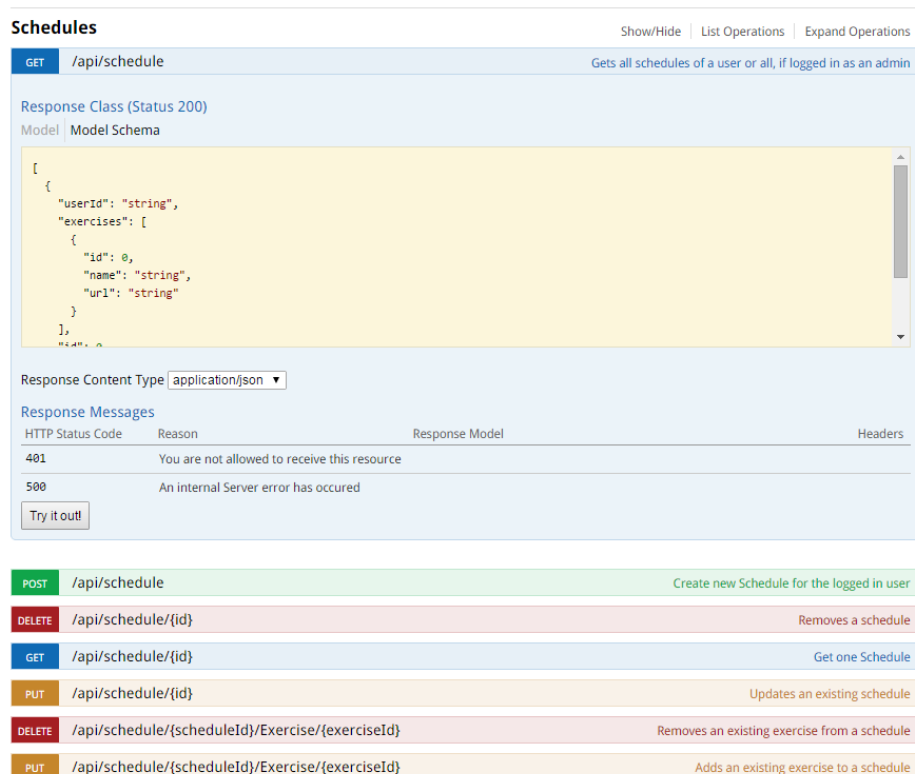


Abbildung 6.1.: Swagger UI der Web Api

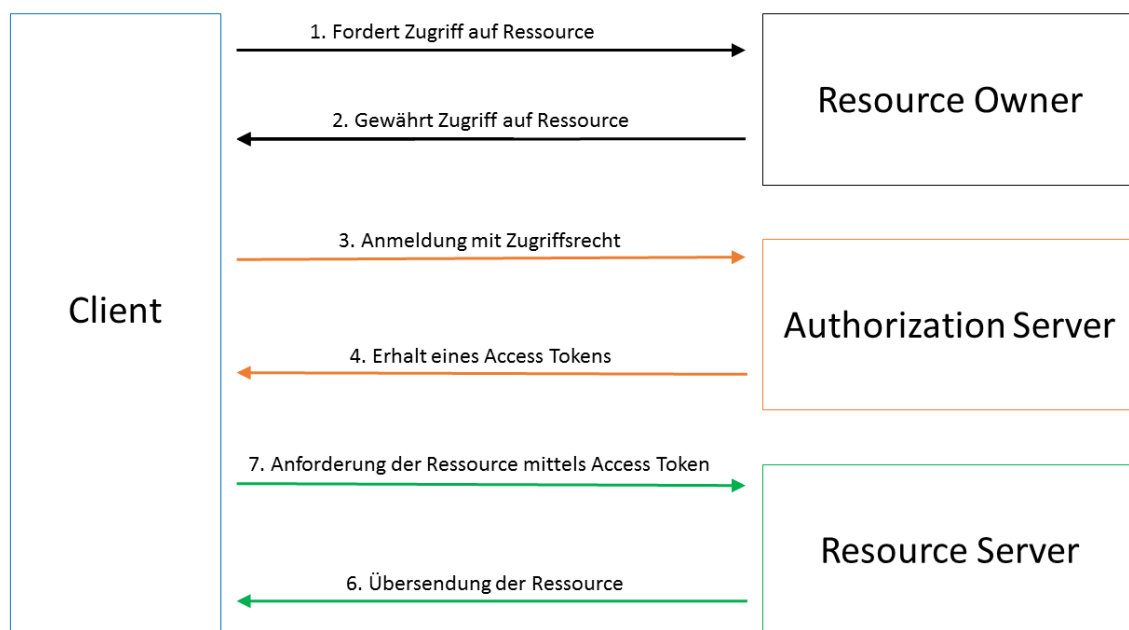
### 6.3. Einbindung von Authentifizierung & Autorisierung

Wie bereits in Kapitel 4.4 beschrieben, darf nicht jeder Nutzer auf alle Daten zugreifen. Um dies sicherzustellen, wurde ein Login-Mechanismus implementiert, welcher bekannte Nutzer authentifiziert. Da jedoch nicht alle authentifizierten Nutzer alle bereitgestellten Web API-Methoden benutzen dürfen, wurden auf Basis des *Role Ba-*

sed Access Control (RBAC) Rollen implementiert, welche den Nutzer zur Nutzung verschiedener Aufrufe autorisieren. Zur Umsetzung dieser Anforderungen wurde das Protokoll *OAuth2* implementiert.<sup>15</sup>

### 6.3.1. Das Protokoll *OAuth2*

OAuth2 ist ein Protokoll-Standard zur Authentifikation und zur Delegation von Zugriffsrollen. Die Struktur von OAuth2 kennt vier Instanzen, welche in diesem Vorgang miteinander kommunizieren. Diese sind *Client*, *Resource Owner*, *Authorization Server* und *Resource Server*.<sup>16</sup> Abbildung 6.3 beschreibt diesen Vorgang.



**Abbildung 6.2.:** Ressourcenzugriff durch OAuth2

#### Client

Der *Client* ist ein Endpunkt, welcher eine Ressource (beispielsweise Trainingspläne) abrufen möchte. In diesem Projekt stellen die Web- und die native App die Clients

<sup>15</sup>JOUDEH: Token Based Authentication using ASP.NET Web API 2, Owin, and Identity.

<sup>16</sup>STEYER/SOFTIC: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, S. 286.

dar. Diese kommunizieren jeweils mit den anderen Komponenten.

### Resource Owner

Der *Resource Owner* ist, wie der Name schon sagt, der Besitzer der geforderten Ressource. Der *Client* erfragt im ersten Schritt beim *Resource Owner* den Zugriff zu einer Ressource.

In diesem Projekt registriert sich der Nutzer an der Web API. Anschließend kann er unter seinem Account Daten (Trainingspläne und Trainings) anlegen. Diese angelegten Daten sind die geforderten Ressourcen. Da diese vom Nutzer selbst angelegt wurden, erhält er automatisch die Erlaubnis (*Grant*) zur Anfrage am *Authorization Server*.<sup>17</sup>

### Authorization Server

Der Nutzer meldet sich nun mit der erhaltenen Erlaubnis am *Authorization Server* an. Dieser hat Kenntnis über alle vorhandenen Nutzer und deren Rollen.<sup>18</sup> Bei erfolgreicher Anmeldung erhält der Nutzer ein kurzlebiges *Access-Token*, den Typen des *Access-Tokens*, dessen Ablaufdatum und ein langlebiges *Refresh-Token*. Das *Access-Token* wird im nächsten Schritt benutzt, um die gewünschte Ressource anzufordern. Das *Refresh-Token* wird benötigt, um ein neues *Access-Token* anzufordern. Die beiden Token-Arten werden nochmal genauer in Abschnitt 6.3.2 besprochen.<sup>19</sup>

### Resource Server

Der *Resource Server* enthält die geforderte Ressource. Ab diesem Kommunikationsschritt muss das *Access-Token* bei jeder Anfrage mitgesendet werden. Konkret passiert dies, indem der Header der Anfrage um den Schlüssel *authorization* erweitert wird.

Durch die strikte Trennung dieser Instanzen ist es ohne Weiteres möglich, dass unterschiedliche Systeme die jeweiligen Aufgaben übernehmen. Deshalb hat sich in

---

<sup>17</sup>JOUDEH: Implement OAuth JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1.

<sup>18</sup>DERS.: ASP.NET Identity 2.1 Roles Based Authorization with ASP.NET Web API.

<sup>19</sup>STEYER/SOFTIC: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, S. 287.

letzter Zeit etabliert, dass immer häufiger *Single-Sign-On*-Szenarien implementiert werden. Dabei muss sich der Nutzer nur an einer Stelle registrieren (z.B. Bei *Facebook* oder *Twitter*). Will der Nutzer nun auf eine andere Ressource zugreifen, kann der *Ressource-Server* ein *Access-Token* vom *Facebook*-Authorisierungsserver akzeptieren. Dies hat für den Nutzer den Vorteil, dass er sich nicht bei mehreren Seiten registrieren muss, sondern jedes mal Zugriff über den Authorisierungsserver mithilfe seiner *Credentials* erhält.<sup>20</sup>

### 6.3.2. JWT and Bearer Token

Sowohl das Access-Token als auch das Refresh-Token sind JSON Web Token (JWT). Diese werden von Instanzen des OAuth2 zur Kommunikation benutzt. Es sind codierte und meistens auch signierte Repräsentationen von Daten. Zur genaueren Betrachtung des Aufbaus, wird folgend ein Access-Token näher beschrieben. Es besteht aus 3 Teilen, welche mit einem Punkt voneinander getrennt sind. Die Daten selber sind mithilfe des Kodierungsverfahren *base64* verschlüsselt.<sup>21</sup> Die Bestandteile sind:

- **Header**

Hier wird der Typ des Tokens und der Algorithmus, welcher für die Verschlüsselung benutzt wurde, angegeben.

- **Payload**

Die zu übermittelnden Daten werden als JSON-Objekt bereitgestellt. Das Objekt enthält sowohl die Informationen für die Kommunikation, wie beispielsweise den Nutzernamen und die Rollen, als auch Meta-Daten über das Token (z.B. das Ablauf-Datum).

- **Signatur**

Damit gewährleistet ist, dass die Daten nicht verändert wurden, werden Sie mit einem Client-Secret verschlüsselt. Dies bedeutet aber auch, dass der Server jeden Client kennen muss, welcher sich beim Authorization Server anmelden will.

Da es sich bei diesem Projekt um einen Prototyp handelt, wurde die Implementierung der Client-Verwaltung nicht durchgeführt, da es für den Ablauf nicht

---

<sup>20</sup>Ebd., S. 294.

<sup>21</sup>Ebd., S. 289.

zwingend benötigt wird. Der Server lässt alle gültigen Access-Token und alle bekannten Refresh-Token zu. Im produktiven Einsatz müsste diese Komponente dringend nachträglich implementiert werden, da sonst eine Sicherheitslücke entsteht.<sup>22</sup>

Wie bereits im Abschnitt zum *Authorization Server* (siehe 6.3.1) beschrieben, wird für das Access-Token eine recht kurze- und für das Refresh-Token eine sehr lange Lebenszeit gewählt. Dies hat zwei Vorteile:

Das *Access-Token* wird bei Requests an den Server mitgesendet. Sollte das Token von Dritten abgefangen werden, können diese nur für kurze Zeit im Namen des Nutzers Aktionen durchführen. Das Abgreifen eines solchen Tokens wird im produktiven Gebrauch, durch zusätzliche Sicherheitsmaßnahmen, wie die Nutzung des Protokolls *Hyper Text Transfer Protocol Secure (HTTPS)* erschwert.

Da das Refresh-Token ausschließlich zum Erneuern des Access-Tokens benutzt wird, ist die Gefahr, dass es abgefangen wird wesentlich geringer, wodurch die lange Lebensdauer vertretbar ist.

Außerdem bleiben durch die kurze Lebensdauer des Access-Tokens die Daten immer aktuell. Sollte sich an den Daten des Nutzers etwas ändern (z.B. wird eine Rolle hinzugefügt oder entzogen) wird diese Änderungen beim nächsten Abrufen eines *Access-Tokens* in die Payload codiert.<sup>23</sup> Somit ist immer gewährleistet, dass der Nutzer nur die Funktionalität nutzt, für die er auch autorisiert ist.

### 6.3.3. Zugriffsbeschränkung per CORS

Im vorherigen Abschnitt wurden Maßnahmen beschrieben, durch die nur autorisierte Nutzer an geschützte Daten herankommen. Mit *Cross-origin resource sharing (CORS)* wird ein weiterer Mechanismus vorgestellt, welcher den Zugriff auf die Daten per *Asynchronous JavaScript and XML (AJAX)* beschränkt.

Um den Nutzer davor zu schützen, dass eine Webseite im Hintergrund Daten von anderen Quellen nachlädt, ist in jedem Browser eine *Same-Origin-Policy* implementiert. Diese besagt, dass nur Daten aus der gleichen Domäne, aus der der *AJAX*-Aufruf abgesetzt wurde, abgerufen werden dürfen.

Häufig ist es trotzdem nötig, auf fremden Domänen zuzugreifen. Deshalb wurden

---

<sup>22</sup>ATLASSIAN: Understanding JWT.

<sup>23</sup>JOUDEH: Enable OAuth Refresh Tokens in AngularJS App using ASP .NET Web API 2, and Owin.

schnell Hilfskonstrukte, wie das Vorgehensmodell *JSONP*, eingeführt. Da diese jedoch von vielen Entwicklern als nicht elegant empfunden wurden,<sup>24</sup> wurde mit CORS ein standardisierter Weg entwickelt, um Daten von fremden Domänen abzurufen.

Hierbei wird beim Server eine Liste an gültigen Domänen für eine domänenübergreifende Anfrage hinterlegt. Soll nun vom Browser eine Anfrage an den Server gesendet werden, wird über das HTTP Verb unterschieden, ob durch diese Anfrage eine Resource auf dem Server verändert wird. Dies geschieht bei PUT, DELETE und POST, wobei letzteres eine Ausnahme bildet. Werden per POST Daten in einem Format übermittelt, welches beim Absenden eines Formulars genutzt wird (z.B. *application/x-www-form-urlencoded*), wird die Anfrage wie ein nicht-ändernder Aufruf behandelt.

Wenn nun eine Datenänderung im Sinne von CORS durch den Aufruf angestoßen wurde oder wenn der Aufruf zusätzliche Schlüssel im Header enthält, wird vor der Ausführung ein *Preflight* gesendet. Dies ist eine *glsOPTIONS*-Anfrage, welche genutzt wird, um die Durchführung der bevorstehenden Anfrage zu validieren. Enthält die Antwort im Header nicht den Schlüssel *Access-Control-Allow-Origin* mit der aufrufenden Domäne, wird vom Browser ein Fehler erzeugt. Andernfalls wird die Abfrage an den Server gesendet.<sup>25</sup> Dadurch ist gewährleistet, dass nur berechtigte Clients anfragen an den Server senden. Es wurde auf weitere Implementierung von Polyfills verzichtet, da CORS bereits in allen modernen Browsern genutzt werden kann.<sup>26</sup>

## 6.4. Testen der Server-Funktionalität

Die erwartete Funktionsweise des Servers ist Grundvoraussetzung für die Umsetzung der Clients. Um diese zu bewerkstelligen wurde im Rahmen dieser Arbeit die *ManagementApi* entwickelt. Dies ist eine portable DLL, welche alle Anfragen an den Server in Methoden kapselt. Hierbei wurde bei der Erstellung der DLL darauf geachtet, dass sie sowohl in klassischen Testprojekten als auch zur Umsetzung der nativen App genutzt werden kann. Darüber hinaus war ein Design-Ziel, dass alle Methoden auch asynchron aufrufbar sind, damit Nutzer der DLL in der Erstellung ihres Programmblaufes größere Flexibilität besitzen.

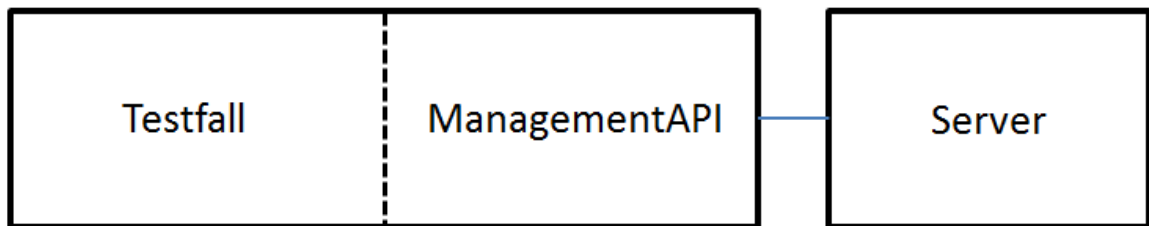
---

<sup>24</sup>STEYER/SOFTIC: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, S. 102.

<sup>25</sup>Ebd., S. 102.

<sup>26</sup>CANIUSE.COM: Can I Use: Cross-Origin Resource Sharing.

Jeder Testfall Benutzt die ManagementApi als Schnittstelle zum Server, um verschiedene Anwendungsfälle zu prüfen. Abbildung beschreibt den schematischen Aufbau der Tests. Die erzeugten Methoden wurden iterativ entwickelt und direkt getestet.



**Abbildung 6.3.:** Aufbau eines Testfalls

Hierbei wurden zur Verifikation der Funktionalität ausschließlich Positivtests erstellt. So ist sichergestellt, dass jede Methode der *ManagementApi* die gewünschten Aktionen auf dem Web Service durchführt.

Der nachfolgende Codeausschnitt 6.5 zeigt beispielhaft die Entwicklung eines Testfalls:

```
1 [TestMethod]
2 public void UpdateCurrentUserData()
3 {
4     const string NEWNAME = "TestUser";
5
6     using (var service = new ManagementService(ServiceUrl))
7     using (IManagementSession session = service.LoginAsync(USERNAME,
8         PASSWORD).Result)
9     {
10         var data = session.Users.GetUserDataAsync().Result;
11         Assert.AreEqual(USERNAME, data.UserName);
12
13         try
14         {
15             data.UserName = NEWNAME;
16             session.Users.UpdateUserDataAsync(data).Wait();
17
18             data = session.Users.GetUserDataAsync().Result;
19             Assert.AreEqual(NEWNAME, data.UserName);
20         }
21     }
22 }
```



```
19     }
20     finally
21     {
22         data.UserName = USERNAME;
23         session.Users.UpdateUserDataAsync(data).Wait();
24     }
25 }
26 }
```

**Quelltext 6.5:** Implementierung des Tests 'Nutzer kann eigene Daten anpassen'

Dadurch, dass die Klassen *ManagementService* und *ManagementSession* das Interface *IDisposable* implementieren, kann für jeden Test unabhängig eine neue Session erstellt werden, in der der Test läuft. Ist der *using*-Block vollständig durchlaufen, wird die *Dispose*-Methode aufgerufen, welche die verwendeten Ressourcen wieder freigibt (siehe Zeile 6f.).



# 7. Realisierung des Clients als native App

Dieses Kapitel widmet sich der Implementierung der nativen Applikation. Im Kapitel 4 wurde eine grobe Übersicht zu der Umsetzung und der Funktionsweise dieser App gegeben, die nun konkretisiert wird. Dabei werden nachfolgend die verwendeten Komponenten und Techniken erläutert und deren Zusammenhänge dargestellt.

## 7.1. Allgemeine Funktionsweise einer Android-App

Grundlegend für die Entwicklung einer Android-App ist das Wissen über die Basis des Systems, auf dem entwickelt wird. Bei dem Betriebssystem Android handelt es sich um eine Art eines monolithischen Multiuser-Linux-Systems.<sup>1</sup> Dieses Betriebssystem stellt die Hardwaretreiber zur Verfügung und führt die Prozessorganisation, sowie die Benutzer- und Speicherverwaltung durch.<sup>2</sup>

Jede Applikation wird in einem eigenen Prozess gestartet. In diesem Prozess befindet sich eine Sandbox, die eine virtuelle Maschine mit der Applikation ausführt. Die Kommunikation aus der Sandbox heraus kann nur über Schnittstellen des Betriebssystems geschehen. Somit kann eine Applikation nur auf zugewiesene und freigegebene Ressourcen im System zugreifen. Ein weiterer Vorteil dieser internen Architektur liegt in der Robustheit des Systems. Wenn eine Applikation durch Fehler terminiert, wird nur der zugewiesene Prozess beendet und das Betriebssystem bleibt von diesem Problem unberührt.<sup>3</sup>

---

<sup>1</sup>OPEN HANDSET ALLIANCE: Application Fundamentals.

<sup>2</sup>BECKER/PANT: Android 2 - Grundlagen und Programmierung, S. 19ff.

<sup>3</sup>OPEN HANDSET ALLIANCE: System Permissions.

Android-Applikationen werden in der Programmiersprache Java geschrieben, mit einem Java-Compiler kompiliert und dann von einem Cross-Assembler für die entsprechende Virtuelle Maschine (VM) aufbereitet. Das Produkt ist ein ausführbares Android Package (.apk).<sup>4</sup>

## 7.2. Komponenten einer Android-App

Im Folgenden werden die Android-Komponenten, welche für die Umsetzung relevant sind, genauer betrachtet.

### 7.2.1. User Interfaces

*User Interfaces* sind die Bildschirmseiten einer Android-Applikation. Über diese Seiten wird die Benutzerinteraktion geführt. Das *User Interface* besteht aus zwei Arten von Elementen. Zum einen aus *Views*, die es ermöglichen direkte Interaktionen mit dem Benutzer zu führen. Zu nennen sind dabei *Buttons*, Textfelder und Checkboxes. Zum Anderen werden *View Groups* verwendet, um *Views* sowie andere *View Groups* anzuordnen.<sup>5</sup>

Das *User Interface Layout* ist durch eine hierarchische Struktur gekennzeichnet. Zum Anlegen einer solchen Struktur gibt es verschiedene Möglichkeiten. Zum einen kann man ein *View*-Objekt anlegen und darauf die Elemente platzieren. Aus Gründen der Performance und der Übersicht ist die Möglichkeit einer XML-Datei jedoch zielführender. Aus den Knoten der erstellten Datei werden zur Laufzeit *View*-Objekte erzeugt und angezeigt. Die erzeugten *User Interface Layouts* werden unter *res/layout* im Android-Betriebssystem hinterlegt. Des Weiteren können Ressourcen in den *User Interface Layouts* verwendet werden. Unter Ressourcen versteht man Elemente, die zum Verzieren von Oberflächen verwendet werden können. Darunter fallen beispielsweise Grafiken oder *Stylesheets*, die über den jeweiligen Ressourcen-Schlüssel aufgerufen und verwendet werden.<sup>6</sup>

---

<sup>4</sup>OPEN HANDSET ALLIANCE: Application Fundamentals.

<sup>5</sup>BECKER/PANT: Android 2 - Grundlagen und Programmierung, S. 40f.

<sup>6</sup>OPEN HANDSET ALLIANCE: User Interface.

### 7.2.2. Activities

*Activities* gehören zu den App-Komponenten, da sie ein grundsätzlicher Bestandteil einer Applikation sind. Es gibt im Normalfall mehrere *Activities* in einer App.

Die eigentliche Aufgabe liegt in der Bereitstellung eines Fensters, das dann auf den Bildschirm, der für die App vom Betriebssystem bereitgestellt wird, gelegt wird. Anschließend können Benutzerinteraktionen vom Fenster angenommen werden.<sup>7</sup> Das Fenster wird mit Hilfe von *SetContentView()* aufgerufen. Zur Benutzerinteraktion werden dann die bereits vorgestellten *View*-Elemente verwendet. In der *Activity* werden folgend die Verarbeitung und Auswertung der Eingaben durchgeführt.

In jeder Applikation muss es eine *MainActivity* geben, die beim Start der Applikation vom Android-Betriebssystem gestartet wird. Zudem muss eine *Activity* im AndroidManifest mit dem Attribut *Launcher* versehen werden, um diese dann als Einstiegspunkt in das Menü des Betriebssystems zu setzen. Dabei ist es empfehlenswert, dass dieselbe *Activity* sowohl das Main- als auch Launcher-Attribut erhält, da es sinnvoll ist, dass die zuerst aufgerufene Seite einer App auch gleichzeitig die Startseite ist.

Das Manifest liegt im Wurzelverzeichnis der App und stellt dem Betriebssystem wichtige Informationen der Applikation zur Verfügung. Dieses Manifest wird vor Ausführung der App analysiert und ausgewertet. Darin kann beispielsweise festgelegt werden, welche Komponenten oder anderen Applikationen auf entsprechende *Activities* zugreifen dürfen. Wenn eine *Activity* nicht von außerhalb der App erreichbar sein soll, sollte kein sogenannter *Intent*-Filter gesetzt werden.

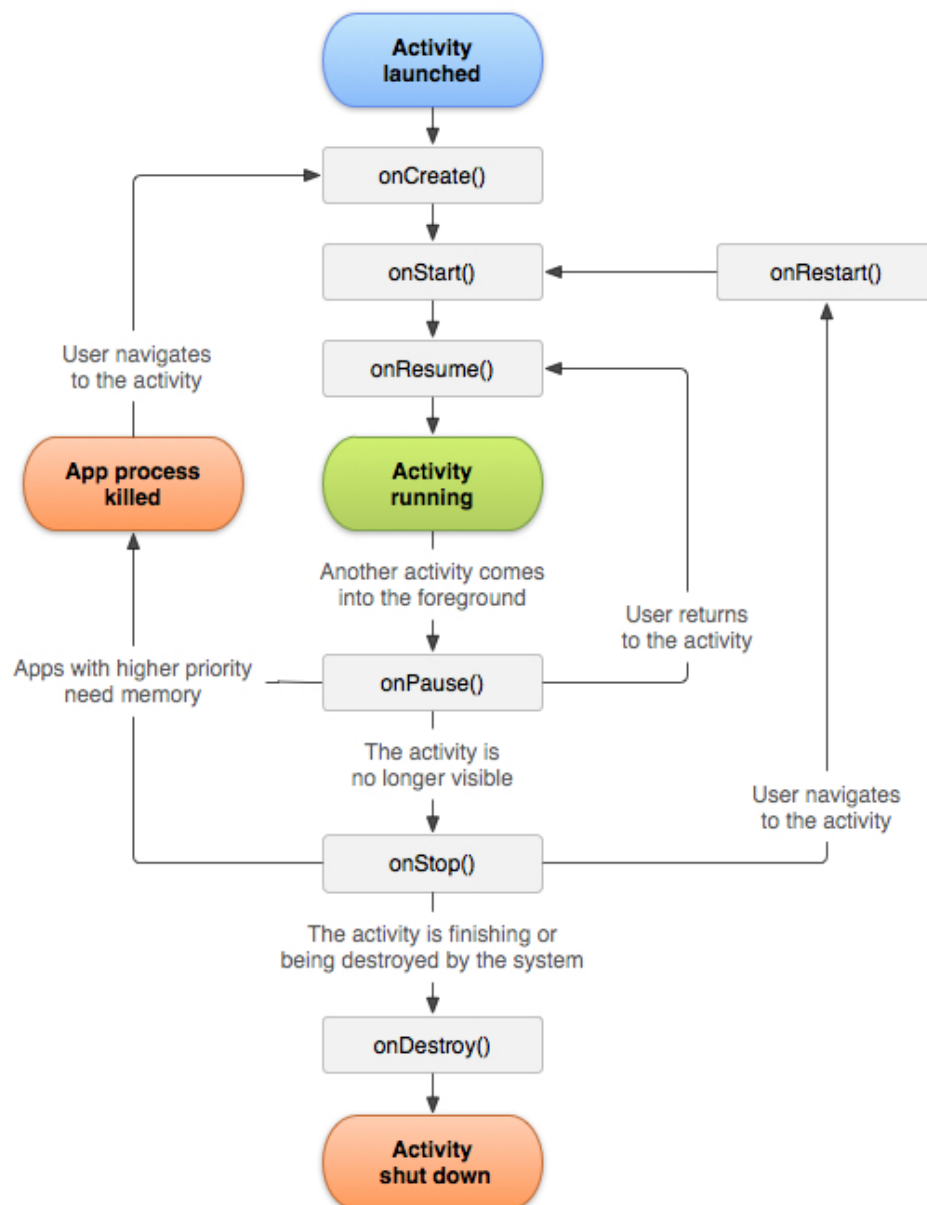
Da eine App normalerweise aus mehreren *Activities* besteht, müssen diese gestartet werden und untereinander kommunizieren. *Activities* starten sich gegenseitig, weshalb der Aufruf einer *Activity* aus einer anderen erfolgt. Um eine neue *Activity* starten zu können, ist ein *Intent* von Nöten.<sup>8</sup> Ein *Intent* ist ein Nachrichtenobjekt innerhalb von Android, welches zur Kommunikation zwischen App-Komponenten verwendet wird. In diesem Fall zwischen zwei *Activities*. Zur Ansteuerung benötigt es den Namen der zu startenden Komponente, um eine Verbindung dorthin aufbauen zu können, und eine *Action*, die ausgeführt werden soll. Zudem können Daten übergeben werden, die anschließend als Datenpakete mit dem Aufruf der Komponente mitgegeben werden. Diese Daten sind dann in der gestarteten Komponente aus dem dort vorhandenen Intent auslesbar. Zusätzlich gibt es die Möglichkeit Aktionen vom

---

<sup>7</sup>BECKER/PANT: Android 2 - Grundlagen und Programmierung, S. 40.

<sup>8</sup>Ebd., S. 135ff.

Betriebssystem ausführen zu lassen. Beispielsweise kann man einen *Intent* mit der Aktion zum Starten des E-Mail-Programms übergeben. Daraufhin wird die Applikation geöffnet, die im Betriebssystem als Standard-E-Mail-Programm hinterlegt ist.<sup>9</sup>



**Abbildung 7.1.:** Android Activity-Lifecycle

Quelle: <https://developer.android.com/guide/components/activities.html>

<sup>9</sup>OPEN HANDSET ALLIANCE: Intents and Intent Filters.

Eine *Activity* kann drei Status in einem *Lifecycle* einnehmen. Zum einen kann die *Activity* im Status *Resumed* - oft auch *Running* genannt - sein, momentan im Vordergrund der App sein und die Interaktionen entgegennehmen. Des Weiteren kann eine *Activity* pausieren, wenn eine andere Oberfläche im User-Fokus steht. Dabei ist die *View* der betrachteten *Activity* jedoch immer noch teilweise sichtbar, da die darüberliegende *View* zum Beispiel nicht den gesamten Bildschirm in Anspruch nimmt. Anders verhält es sich, wenn die *View* der betrachteten *Activity* komplett überdeckt ist. Dann befindet sich die *Activity* nämlich im Status *Stopped*. Abbildung 7.1 zeigt die möglichen Übergänge zwischen den verschiedenen Status.<sup>10</sup>

Essentiell ist die *OnCreate*-Methode, in der Initialisierungen durchgeführt werden sollten und die *View* gestartet wird. Kehrt der Benutzer nach einer Bearbeitungspause zu der aktuellen *Activity* zurück, wird *OnResume* ausgeführt.

Die *OnStop*-Methode sollte zur Persistierung von Daten verwendet werden und die darauf folgende Methode *OnDestroy* alle Aufrufe eines Destruktors enthalten.<sup>11</sup>

Die Methoden sollten aus Performancegründen jedoch minimal und agil gehalten werden.

### 7.2.3. Services

*Services* sind, genauso wie *Activities*, App-Komponenten, die zu den Grundbausteinen einer Android-App gehören. *Services* unterscheiden sich jedoch hinsichtlich ihrer Aufgaben stark von *Activities*. So sind sie dazu da, Aufgaben im Hintergrund zu erledigen. Zudem besitzen sie keine zugehörige *View*, sondern werden von anderen App-Komponenten, wie beispielsweise einer *Activity* gestartet.<sup>12</sup> Sie laufen im *Main-Thread* des Prozesses der aufrufenden Komponente. Ein *Service* erstellt weder einen eigenen *Thread*, noch einen eigenen Prozess zur Abarbeitung der Aufgaben.<sup>13</sup> Diese Eigenschaft der *Services* sollte vom Entwickler bedacht werden. Denn daraus kann man ableiten, dass rechenintensive Aufgaben in einem explizit gestarteten *Thread* abgearbeitet werden sollten, um Fehler der Art *Application Not Responding (ANR)* zu vermeiden und die Benutzeroberfläche nicht unnötig zu verlangsamen.

Ein weiterer Vorteil ist, dass *Services* Aufgaben auch dann noch ausführen können,

---

<sup>10</sup>DERS.: *Activities*.

<sup>11</sup>BECKER/PANT: *Android 2 - Grundlagen und Programmierung*, S. 289.

<sup>12</sup>Ebd., S. 163.

<sup>13</sup>OPEN HANDSET ALLIANCE: *Services*.

wenn die App, zu der sie gehören, geschlossen wurde. So können noch nicht abgeschlossene *Up-* oder *Downloads* noch beendet werden oder das Abspielen von Musik bei ausgeschaltetem Bildschirm fortgeführt werden.<sup>14</sup>

Bei Android wird grundsätzlich zwischen zwei Arten von *Services* unterschieden. Zum einen gibt es *Started-Services*, die durch eine App-Komponente mit dem Befehl *startService()* gestartet werden. Grundsätzlich ist dieser Aufruf uneingeschränkt von allen App-Komponenten möglich, soweit die Einstellungen im Android-Manifest diese zulassen. Weiterhin laufen *Started-Services* im Hintergrund der App weiter, auch wenn die Komponente, die den *Service* gestartet hat, zerstört oder beendet wurde. Deshalb führt diese Art des *Services* im Normalfall eine Aufgabe aus und stoppt sich anschließend nach der Fertigstellung selbst.

Auf der anderen Seite gibt es *Bound-Services*, die durch einen Aufruf von *bindService()* einer anderen App-Komponente gestartet werden. In diesem Schritt verbinden sich die Komponente und der *Service* über eine Art *Client-Server Interface*, das zur Kommunikation bereitgestellt wird. Dieses *Interface* ist vom Typ *IBinder* und sorgt für den Austausch von *Requests* und *Results*. Eine weitere Besonderheit eines *Bound-Services* besteht darin, dass der *Service* nur so lange besteht, wie mindestens eine Komponente an diesen gebunden ist. Natürlich ist es möglich, dass sich mehrere Komponenten gleichzeitig an diesen *Service* binden können. Löst sich jedoch die letzte Komponente wieder, wird der *Service* zerstört.<sup>15</sup>

Zum Erstellen eines *Services* muss von der Klasse *Service*, oder davon abgeleitete Klassen, geerbt werden. Danach müssen die vorgegebenen Methoden überschrieben werden, denn *Services* besitzen, genauso wie *Activities*, einen Lebenszyklus. Dabei muss jedoch wieder zwischen den beiden Arten von *Services* unterschieden werden.<sup>16</sup>

Die umrahmenden Methoden (grau dargestellte Elemente in der Abbildung 7.2 außerhalb des gelben Bereichs) sollten, ähnlich zu den Methoden der *Activities*, für die Initialisierung und Freigabe der verwendeten Komponenten sorgen. *onStartCommand()* wird immer dann aufgerufen, wenn der *Service* wieder von einer Komponente aufgerufen wird. Dann befindet er sich im Zustand *Running* und führt die ihm zugewiesenen Aufgaben durch. Wenn der *Service* zerstört wird, sei es durch Speichermangel des Endgeräts, das Beenden durch eine Komponente oder den *Service* selbst, wird

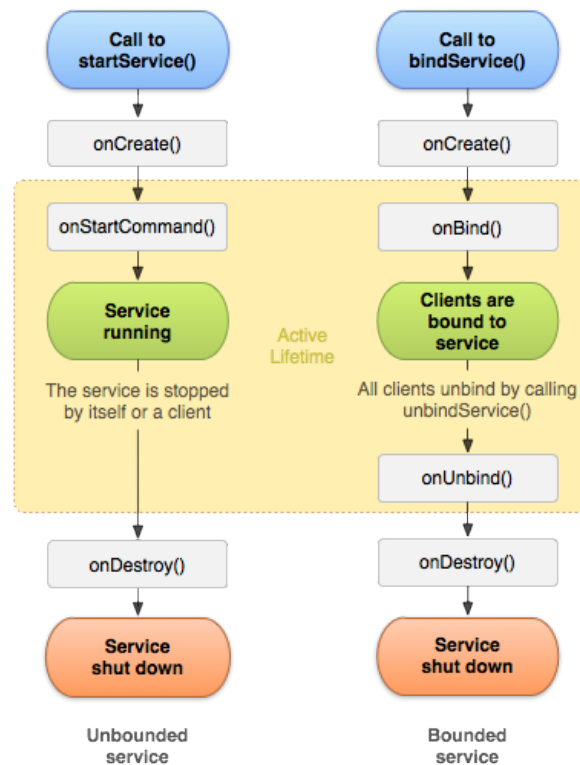
---

<sup>14</sup>BECKER/PANT: Android 2 - Grundlagen und Programmierung, S. 161f.

<sup>15</sup>OPEN HANDSET ALLIANCE: Services.

<sup>16</sup>BECKER/PANT: Android 2 - Grundlagen und Programmierung, S. 168f.





**Abbildung 7.2.:** Android Service-Lifecycle

Quelle: <https://developer.android.com/guide/components/services.html>

`onDestroy()` ausgeführt.

*Bound-Services* verhalten sich sehr ähnlich zu den *Started-Services*, sind jedoch von dem Binden und Lösen von Komponenten abhängig und führen demnach die entsprechenden Methoden zur Ausführung ihrer Aktionen aus.<sup>17</sup>

#### 7.2.4. Prozesse und Threads

Sobald eine Applikation gestartet wird, und keine Komponenten daraus bereits laufen, wird vom Android-Betriebssystem ein neuer Prozess mit einem dazugehörigen *Main-Thread* erzeugt. Standardmäßig werden alle Operationen dieser App in diesem Prozess und diesem *Thread* ausgeführt. Laufen Teile einer Applikation jedoch noch im Hintergrund, wie es bei Services möglich ist (siehe 7.2.3), und die App wird vom Benutzer erneut gestartet, so wird diese Komponente in dem noch bestehenden Prozess und *Thread* eingepflegt.

<sup>17</sup>OPEN HANDSET ALLIANCE: Services.

Es gibt jedoch auch die Möglichkeit verschiedene App-Komponenten auf mehrere Prozesse zu verteilen. Dazu genügt ein Eintrag im Android-Manifest. Dadurch ist es dann auch möglich Komponenten verschiedener Applikationen in einem Prozess laufen zu lassen. Voraussetzung dafür ist, dass diese beiden Applikationen mit demselben Zertifikat generiert wurden und dieselbe Linux *UserID* besitzen.<sup>18</sup>

Prozesse können aber auch durch das Betriebssystem zerstört werden, wenn die Geräte-Ressourcen zum Beispiel erschöpft sind und neue freigegeben werden müssen. Hierzu gliedert Android die Prozesse in eine Hierarchie ein und beendet die Prozesse, die zum Beispiel vom Benutzer seit längerer Zeit nicht mehr verwendet wurden oder keinen direkten Kontakt zur aktuellen Anzeige besitzen.

Der angesprochene *Main*- oder auch *UI-Thread* beim Starten einer App, ist der Hauptakteur für die Kommunikation mit dem Betriebssystem. So werden alle Aufrufe an die Komponenten des Android UI Toolkit über diesen *Thread* abgewickelt. Demnach müssen über diesen *Thread* alle *Callback*-Methoden von Systemeigenschaften, wie *OnClick()*, darin bearbeitet werden. Daraus ergibt sich, dass aufwendige Aufgaben, die zum Beispiel Netzwerkverbindungen verwenden, in andere *Threads* verlagert werden sollten, um dem Benutzer eine Oberfläche ohne Wartezeiten zu ermöglichen. Die einzige Einschränkung dabei ist, dass niemals von einem anderen *Thread*, als dem *UI-Thread*, auf die *Android UI Toolkits* zugegriffen werden darf. Dies muss bei der Implementierung bedacht werden.<sup>19</sup>

Asynchrone *Tasks* können verwendet werden, um die Aufgaben außerhalb des *UI-Threads* ausführen. Auf das Ergebnis dieser Ausführungen kann dann wieder zugegriffen werden.

### 7.2.5. SQLite

SQLite ist eine in sich geschlossene und serverlose Structured Query Language (SQL)-Datenbank. Sie besteht aus einer *In-Process*-Bibliothek, die es ermöglicht, eine Datenbank ohne eigenen Server-Prozess zu betreiben. Dabei liegt die Datenbank mitsamt aller Tabellen, *Views* und *Trigger* in einer einzigen Datei vor. Diese Datei ist zudem so konzipiert, dass sie plattformübergreifend zwischen 32- und 64-Bit-Systemen kopiert werden kann. Weitere Vorteile von SQLite liegen in der sehr

---

<sup>18</sup>OPEN HANDSET ALLIANCE: Processes and Threads.

<sup>19</sup>BECKER/PANT: Android 2 - Grundlagen und Programmierung, S. 160f.

sparsamen Speicherung der Daten und der, durch die gemeinfreie Lizenz, große Unterstützung durch Drittanbieter-Programmen. So gibt es für alle gängigen mobilen Systeme eine meist schon integrierte Unterstützung von SQLite-Datenbanken. Android unterstützt diese Datenbankart als präferierte Datenhaltung.<sup>20</sup>

## 7.3. Was ist Xamarin Platform?

Mit Xamarin Platform können plattformunabhängige Programme unter anderem in C# programmiert werden.<sup>21</sup> Es ist ein Produkt der Firma Xamarin, die ihren Sitz in San Francisco hat. Diese Firma entwickelt Software für die Erstellung von nativen Apps auf Basis des *Open Source*-Projekts Mono. Mono seinerseits hat mehrere Vorteile:

- **Popularität**

Es kann auf die Erfahrung von Millionen C# -Entwicklern zurückgegriffen werden.<sup>22</sup>

- **Höhere Programmiersprache**

Es können die Vorteile von höheren Programmiersprachen verwendet werden. Zu nennen sind dabei besonders *Threading*, automatische Speicherverwaltung und *Reflection*.

- **Klassenbibliotheken**

Die Verwendung von bestehenden Klassenbibliotheken erleichtern das Umsetzen komplexer Aufgaben.

- **Cross-Platform**

Die fertiggestellte Software kann auf allen gängigen Systemen verwendet werden.

Vorher war es nötig, drei Applikationen für die verbreitetsten mobilen Betriebssysteme, zu entwickeln. Nun wird die Funktionalität plattformunabhängig umgesetzt und darauf plattformspezifische Anpassungen gemacht (siehe Abbildung 7.3). Die *UI-Guidelines* des jeweiligen Betriebssystems müssen weiterhin eingehalten werden.<sup>23</sup>

---

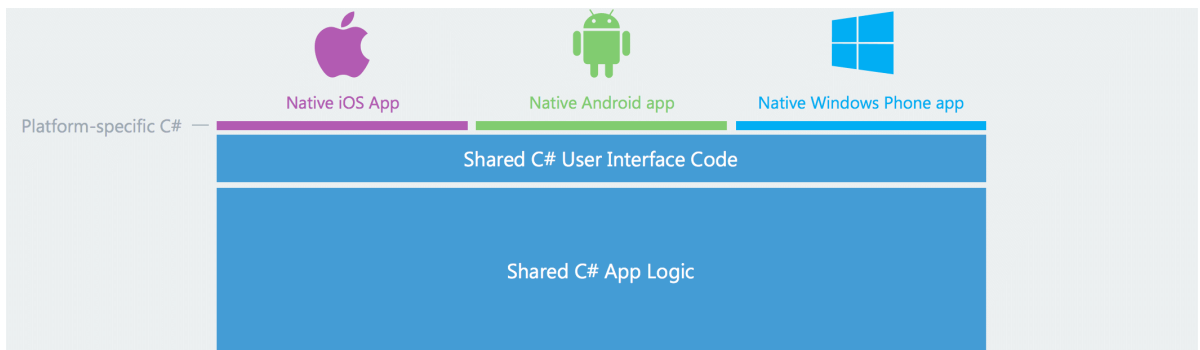
<sup>20</sup>Ebd., S. 226f.

<sup>21</sup>XAMARIN INC.: Cross-Platform - Application Fundamentals.

<sup>22</sup>DERS.: Create native iOS, Android, Mac and Windows apps in C #.

<sup>23</sup>Ebd.

### 7.3.1. Multiplattform-Unterstützung



**Abbildung 7.3.:** Xamarin Platform

Quelle: <http://xamarin.com/platform>

Durch Xamarin Platform ist es möglich alle Funktionalitäten der gewünschten Betriebssysteme in vollem Umfang zu verwenden. Diese Tatsache liegt daran, dass erstellte Projekte in die nativen Sprachen des Systems überführt und dann normal kompiliert werden. Durch die Nutzung der Standard-Steuerelemente eines jeden Betriebssystems sorgt dafür, dass die Benutzer keinen Unterschied zu einer App erkennen können, die ausschließlich für ein Betriebssystem entwickelt wurde. Auch plattformspezifische Funktionen können verwendet werden. Zudem werden alle Vorteile der Programmiersprache C# ausgenutzt.<sup>24</sup> So ist die Unterstützung von asynchronen Funktionen in dieser Sprache vollständig umgesetzt. Des Weiteren können *Shared Projects* zur Entwicklung verwendet werden, sowie Portable Class Library (PCL)s und *NuGet*-Pakete eingebunden werden, um den Funktionsumfang erweitern zu können.<sup>25</sup>

### 7.3.2. Besonderheiten der Android-Umsetzung

Bei der Entwicklung einer Android-App mit Hilfe von Xamarin Platform vereint man die Vorteile zweier Systeme. Zum einen hat man den Vorteil der Reife von *Visual Studio* in Kombination mit C#, zum anderen kann man alle Besonderheiten der Android-Entwicklung einbeziehen und verwenden.

<sup>24</sup>XAMARIN INC.: Create native iOS, Android, Mac and Windows apps in C #.

<sup>25</sup>DERS.: Cross-Platform - Application Fundamentals.

So muss man am Anfang der Entwicklung auswählen, welche Android-API als Minimalvoraussetzung verwendet werden und welche Version vorrangig unterstützt werden soll. Zweifellos ist es möglich die von dem Endgerät verwendete Android-Version abzufragen und dementsprechend die Funktionalität der App anzupassen. Zudem können *Java-Packages* eingebunden und verwendet werden, um bekannte Funktionen auch in C# verwenden zu können.

Eine weitere Entwickler-Unterstützung ist das automatische Führen des Android-Manifestes. Dabei werden zwar nur rudimentäre Einstellungen aus der Entwicklung übernommen, aber auch diese Unterstützung ist für Neulinge auf dem Gebiet der Android-Entwicklung eine Beihilfe.

### 7.3.3. Android Emulator

Zum Testen der Android-App konnte ein Emulator verwendet werden, der in dem Xamarin-Plugin für *Visual Studio* bereitgestellt wurde. Damit konnten dann verschiedene Szenarien, wie Verbindungsverlust oder Speicherknappheit, nachgestellt werden. Für die App wurde das Android-API-Level 19 als Minimal-Anforderung gewählt, da eine zeitnahe Android-Version verwendet werden sollte. Das *Target*-Level wurde auf das API-Level 21 gesetzt. Dies steht für Android 5.0 mit dem Namen Lollipop.<sup>26</sup>

## 7.4. Entwicklung der App mit Xamarin Platform

Im Folgenden wird auf die Umsetzung der nativen Android-Applikation, mit Hilfe von Xamarin Platform, eingegangen. Anfangs wurde beim Anlegen des Projektes das Android-API-Level, wie oben beschrieben, auf 19-21 gesetzt und die Entwicklung darauf abgestimmt.

### 7.4.1. Anlegen der Layouts

Es müssen verschiedene *Views* angelegt werden. Darunter sind Login-Fenster, Übersichtsseiten, die die Trainingspläne und Übungen anzeigen, und eine Seite zum Eintragen von Trainingsdaten.

---

<sup>26</sup>DERS.: Understanding Android API Levels.

Beim Aufbau der Oberflächen sind die Schritte eindeutig identifizierbar, da diese Prozedur für alle *Layouts* gleich abgehandelt werden sollte.

Zuerst wird ein *Layout* angelegt und die Bedienelemente (*Widgets*) darauf angeordnet. Dazu kann man *ViewGroups* zur Gruppierung verwenden und die Ausrichtung auf dem Fenster wird mit Hilfe von *Layouts* möglich.<sup>27</sup>

Probleme entstanden dabei jedoch nur manchmal bei der Ausrichtung von *Widget*-Elementen, die aber zügig gelöst werden konnten.

Des Weiteren muss für die Oberfläche dann eine dazugehörige *Activity* angelegt werden, die das Fenster aufruft und die Interaktionen hinter den *Buttons* ausführt.

Zum Anzeigen der Login- und Registrieren-Oberflächen wurden Dialoge verwendet, um dem Benutzer zu zeigen, dass sich die Interaktion zum Anmelden vor dem Zugang zu der eigentlichen Applikation vollzogen werden muss.

### Aufbau der Registrierungsseite

Auf der Seite zur Registrierung sollen Daten über den Fitnesszustand, das Geschlecht und die körperliche Betätigung aus einer Auswahl wählbar sein. Dazu sollten dementsprechend *Spinner*, scrollbare Auswahlfelder, verwendet werden. Dabei lag die Schwierigkeit jedoch in der Belegung mit den auswählbaren Werten. Diese mussten zuerst eingesetzt und dann nach der Übergabe an die *Activity* wieder zurückformatiert werden. Es werden von der Oberfläche nämlich nur *Strings* übergeben. Zur Aufnahme der Daten Geschlecht, Job und Beruf gibt es vorher definierte Daten für die Eingabe auf dem Server. Dabei hat sich anfangs die Schwierigkeit ergeben, dass die *Spinner*, die scrollbaren Auswahlfelder, mit den festgelegten Daten besetzt und dann in dem entsprechenden Datentypen wieder ausgelesen werden müssen. Die Belegung der *Spinner* erfolgt über einen typisierten *ArrayAdapter*, der die Werte des übergebenen *Enums* ausgibt. Das Einlesen des ausgewählten Wertes wird über jeweils eine ausgelagerte Funktion geregelt, die die Auswahl, die als *String* erhalten wird, in den jeweiligen Typ überführt. Anhand der erhaltenen *ServerException* wird dann ausgegeben, welche Eingabe für die Registrierung falsch eingegeben wurde. Dies ist auch einer der Gründe, weshalb die Registrierung nur im Online-Modus der App unterstützt wird. Zudem muss der Server überprüfen, ob der Benutzername verwendet werden kann.

---

<sup>27</sup>BECKER/PANT: Android 2 - Grundlagen und Programmierung, Vgl. S. 58ff.

Nach erfolgreicher Registrierung wird der Dialog wieder ausgeblendet und die Konnektivitätsanzeige der Startseite wird mit dem aktuellen Status belegt. Dies geschieht nicht beim Statuswechsel, da dies mit der Architektur der Online-Status-Abfrage in Verbindung steht. Diese wird in einem eigenen *Thread* durchgeführt und dieser kann keine Änderungen an der Oberfläche vornehmen. Eine Verbesserungsmöglichkeit wäre deshalb ein Aufruf in dem Abfrage-*Thread*, der dann auf dem *UI-Thread* ausgeführt wird. Dann würde die Anzeige immer direkt beim Statuswechsel aktualisiert werden.

### Aufbau der Loginseite

Beim Login wird die Kombination Benutzername und Passwort beim *OnOffServiceLocal* abgefragt, wenn der *Login-Button* betätigt wird. Bei einer ungültigen Eingabe wird ein Hinweis an dem Feld des Passwortes angezeigt, der angibt, dass die Login-Daten falsch sind und eine erneute Eingabe erforderlich ist. Aus Sicherheitsgründen wird nicht angegeben, ob der Benutzername schon nicht in der Datenbank vorhanden ist, oder, ob nur die Kombination fehlerhaft ist. Nach einer validen Eingabe wird der Login-Dialog ausgeblendet und es folgt eine Weiterleitung zu den Trainingsplänen des nun eingeloggten Benutzers.

### Aufbau von Seiten mit Listen

Die Darstellung der Trainingspläne und der zugehörigen Übungen auf einer folgenden Seite sind technisch gesehen gleich. Einzig die Beziehung der zu ladenden Daten und die übergebenen Werte ändern sich. Es soll eine Auflistung der Daten geschehen, die es durch die Auswahl eines Listenfeldes ermöglicht, zur nächsten Seite weiterzuleiten. Als herausfordernd hat sich dabei das Belegen der *ListView* zur Anzeige der tabellarischen Daten und die Weitergabe der *ScheduleId* und der *UserId* herausgestellt. Die *ScheduleId* und die *UserId* werden benötigt, um die Übungen zu einem Trainingsplan ermitteln zu können. Zuvor wurde die *ListView* erstellt. Dafür sind ein *ScheduleListViewAdapter* und ein *ScheduleView* von Nöten. Der *ListViewAdapter* erweitert die Klasse *BaseAdapter* und gibt bei einem Klick die Position des Elements im

Array der Trainingspläne zurück. Das unsichtbare Feld *txtScheduleViewID* im *ScheduleView* ist notwendig, um dieses bei einem Klick auslesen zu können und dann an die folgende Übersichtsseite der Übungen übergeben zu können.

### Parameterweitergabe an die Activity mittels Intent

Zu Anfang wurde vergeblich versucht, die *UserId* und die *ScheduleId* über den Aufruf der *ExerciseActivity* mit zu übergeben. Diese Möglichkeit hat sich im Nachhinein als Fehler herausgestellt und eine weitere Einarbeitung in die vorherig erläuterten *Intents* (siehe Kapitel 7.2.2) durchgeführt. Danach wurde die Umsetzung dahingehend fortgeführt, dass die Funktion *PutExtra()* des *Intents* dazu genutzt wurde, Daten zu übertragen. In der *ExerciseActivity* werden diese dann wieder ausgelesen und weiterverwendet. Beide Werte sind essentiell, um die Übungen des angemeldeten Benutzers zu seinem ausgewählten Trainingsplan zu erhalten (siehe Code-Beispiel 7.1).

```
1  /// <summary>
2  /// Clichevent auf ein Element des ListViews
3  /// Geht zu dem ausgewaehlten Training
4  /// </summary>
5  /// <param name="sender"></param>
6  /// <param name="e"></param>
7  private void lv_ItemClick(object sender, AdapterView.ItemClickEventArgs e)
8  {
9      string selectedExerciseName = exercises[e.Position].Name.ToString();
10     int exerciseId = Integer.ParseInt(exercises[e.Position].Id.ToString());
11     string selectedExerciseDescription =
12         exercises[e.Position].Description.ToString();
13
14     var practiceActivity = new Intent(this, typeof(PracticeActivity));
15     practiceActivity.PutExtra("Exercise", exerciseId);
16     practiceActivity.PutExtra("Schedule", scheduleId);
17     practiceActivity.PutExtra("User", userId);
18     StartActivity(practiceActivity);
19 }
```



**Quelltext 7.1:** Übertragen von Daten zwischen Activities

Bei der Übergabe der Daten an die *PracticeActivity* wird zudem noch die *ExerciseId* übertragen, um alle nötigen Fremdschlüssel für das Anlegen des Trainings zu besitzen.

Eine Anmerkung zu der Übergabe der *UserId*: Diese muss über die *Activities* übertragen und kann nicht einfach aus der aktuellen *UserSession* des Benutzers gelesen werden, da man davon ausgehen muss, dass sich der Benutzer auch offline einloggen kann. Demnach hat man im Online-Modus zwei Möglichkeiten die *UserId* zu erhalten, im Offline-Modus hingegen ist dies die einzige Lösung.

```
1 private async void bt_ItemClick(object sender, EventArgs e)
2 {
3     try
4     {
5         scheduleId = Intent.GetIntExtra("Schedule", 0);
6         exerciseId = Intent.GetIntExtra("Exercise", 0);
7         userId = Intent.GetStringExtra("User");
8         double weight = Double.Parse(txtWeight.Text);
9         int repetitions = Java.Lang.Integer.ParseInt(txtRepetitions.Text);
10        int numberOfRepetitions =
11            Java.Lang.Integer.ParseInt(txtNumberOfRepetitions.Text);
12
13        bool result = await ooService.createPracticeAsync(scheduleId,
14            exerciseId, userId, DateTime.Now, weight, repetitions,
15            numberOfRepetitions);
16        if(result)
17        {
18            //Zurueck zu der Uebungsseite
19            OnBackPressed();
20        }
21    }
22    else
23    {
24        new AlertDialog.Builder(this)
```

```
21         .SetMessage("Anlegen ist fehlgeschlagen")
22         .SetTitle("Error")
23         .Show();
24     }
25 }
26 catch (ServerException ex){[...] }
27 catch (FormatException exc){[...] }
28 catch (Exception exce){[...] }
29 }
```

### Quelltext 7.2: Auslesen von Daten zwischen Activities

Wie im Codebeispiel 7.2 ersichtlich, kann man die übergebenen Informationen zwischen *Activities* aus dem *Intent* auslesen. Zudem kann man erkennen, dass man dank Xamarin das *Parsen* einer *Integer*-Zahl über eine *Java*-Funktion durchführen kann (zu sehen im Codebeispiel 7.2 Zeile 9).

### 7.4.2. Umgang mit Online-/Offline-Situation

Die Prüfung der Kollektivität und daraus resultierende Ablauflogik wird durch den *OnOffService* durchgeführt. Dieser ist die Schicht zum Verteilen der An- und Abfragen, abhängig von dem Verbindungsstatus. Demnach werden immer Methoden dieser Klasse von den *Activities* aufgerufen, wenn Daten abgerufen oder abgelegt werden sollen. Dann wird in der Methode eine Unterscheidung gemacht, ob das Endgerät gerade online oder offline ist und dementsprechend die Interaktion mit der lokalen Datenbank (siehe Kapitel 7.4.3) oder dem Server durchgeführt. Zudem wurde dabei immer die Konvertierung verschiedener Typen durchgeführt, die durch die Architektur nötig wurden.

```
1 public async Task<Guid> SignIn(string username, string password)
2 {
3     User user = new FITNat.DBModels.User();
4     Guid userId;
5     user.Username = username;
6     user.Password = password;
```

```
7     if (Online)
8     {
9         try
10        {
11            bool success = await mgnService.SignIn(username, password);
12            if (success)
13            {
14                userId = mgnService.actualSession().CurrentUserId;
15                user.UserId = userId.ToString();
16                if (db != null)
17                {
18                    db.insertUpdateUser(user);
19                }
20                return userId;
21            }
22        }
23        catch(ServerException ex){[...]throw;}
24        catch(Exception exc){[...] }
25        return new Guid();
26    }
27    else
28    {
29        try
30        {
31            //Lokal nachschauen
32            Guid result = db.findUser(username, password);
33            if (result != null)
34                return result;
35        }
36        catch(Exception exc){[...]throw;}
37        return new Guid();
38    }
39 }
```

**Quelltext 7.3:** Login über den *OnOffService*

Im Code-Beispiel 7.3 kann man die Umsetzung dieser Aufgabe anhand des Logins erkennen. Zum Aufbau der lokalen Datenhaltung wird in diesem Schritt schon der

*User*, der sich gerade einloggt, mit der *UserId* gespeichert. Die *Exceptions* werden bewusst nicht in diesem Schritt behandelt, um der aufrufenden *Activity* die originalen Fehlermeldung des Servers übergeben und den Fehler dann in der Oberfläche darstellen zu können.

Alle Methoden der Klasse *OnOffService* sind asynchron. Das liegt zum einen an den asynchronen Aufrufen, die an den Server gestellt werden. Diese ermöglichen es, dass Aufgaben gestartet werden können und zu einem späteren Zeitpunkt auf die Ergebnisse zugreifen zu können. Dadurch werden die asynchronen Aufgaben ausgelagert und entlasten den *Main-Thread*. Deshalb würde es die Performancevorteile verspielen, wenn man diese asynchronen Methoden dann beim Serverabruf synchron verwenden würde. Auch wenn dabei noch Verbesserungen in der App vorgenommen werden können, um die Ressourcen des Gerätes optimal auszunutzen. Unter Verwendung eines eigenen *Threads* zum Abarbeiten der Server-Anfragen könnten weitere Leistungssteigerungen erreicht werden. Dabei wurde dann aber der Aufwand und die Probleme der *Thread*-Synchronisierung als ein für diese Arbeit zu großer Aufwand geschätzt. Besonders, da die Server-Methoden größtenteils Rückgabewerte liefern, die für die Weiterverarbeitung essentiell sind. Möglich wäre eine Optimierung mit einem startenden *Thread* in den Server-Aufrufen, die dann neben dem *UI-Thread* laufen und bei Fertigstellung die benötigten Daten wieder in den startenden *Thread* übertragen. Damit würde man eventuelle Ladezeiten der Oberfläche minimieren oder sogar vollständig verhindern.

### 7.4.3. Lokale Datenbank

Technologisch wird eine SQLite-Datenbank aus den bereits in Kapitel 7.2.5 erläuterten Vorteilen genutzt.

Die lokale Datenbank dieser App wird für die Umsetzung des *Caches* (siehe Kapitel 7.4.6) benötigt. Darin werden die lokalen Daten gespeichert und mit dem Server abgeglichen. Die Erstellung der Datenbank findet beim Start des *OnOffServices* statt. Zur Verbesserung der Leistung wird die Erstellung in einem separaten Thread durchgeführt, da kein Rückgabewert erwartet wird und der *Main-Thread* so entlastet wird. Die zur Erstellung der Server-Datenbank verwendeten Models konnten in diesem Zusammenhang nicht verwendet werden, da die Annotation der OR-Mapper nicht äquivalent sind. Des Weiteren ist es nicht möglich Fremdschlüssel in SQLite zu deklarieren. Diese wurden nun programmatisch oder über Beziehungstabellen gepflegt.

Daraus ergibt sich eine Verbesserungsmöglichkeit für eine neue Version der Applikation. Als hilfreich könnte sich dabei eine *SQLite-Extension* herausstellen, die der Datenbank dann einen größeren Funktionsumfang bereiten und die Anzahl der direkten SQL-Befehle minimieren würde. Diese wurde testweise eingepflegt, funktionierte aber nicht erwartungsgemäß.<sup>28</sup>

```

1  [Table("User")]
2  public class User
3  {
4      [PrimaryKey, AutoIncrement]
5      public int LocalId { get; set; }
6      public bool wasOffline { get; set; }
7      public string UserId { get; set; }
8      public string Username { get; set; }
9      public string Password { get; set; }
10     public override string ToString()
11     {
12         return string.Format("[User: LocalId={0}, UserId={1}, Username={2},
13                               Password={3}]", LocalId, UserId, Username, Password);
14     }
15 }

```

**Quelltext 7.4:** *UserModel* für die lokale Datenbank

Der lokale *User* (siehe Code-Beispiel 7.4) besitzt eine lokale Id als *PrimaryKey* zur Identifizierung. Geplant war im Vorfeld jedoch eine Kombination aus *wasOffline LocalId*. Da SQLite jedoch keinen *PrimaryKey* aus zwei Attributen unterstützt, musste dieser Plan verworfen werden. Die gespeicherte *UserId* ist der Guid vom Server, der als *Session*-Ersatz gehalten wird. Die anderen benötigten Tabellen werden nach diesem Muster auch erstellt.

Die Interaktion mit der lokalen Datenbank wird synchron durchgeführt, da die asynchrone Schnittstelle nicht alle benötigten Methoden zur Verfügung stellt. Beim Zugriff zur Datenbank wird auf eine Mischung aus direkten SQL-Befehlen und der Nutzung von SQLite-Methoden zurückgegriffen.<sup>29</sup> Einfache Such- oder Einfüge-Operationen

<sup>28</sup>TWINCODERS: SQLite-Net Extensions.

<sup>29</sup>XAMARIN INC.: Using SQLite.NET ORM, Vgl.

werden vom *Framework* bereitgestellt, wohingegen Abfragen über die erstellten Beziehungstabellen selbst umgesetzt wurden.

### 7.4.4. Verbindungsaufbau über den ManagementService

Die Verbindung zum Server geschieht über das bereitgestellte *Package fIT.WebApi.Client.Portable*. Alle benötigten Funktionen werden darin bereitgestellt. Damit wird dann die Kommunikation über REST mit dem Server abgewickelt.

In der App wird der *ManagementService* zum Verbindungsaufbau verwendet. Da ein internes Routing über den *OnOffService* durchgeführt wird, die Verbindung zur lokalen Datenbank von der *LocalDB* gemacht wird, gibt es zur Verbindung mit dem Server den *ManagementLocalService*. Dieser *Service* arbeitet mit dem Server direkt zusammen und ist ausschließlich für das Abrufen von Daten vom Server verantwortlich. Rückgabewerte und Fehler werden meist einfach weitergereicht.

Zur Veranschaulichung zeigt das Code-Beispiel 7.5 einen Ausschnitt aus dem Online-Login, der veranschaulicht, wie die Kommunikation aufgebaut wird.

```
1 public static ManagementService service { get; private set; }
2
3 public async Task<bool> SignIn(string username, string password)
4 {
5     bool result = false;
6     this.username = username;
7     this.password = password;
8     try
9     {
10         session = await getSession(username, password);
11         result = true;
12     }
13     catch (ServerException e){[...]throw;}
14     catch (Exception exc){[...]}
```

**Quelltext 7.5:** Login am Server

### 7.4.5. Verbindungsprüfung zum Server

Die Verbindungsprüfung zum Server geschieht im *OnOffService*. Dafür wird eine Endlosschleife in einem eigenen *Thread* gestartet, um in einem festen Intervall (alle 10 Sekunden) einen Ping-Befehl zum Server zu schicken und damit die Erreichbarkeit des Servers zu überprüfen (siehe Quellcode-Beispiel 7.6). Diese Methode wird vom *ManagementService* des eingebundenen *Packages* bereitgestellt. Das Zeitintervall könnte durch Tests noch feiner eingestellt werden, um mit einem aktuelleren Status intern arbeiten zu können.

```
1 Task.Run(async () =>
2 {
3     while (true)
4     {
5         bool status = false;
6         try
7         {
8             status = await mgnServiceServer.PingAsync();
9         }
10        catch(Exception ex)
11        {
12            status = false;
13        }
14        finally
15        {
16            if (status)
17            {
18                //Online = true;
19                setzeStatus(true);
20                //vorher Offline => jetzt die Aktionen ausfuehren, die nur
                //    lokal gemacht werden konnten
21                if (WasOffline)
22                {
23                    await checkSync();
24                    setzeWasOffline(false);
25                }
26            }
27        }
28    }
29 }
```

```
26         }
27         else
28         {
29             setzeStatus(false);
30             setzeWasOffline(true);
31         }
32         //Timeout 10sek.
33         System.Threading.Thread.Sleep(10000);
34     }
35 }
36 });
```

### Quelltext 7.6: Verbindungsüberprüfung

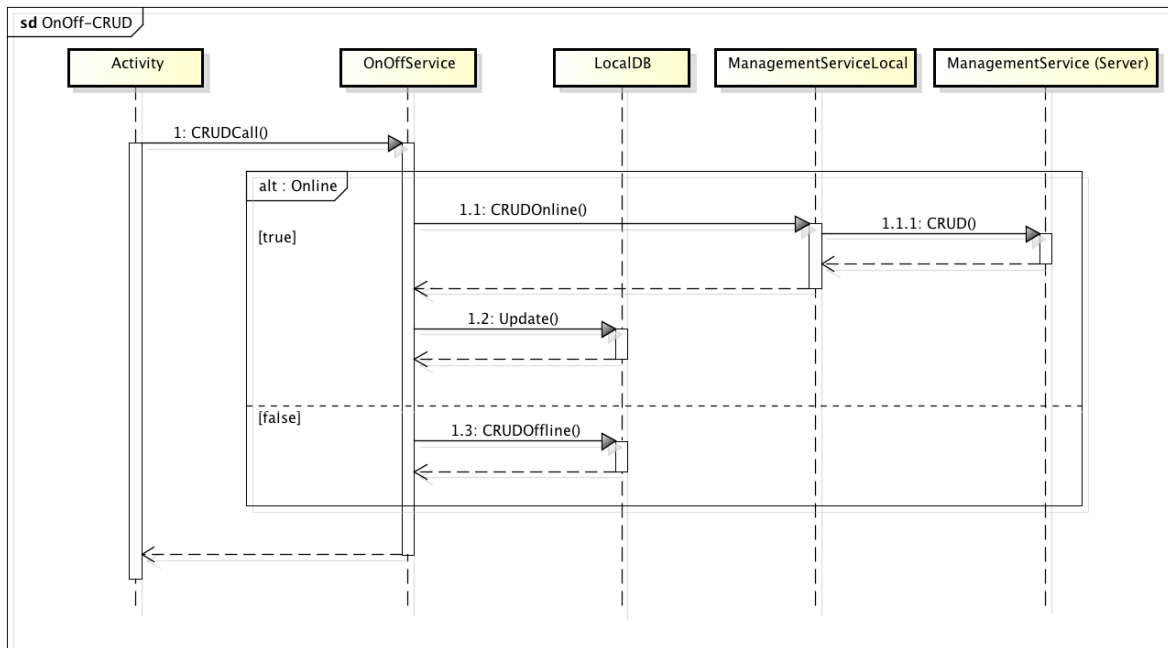
Als elegante Möglichkeit zur Überprüfung, ob eine Verbindung zum Internet besteht, hätte auch eine interne Android-Funktion zur Überprüfung der *InternetConnectivity* ausgereicht. Da aber auch davon ausgegangen werden muss, dass der Server nicht erreichbar ist, die Internetverbindung jedoch noch, könnte besonders dieses Szenario dann eine Reihe von Fehlern verursachen. Deshalb wird die einfach Überprüfung mit Hilfe eines regelmäßigen Ping-Befehls präferiert.

Eine Verbesserung dieses Algorithmus liegt in dem Blockieren des Pings, um den sehr unwahrscheinlichen Fall eines Dirty Reads auf die *Online*-Variable zu vermeiden. Da dieser Fehlerfall als irrelevant eingestuft wurde, wurde diese Umsetzung niedriger priorisiert.

### 7.4.6. Umsetzung des Caches

Der *Cache* (siehe Kapitel 3.1 zur Definition) ist im Fall dieser nativen App ein Zusammenschluss mehrerer im Vorfeld genannter Komponenten und Funktionen. Zum einen wird die Datenhaltung des *Caches* über die lokale Datenbank geregelt, die Verbindungsüberprüfung aus dem vorherigen Artikel wird für die Entscheidung des Verbindungsstatus verwendet. Insgesamt findet sich der *Cache* in der gesamten Logik der App wieder. So werden die Daten in der lokalen Datenbank aktualisiert, falls im Online-Modus Daten abgefragt werden. Diese werden dann mit den lokalen Daten abgeglichen und bei Bedarf aktualisiert (siehe Abbildung 7.4).





powered by Astah

**Abbildung 7.4.:** Sequenzdiagramm CRUD

Die Hauptfunktionalität und -schwierigkeit lag in dem Szenario, dass die App gerade wieder eine Verbindung zum Server aufbaut und im Vorfeld Daten im Offline-Modus gespeichert hat, die dem Server noch nicht bekannt sind. Dieser Fall ist in dem Codeausschnitt 7.6 zu sehen. Dabei wird überprüft, ob die App in dem vorherigen Intervall noch im Offline-Modus war.

```

1 private async Task checkSync()
2 {
3     List<Practice> offPractices = db.GetOfflinePractice();
4     int result;
5     if(offPractices.Count != 0)
6     {
7         foreach (var item in offPractices)
8         {
9             try
10            {
11                User u = db.findUser(item.UserId);
12                result = await mgnService.recordPractice(item.ScheduleId,
                    item.ExerciseId, item.UserId, item.Timestamp, item.Weight,

```

```
        item.Repetitions, item.NumberOfRepetitions, u.Username,
        u.Password);
13         if (result != 0)
14             item.Id = result;
15     }
16     catch (Exception ex){[...] break;}
17 }
18 }
19 }
```

**Quelltext 7.7:** Synchronisation der Offline-Daten

Ist dem so, wird in der lokalen Datenbank nach Trainings gesucht, die offline angelegt wurden (siehe Code-Beispiel 7.7). Diese werden dann mit dem Server abgeglichen und hochgeladen. Das Hochladen geschieht einzeln, damit im Fall eines abrupten Verbindungsverlustes maximal ein Datensatz verloren geht. Dabei wäre es möglich eine Transaktionsverwaltung für die Verbesserung zu integrieren, um dieses Problem zu verhindern. Bei dem Anlegen des Trainings fällt auf, dass die Attribute Benutzername und Passwort noch einmal übergeben werden. Diese Maßnahme musste ergriffen werden, um eine Verbindung zum Server herstellen zu können. Dazu wird eine *Session* benötigt, die vorher noch nicht besteht, für den *Upload* aber essentiell ist. Somit wird die *Session* vor dem Hochladen abgerufen.

## 8. Realisierung des Clients als Web Applikation

In diesem Kapitel wird die Implementierung des Clients als Web-Applikation beschrieben. Hierbei wird genauer auf die verschiedenen Design-Entscheidungen und genutzten Techniken näher eingegangen.

### 8.1. Definition einer Single Page Application

Die Web-Applikation wurde als SPA implementiert. Bei dieser Art der Webanwendung wird die Verarbeitung von Anfragen vom Server auf die Clients verschoben. Die daraus entstehende Anwendung benutzt nur wenige statische Hauptseiten, um den Inhalt darzustellen. Alle dynamischen Inhalte der Seite werden nachträglich per AJAX-Requests hinzugeladen. Hierbei wird die Verarbeitung der nachgeladenen Daten auf dem Client durchgeführt und anschließend in die vorhandene *Hypertext Markup Language (HTML)*-Struktur übernommen, wobei der Grad der Autonomie des Clients vom genutzten Framework abhängt. Diese Auslagerung der Verarbeitung begünstigt die Nutzung des RESTful Web Services (siehe Kapitel 6), welcher die Daten liefert, die die SPA dann verarbeiten kann. Daraus ergibt sich, dass nur eine lose Verbindung zum Web Server besteht, sodass nur die wenigen statischen Seiten und deren eingebundene Skripte abgerufen werden müssen. Die sonstige Kommunikation besteht nur noch zwischen der SPA und dem Web Service.<sup>1</sup>

---

<sup>1</sup> STEYER/SOFTIC: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, S. 31f.

Im Laufe dieses Kapitels wird noch darauf eingegangen, wie auch diese beiden Verbindungen durch geeignete Techniken bis auf ein Mindestmaß reduziert wurden.

## 8.2. AngularJS

Zur Umsetzung der SPA wurde das quelloffene Framework *AngularJS* von *Google* gewählt, welches sich immer größerer Popularität erfreut.<sup>2,3</sup> Es bietet alle Möglichkeiten, um eine Web-Anwendung als weitestgehend autonomen Fat-Client zu implementieren. Dadurch wird die Möglichkeit geschaffen, eine Applikation zu entwickeln, welche selbst dann akzeptabel reagiert, wenn ein aufrufendes Endgerät keine Verbindung zum Internet besitzt. Hierzu stellt es zusätzliches Markup bereit, welches zu Laufzeit interpretiert und ausgeführt wird. Die dazu nötigen Komponenten von AngularJS und deren Einsatz werden nachfolgend genauer erläutert.

### 8.2.1. Abgrenzung des Begriffs: Komponente

Wenn in diesem Kapitel der Begriff Komponente verwendet wird, ist eine abgeschlossene, logische Einheit im AngularJS-Umfeld gemeint. Auf einige dieser Komponenten, wie Services und Controller, wird im späteren Verlauf dieses Kapitels detaillierter eingegangen. AngularJS nutzt den Begriff des *Modules* als einen Container für verschiedene Komponenten.<sup>4</sup>

### 8.2.2. Dependency Injection

AngularJS wurde so konstruiert, dass zu jedem Zeitpunkt eine gute Testbarkeit gewährleistet ist.<sup>5</sup> Aus diesem Grund setzt AngularJS für die Verbindung von verschiedenen Funktionen auf *Dependency Injection*.

Hierbei werden beim Ausführen einer Funktion die genutzten Parametern geprüft.

---

<sup>2</sup>W3-TECH: Usage statistics and market share of AngularJS for websites.

<sup>3</sup>STEYER/SOFTIC: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, S. 33.

<sup>4</sup>ANGULARJS: Online-Dokumentation: Modules.

<sup>5</sup>ANGULARJS: Testability Built-in.

Gibt es eine bekannte Komponente, die den gleichen Namen wie der geforderte Parameter besitzt, erzeugt AngularJS ein Objekt dieser Komponente und übergibt dieses an die Funktion. Dieser Vorgang wird rekursiv durchgeführt, sodass ebenfalls Dependency Injection für die Erzeugung der jeweiligen Parameter-Objekte benutzt wird.<sup>6</sup>

### 8.2.3. Services

*Services* sind abgeschlossene Komponenten, welche bestimmte Funktionalitäten kapseln und bereitstellen. AngularJS bietet neben der Möglichkeit, eigene Services zu erstellen, eine eigene Auswahl, um verschiedene wiederkehrende Aufgaben durchzuführen. Dabei sind die Services von AngularJS immer mit dem Präfix *\$* versehen. Einer der wichtigsten Services für die Umsetzung dieser Arbeit war beispielsweise *\$http*. Dieser stellt Funktionen bereit, welche zur Kommunikation eines Web Services mittels HTTP benötigt werden.

Services können von Controllern (siehe 8.2.5) verwendet werden, um Daten zu erhalten und an die Oberfläche weiterzugeben. Hierbei verwenden Sie Dependency Injection, um auf einen *Service* zuzugreifen, wobei nur beim ersten Zugriff ein neues Objekt erzeugt wird (Singleton-Muster). Fordert eine weitere Komponente ein Objekt des Services an, wird das bereits erstellte Objekt zurückgegeben.<sup>7</sup>

### 8.2.4. Promises

Im letzten Abschnitt wurde der Service *\$http* angesprochen (siehe 8.2.3), welcher Methoden zur Kommunikation mit einem RESTful Web Service bereitstellt. Würde diese Kommunikation synchron ausgeführt werden, würde diese AngularJS bis zum Erhalt der Antwort blockieren, da *Javascript* immer nur in einem Thread ausgeführt wird.<sup>8</sup> Deshalb wurde das Prinzip der *Promises* eingeführt. Dies erlaubt die Abarbeitung von asynchronem Code, indem beim Aufruf einer asynchron abzuarbeitenden Methode ein Promise-Objekt zurückgegeben wird. Dieses repräsentiert ein Versprechen über ein späteres Ergebnis. Auf dieses Ergebnis kann mit den Methoden *then* und *catch* reagiert werden.

---

<sup>6</sup>ANGULARJS: Online-Dokumentation: Dependency Injection.

<sup>7</sup>DERS.: Online-Dokumentation: Services.

<sup>8</sup>KANTOR: JavaScript is single-threaded.

Then wird nach erfolgreicher Abarbeitung der asynchronen Methode ausgeführt. Tritt bei der Verarbeitung ein Fehler auf, wird die *catch*-Methode ausgeführt. Da diese beiden Methoden jeweils selber Promise-Objekte zurückgeben, ist ein Verketteten der Aufrufe möglich. Die Nutzung von Promisses ist im Module *\$q* gekapselt, welches sich an dem Projekt *q*<sup>9</sup> orientiert.<sup>10,11</sup>

### 8.2.5. MVC

AngularJS nutzt das Architektur-Muster *Model-View-Controller (MVC)* um Datenbeschaffung bzw. -haltung, Datenverarbeitung und Datenpräsentation strikt zu trennen.<sup>12</sup>

Zur Beschaffung werden entweder Services, welche per Dependency Injection hinzugeladen werden, benutzt oder Funktionen zur Erzeugung von Model-Objekte erstellt, welche mit Konstruktoren aus dem objektorientierten Umfeld verglichen werden können.

Die so erhaltenen Datenstrukturen können von Controllern aufgerufen werden. Dabei handelt es sich um Komponenten, welche von AngularJS zur Anreicherung eines bestimmten Markup-Blocks aufgerufen werden. Ihre Aufgabe ist es, die für die Oberfläche benötigten Daten zu besorgen, diese aufzubereiten und sie an die View weiterzugeben. Das Code-Beispiel 8.1 zeigt den Aufbau einer Controller-Komponente, welche Daten für die Navigationsleiste bereitstellt.<sup>13</sup>

```
1 function indexController($scope, $location, $interval, authFactory) {
2   $scope.onlineStatus = 'offline';
3   $scope.logout = function () {
4     authFactory.logout();
5     $location.path('/');
6   }
7   [...]
8   $scope.onlineStatus = true;
9   $scope.navbarExpanded = false;
```

---

<sup>9</sup>KOWAL: Documentation *q*.

<sup>10</sup>ANGULARJS: Online-Dokumentation: Module *\$q*.

<sup>11</sup>STEYER/SOFTIC: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, S. 211ff.

<sup>12</sup>Ebd., S.34.

<sup>13</sup>ANGULARJS: Online-Dokumentation: Controller.

```

10  $scope.authentication = authFactory.authentication;
11  };

```

### Quelltext 8.1: Controller für die Navigationsleiste

Hierbei ist zu sehen, wie mittels Dependency Injection die benötigten Komponenten für die Funktion bereitgestellt werden (siehe Code-Beispiel 8.1 Zeile 1). Interessant ist dabei besonders die Komponente `$scope`. Diese wird verwendet, um Daten zwischen dem Controller und der View (in dem Fall: der statischen HTML-Seite) auszutauschen (siehe Code-Beispiel 8.1 Zeile 8ff.).<sup>14</sup> Dabei stellt AngularJS Funktionen bereit, um diesen Austausch bidirektional durchzuführen. Somit können beispielsweise Daten, welche ein Nutzer in ein Textfeld eingibt, im Controller weiterverarbeitet werden. Die Benutzung der durch den `$scope` bereitgestellten Variablen wird im Code-Beispiel 8.2 gezeigt.

```

1  <nav class="navbar navbar-inverse navbar-fixed-top" role="navigation"
    data-ng-controller="indexController">
2  <div class="container">
3    <div class="navbar-header">
4      [...]
5      <ul class="nav pull-left">
6        <li class="status online navbar-text" data-ng-hide="!onlineStatus">
7          <span class="glyphicon glyphicon-ok "></span> Online
8        </li>
9        <li class="status offline navbar-text" data-ng-hide="onlineStatus">
10         <span class="glyphicon glyphicon-remove"></span> Offline
11       </li>
12     </ul>
13   </div>
14   <div id="navbar" class="navbar-collapse collapse">
15     <ul class="nav navbar-nav navbar-right">
16       <li data-ng-show="!authentication.isAuthenticated">
17         <a href="#/login">Log in</a>
18       </li>
19       <li data-ng-show="!authentication.isAuthenticated">
20         <a href="#/register">Register</a>
21       </li>
22       <li data-ng-show="authentication.isAuthenticated">

```

<sup>14</sup>DERS.: Online-Dokumentation: Scopes.

```
23         <a href="#">Hallo {{authentication.userName}}</a>
24     </li>
25     <li data-ng-show="authentication.isAuthenticated">
26         <a href="#/schedules">Schedules</a>
27     </li>
28     <li data-ng-show="authentication.isAuthenticated">
29         <a href="#" data-ng-click="logout()">Log out</a>
30     </li>
31 </ul>
32 </div>
33 </div>
34 </nav>
```

**Quelltext 8.2:** Navigation der Hauptseite erweitert um AngularJS-Markup

Über das Attribut *data-ng-controller* (siehe Code-Beispiel 8.2 Zeile 1) wird ausgesagt, dass dieser *div*-Block durch die Controller-Funktion *indexController* bearbeitet wird. Nur in diesem Geltungsbereich kann auf die Eigenschaften des Controllers zugegriffen werden.

Hierbei gibt es verschiedene Möglichkeiten, die Daten des Controllers zu benutzen:

- **Nutzung in Direktiven**

AngularJS stellt Direktiven bereit, welche die Darstellung der Webseite beeinflussen. Dies zeigt sich in Zeile 6. Die Direktive *ng-hide* wird benutzt, um dynamisch HTML-Elemente auszublenden. Für die Entscheidung, ob das zugehörige *li*-Element ausgeblendet werden soll, wird die Variable *onlineStatus* aus dem *\$scope* des Controllers abgefragt. Ändert sich die Variable im Controller, wird die Direktive neu ausgewertet.<sup>15</sup>

- **Ausgabe des Wertes**

Der Wert einer Variable kann direkt ausgegeben werden. Dies wird in Zeile 23 gezeigt. Damit AngularJS erkennt, dass eine *\$scope*-Variable ausgegeben werden soll, muss diese von zwei geschweiften Klammern umgeben sein.

- **Zugriff auf Methoden**

In Zeile 29 wird eine Direktive benutzt, um aus der View heraus eine im *\$scope* definierte Funktion aufzurufen.

---

<sup>15</sup>ANGULARJS: Online-Dokumentation: Directiven.



### 8.2.6. Routing

Damit das Markup nicht schnell durch die Nutzung von zusätzlichen Direktiven überladen wird, bietet AngularJS das Module *\$route* zur Implementierung von Routing an. Hierbei wird durch die Direktive *ng-view* ein Block-Element (z.B. ein *div*-Element) als View-Container definiert. Dieser wird abhängig von der aufgerufenen URL mit unterschiedlichen Inhalten befüllt. Das Code-Beispiel 8.3 zeigt solch eine Routing-Konfiguration.

```
1  fIT.config(["$routeProvider", function ($routeProvider) {
2    $routeProvider.when("/", {
3      controller: "scheduleController",
4      templateUrl: "app/views/schedules.html"
5    }).when("/schedule/:id", {
6      controller: "scheduleController",
7      templateUrl: "app/views/schedule.html"
8    }).when("/login", {
9      controller: "loginController",
10     templateUrl: "app/views/login.html"
11   }).when("/register", {
12     controller: "signupController",
13     templateUrl: "app/views/signup.html"
14   }).otherwise({
15     redirectTo: "/"
16   });
17 });
```

**Quelltext 8.3:** Routing mit AngularJS

Stimmt eine Route überein, wird der konfigurierte Controller aufgerufen. Dessen Scope wird nach der Abarbeitung an die definierte View weitergegeben. Die Views sind hierbei HTML-Dateien mit Markup-Schnipsel, die innerhalb des View-Containers erzeugt werden. Diese Markup-Schnipsel liegen ebenfalls auf dem Server.

## 8.3. Umsetzung

Durch die Nutzung der vorgestellten Komponenten und eines Tutorials<sup>16</sup> war es möglich, einen Prototyp umzusetzen, welcher die grundlegenden Anforderungen, die in Kapitel 4 definiert wurden, erfüllt. Nachfolgend werden einige Teilaspekte im Zusammenhang mit der Implementierung näher beleuchtet.

### 8.3.1. Layout mit Twitter Bootstrap

Zur Erstellung einer Oberfläche wurde vollständig auf das quelloffene Cascading Style Sheet (CSS)-Framework *Bootstrap* von *Twitter* gesetzt. Dies ist unter dem Aspekt *designet, einmal definiertes CSS auf allen Endgeräten eine natürliche und gut nutzbare Anwendung entsteht*.<sup>17</sup> Dies liegt daran, dass Bootstrap es erlaubt, unter Nutzung eines integrierten Grid-Systems, eine -responsive Applikation zu erstellen.

Durch den großen Umfang des Frameworks und da im ersten Schritt nur ein Prototyp erzeugt werden sollte, war es möglich, alle Anforderungen in die Web-Applikation zu integrieren, ohne, dass weitere Implementierung von CSS nötig war.

### 8.3.2. Herausforderungen durch das statusloses Protokoll HTTP

Da HTTP ein statusloses Protokoll ist, ist es nicht ohne Weiteres möglich, ein einmal abgerufenen *Access-Token* wiederzuverwenden. Um dies dennoch zu erreichen, wurde Service *authFactory* entwickelt, welcher das ein- und ausloggen verwaltet. Hierbei wird unter Zuhilfenahme der *LocalStorage*-API ein erhaltenes *Access-Token* mit dem Nutzernamen und dem Ablaufdatum persistiert. Ist dieser Datensatz vorhanden und das Ablaufdatum noch nicht erreicht, gilt der Nutzer als angemeldet und kann auf seine Trainingspläne zugreifen.

Gleichzeitig wird für die Verwaltung von Inhalt, welchen nur authentifizierte Nutzer abrufen können, eine weitere Komponente namens *Interceptor* benutzt. Dies ist ein

---

<sup>16</sup>JOUDEH: AngularJS Token Authentication using ASP.NET Web API 2, Owin, and Identity.

<sup>17</sup>TWITTER: Bootstrap Online Documentation.

spezieller Service, der eng mit dem *\$http*-Service verbunden ist.<sup>18</sup> Mit dem Interceptor ist es möglich, eine Request kurz vor- und eine Response direkt nach Erhalt einzusehen und gegebenenfalls darauf zu reagieren. Diese Technik wird verwendet, um vor dem Absenden eines Requests den *authorization*-Header zu setzen, falls ein Access-Token vorhanden ist.

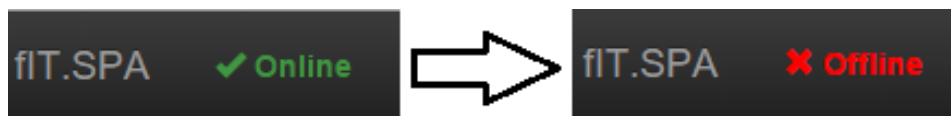
Die Response ist wegen des eingehenden Statuscodes interessant: Wenn der Web Service mit *401 (Unauthorised)* antwortet, ist der Nutzer nicht angemeldet. Daraufhin wird ein möglicher Datensatz mit einem Access-Token gelöscht und der Nutzer wird auf die Login-Seite umgeleitet.<sup>19</sup>

### 8.3.3. Online-Check

Der Nutzer soll eine visuelle Rückmeldung darüber bekommen, ob die Applikation momentan eine Verbindung zum Web Service aufbauen kann oder nicht. Dafür wurde der im Quellcode-Beispiel 8.1 gezeigte Controller um einen Online-Check erweitert. Darin ruft die Anwendungen in einem 5 Sekunden-Intervall die folgende URL auf:

*http://fit-bachelor.azurewebsites.net/api/accounts/ping.*

Schlägt diese Anfrage mit den Statuscode *0* fehlt, liegt keine Verbindung vor und der aktuelle Status ändert sich von Online zu Offline. Die Änderung der grafischen Oberfläche zeigt Abbildung 8.1.



**Abbildung 8.1.:** Veränderung der Statusanzeige

## 8.4. Erweiterung um Offline-Nutzung

Die bisher vorgestellten Komponenten und Umsetzungen führten zum Prototyp einer funktionierenden Single Page Applikation. Diese benötigt jedoch zur Nutzung noch eine Verbindung zum Web Server, auf der die Applikation untergebracht wird und eine

<sup>18</sup>ANGULARJS: Online-Dokumentation: Interceptor.

<sup>19</sup>JOUDEH: AngularJS Token Authentication using ASP.NET Web API 2, Owin, and Identity.

Verbindung zum Web Service, zur Durchführung von Interaktionen. In den folgenden Abschnitten wird beschrieben, wie Techniken eingesetzt wurden, um den Zugriff auf diese Ressourcen auf ein Minimum zu reduzieren.

### 8.4.1. Implementierung des `CachedHttpServices`

Zur Reduzierung der Bindung an den Web Service wurde ein Cache implementiert. Hierbei wurde von der Planung des Caches aus Kapitel 3.2 abgewichen.

Eigentlich sollte für jede Entität ein lokales Pendant erstellt werden, welche die Daten speichert, die bei einem Verbindungsabbruch nicht an den Server gesendet werden können. Durch die besonderen Eigenschaften von JavaScript und der genutzten Datenbank war eine Vereinfachung dieser Planung möglich:

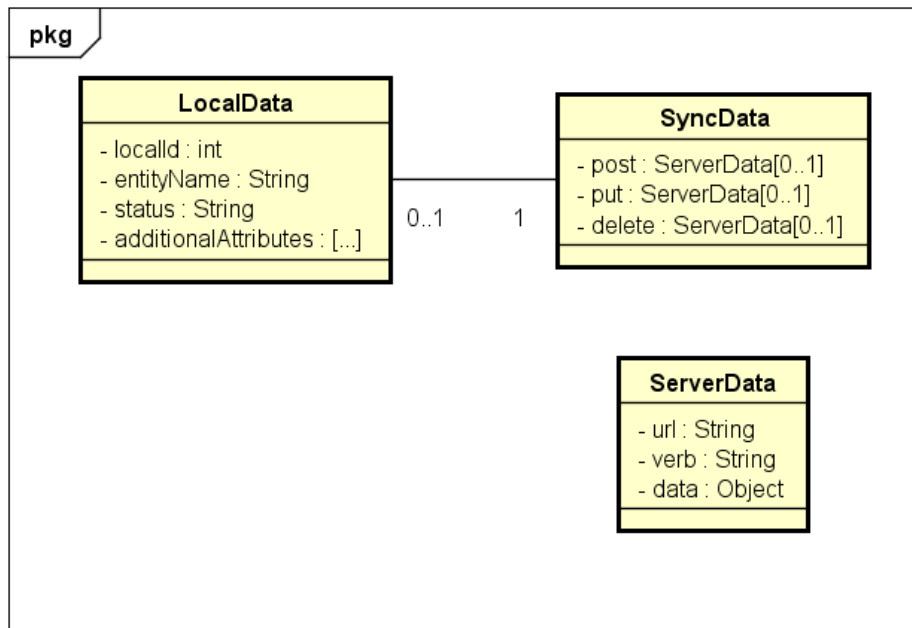
- Javascript erlaubt es, Objekte zur Laufzeit beliebig zu verändern und zu erweitern. Darum kam die Idee auf, die Serverdaten, welche noch nicht an den Server gesendet wurden um Metadaten für die lokale Speicherung zu erweitern und anschließend lokal zu persistieren.
- Diese Möglichkeit wird durch die Datenbank unterstützt. Es handelt sich dabei um die *Indexed Database API*. Diese erlaubt es, innerhalb des Browsers, eine *NoSQL*-Datenbank anzulegen und zu verwalten. Sie wird von den meisten Browsern unterstützt.<sup>20</sup> Da NoSQL-Datenbanken keine festen Schemata kennen, sondern beliebige Datenstrukturen per Index oder Schlüssel bestimmen, ist die Ablage eines dynamisch erstellten Objekts ohne weiteren Aufwand möglich. Zur Nutzung der IndexedDB wurde ein externes AngularJS-Module verwendet.<sup>21</sup>

Durch diese Änderungen kann die Erstellung des lokalen Caches erheblich vereinfacht werden, indem nicht mehr für jede Entität ein lokales Abbild vorgehalten wird. Es gibt eine zentrale Stelle für DB-Entitäten. Der Aufbau wird in Abbildung 8.2 aufgezeigt.

---

<sup>20</sup>CANIUSE.COM: Can I Use: IndexedDB.

<sup>21</sup>BRAMSKI: Angular-IndexedDB.



powered by Astah

**Abbildung 8.2.:** Datenmodell zur Speicherung lokaler Daten

### Umsetzung des Caches über HTTP-Verbs

Durch diese Zentralisierung der Cache-Daten konnte ein Service entwickelt werden, der den *\$http*-Service kapselt und bei jedem Senden einer GET-, POST-, PUT- und DELETE-Anfrage die lokalen Daten mit denen des Web Services synchronisiert. Dafür wird nach dem Senden der Nachricht geprüft, ob der Web Service erfolgreich erreicht wurde.

Ist dies der Fall, werden die neuen Daten mit dem Status *server* lokal aktualisiert. Dies sagt aus, dass die Daten mit denen des Servers übereinstimmen und keine Synchronisation erfolgen muss.

Wenn der Web Service nicht erreicht wurde, werden die Daten mit dem Status *local* zwischengespeichert. Dazu wird ein *SyncData*-Objekt unterhalb des lokalen Datensatzes angelegt. Dieses definiert Eigenschaften mit Daten für spätere Synchronisationsprozesse.

### Synchronisation zwischen Server und SPA

Ist der Web Service wieder erreichbar, werden alle Datensätze mit dem Status *local* synchronisiert, indem die Anfragen nacheinander an den Server gesendet werden. Hierbei wird per Eigenschaft *verb* entschieden, welche Methode des *\$http*-Services genutzt werden soll. Dieser werden dann die gespeicherte URL sowie eventuelle Daten zum Senden an den Server übergeben.

Die Reihenfolge der verarbeiteten HTTP-Verben für die Synchronisation wurde wie folgt gewählt:

- *DELETE*

Besitzt ein Objekt eine *DELETE*-Eigenschaft, werden dessen Daten als erstes verarbeitet. Alle weiteren offenen Synchronisationsprozesse werden im Zuge der Löschung dieses Datensatzes ebenfalls mit gelöscht. Dadurch wird die Anzahl der benötigten Requests verringert.

Besitzt ein Synchronisationsobjekt sowohl eine *POST*- und *DELETE*-Eigenschaft, ohne, dass zwischenzeitlich eine Synchronisation durchgeführt wurde, ist das Objekt lokal erzeugt und anschließend wieder lokal gelöscht worden. Deshalb wird der Datensatz direkt aus dem lokalen Speicher entfernt, ohne, dass Anfragen an den Server gesendet werden müssen. Dies dient ebenfalls der Verringerung der zu sendenden Requests.

- *POST*

Sind keine *DELETE*-Requests vorhanden, wird nach einem offenen *POST*-Request für das Objekt gesucht und bei Fund durchgeführt. Mit dem Ergebnis wird die ID des lokalen Datensatzes angepasst, damit eventuell anstehende *PUT*-Requests auf das richtige Server-Objekt angewandt werden.

- *PUT*

Zum Schluss wird auf offene *PUT*-Requests geprüft und gegebenenfalls durchgeführt.

Nach jeder erfolgreichen Abarbeitung eines Synchronisationsbefehls wird dieser aus dem Synchronisationsobjekt entfernt. Wenn dieses daraufhin keine offenen Befehle mehr besitzt, wird es gelöscht. Damit der Nutzer nicht auf die Abarbeitung der Synchronisationsprozesse warten muss, wird dieser Vorgang komplett asynchron durchgeführt. Durch diese Änderungen ist eine Verbindung zum Web Service optional.

### 8.4.2. Das AppCache-Manifest

Der Einsatz des *cachedHttpService* erlaubt eine Nutzung der SPA auch ohne eine bestehende Verbindung zum Web Service. Doch noch braucht die SPA eine Verbindung zum Web Server, damit statische Dateien wie View-Templates und Script-Dateien nachgeladen werden können. Diese Verbindungen können durch die Nutzung eines *AppCaches* reduziert werden.

Der AppCache wurde im Zuge von HTML5 implementiert und wird von allen gängigen Browsern in der aktuellen Version unterstützt.<sup>22,23</sup> Hierbei wird eine Manifest-Datei auf dem Server abgelegt, welche Aussagen darüber trifft, welche Dateien lokal auf einem Client gespeichert werden sollen. Im Falle dieser Arbeit wurde konfiguriert, dass alle Dateien lokal gespeichert werden (siehe Code-Beispiel 8.4 ab Zeile 5ff.).

```
1  CACHE MANIFEST
2
3  # Time: Fri, 28 Aug 2015 18:34:32 GMT
4
5  CACHE:
6  /Content/bootstrap.min.css
7  /Content/bootstrap-theme.min.css
8  /Content/Site.css
9  /fonts/glyphicons-halflings-regular.woff
10 /fonts/glyphicons-halflings-regular.ttf
11 /Scripts/modernizr-2.8.3.js
12 /Scripts/jquery-1.9.1.min.js
13 /Scripts/bootstrap.min.js
14 /Scripts/angular.min.js
15 /Scripts/angular-route.min.js
16 /Scripts/angular-local-storage.js
17 /app/directive/sameAs.js
18 /app/controllers/homeController.js
19 /app/controllers/indexController.js
20 /app/controllers/signupController.js
21 /app/controllers/loginController.js
22 /app/controllers/scheduleController.js
23 /app/factory/enumFactory.js
24 /app/factory/authFactory.js
```

<sup>22</sup>W3C: HTML5 Application Cache.

<sup>23</sup>CANIUSE.COM: Can I Use: Offline web applications.

```
25 /app/factory/scheduleFactory.js
26 /app/factory/authInterceptorFactory.js
27 /app/views/schedules.html
28 /app/views/login.html
29 /app/views/schedule.html
30 /app/views/signup.html
31 /app/app.js
32
33 NETWORK:
34 *
35
36 SETTINGS:
37 prefer-online
```

### Quelltext 8.4: Cache-Manifest-Datei

Sind diese Dateien erstmal auf dem Client gespeichert, werden sie genutzt, wenn keine Verbindung zum Server besteht.

Damit die Dateien trotzdem auf dem aktuellen Stand sind, wurde im *SETTINGS*-Bereich definiert, dass die Online-Ressourcen vorrangig genutzt werden sollen (siehe Zeile 37). Dies wird aber trotzdem nicht von allen Browser berücksichtigt. Deswegen gibt es eine Möglichkeit, den Client dazu zu zwingen die Ressourcen erneut von Server abzurufen. Hierzu muss die Manifest-Datei angepasst werden. Wenn der Client das nächste Mal eine Verbindung zum Server aufbaut, wird die neue Manifest-Datei heruntergeladen. Dies führt dazu, dass der Client alle lokalen Dateien invalidiert und sich die Dateien, welche das neue Manifest-Datei definiert, erneut herunterlädt. Damit das Aktualisieren der Datei leichter umgesetzt werden kann, wurde ein Zeitstempel als Kommentar in die Manifest-Datei integriert (siehe Zeile 3). Somit kann man leicht ein Neuladen der Serverdateien herbeiführen.

## 8.5. Fazit

Es konnte ein voll funktionsfähiger Prototyp entwickelt werden, welcher die vorher festgelegten Anforderungen umsetzt. Hierbei wurde der Funktionsumfang mit dem Browser *Chrome* der Version 44.0.2403.157 getestet.

Vor- und Nachteile, welche sich während der Entwicklung der SPA ergeben haben, werden gesondert in nachfolgende Kapitel 9 beschrieben und bewertet.



## **9. Gegenüberstellung der clientseitigen Implementierungen**

Ziel dieses Kapitels ist die Gegenüberstellung der Erkenntnisse zur Entwicklung einer verlässlichen mobilen Applikation. Hierbei wird das neu erlangte Wissen zur Umsetzung einer Applikation als SPA und als native App bewertet, sodass mit der vorteilhafteren der beiden Optionen der Messeprototyp umgesetzt werden kann.

Das Kapitel schließt auch gleichzeitig die Entwicklung des Meilensteins 1 ab.

### **9.1. Umsetzung als Web App**

Als Erstes sollen die Vor- und Nachteile der Umsetzung des Clients als Single Page Application aufgezeigt werden.

#### **9.1.1. Vorteile**

Bei der Umsetzung des Clients als Web Applikation zeigen sich die Vorteile besonders in der Umsetzung der Oberfläche.

Durch die Nutzung aktueller Webtechniken und geeigneter Frameworks lässt sich sehr leicht ein einheitliches Aussehen schaffen, welches für verschiedene Anzeigegrößen optimiert ist. Hierbei ist man nicht nur auf mobile Endgeräte beschränkt, sondern erhält zusätzlich eine Webseite, die eine Desktop-Anwendung ersetzen kann. Auch die Umsetzung der Business-Logik konnte ohne großen Einarbeitungsaufwand bewerkstelligt werden. Dabei fällt auf, dass durch das Voranschreiten von HTML5 viele Funktionen, welche vor einigen Jahren nur durch Desktop Applikationen umgesetzt

werden konnten, heute schon problemlos im Browser abbildbar sind. Hierbei zeigten sich aber auch die Schwächen einer Umsetzung als Web Applikation.

### 9.1.2. Nachteile

Wie bereits erwähnt sind viele, jedoch noch nicht alle, Techniken für den Browser umgesetzt. So ist die Umsetzung der *IndexedDB* für *iOS* und *Microsoft*-Geräte noch sehr fehleranfällig.<sup>1</sup>

In diesem Punkt spiegelt sich auch das größte Problem bei der Umsetzung von Web Anwendungen wieder: Unterschiedliche Browser implementieren einige APIs anders oder teilweise auch gar nicht, sodass viel Entwicklungszeit für das Anpassen der Funktionen und Oberflächen für die verschiedenen Browser genutzt werden muss. Wenn es nun so ist, dass Kernkomponenten, wie in unserem Fall die *IndexedDB*, in einigen wichtigen Browsern, wie beispielsweise dem Safari-Browser unter iOS (Nutzung Version 8 bei 7.33% (Stand 31.08.2015)<sup>2</sup>), nicht ausreichend unterstützt werden, ist die Umsetzung dieses Teilaspektes für den produktiven Einsatz fast unmöglich.

Ein weiterer Nachteil ergibt sich aus der Nutzung von AngularJS. Da die gesamte Datenaufbereitung mit der Authentifizierung und dem Routing auf Seiten des Clients passiert, können die lokal gespeicherten Daten, mit Hilfe der Entwicklungswerkzeuge des Browsers, einfach ausgelesen werden. Deswegen wäre es unter Sicherheitsaspekten fahrlässig, die hier vorgestellte Implementierung der Authentifizierung (siehe Kapitel 8.3.2) ohne weitere Sicherheitsmaßnahmen produktiv einzusetzen.

## 9.2. Umsetzung als native App

Da nun die SPA mit den Vor- und Nachteilen vorgestellt wurde, sollen auch die verschiedenen Aspekte bei der Umsetzung als native Applikation vorgestellt werden.

---

<sup>1</sup>CANIUSE.COM: Can I Use: IndexedDB.

<sup>2</sup>Ebd.

### 9.2.1. Vorteile

Die Vorteile einer nativen Applikation liegen besonders in dem großen Funktionsumfang. Dieser kann alle bereitgestellten Funktionen des Betriebssystems ausnutzen. Dazu zählen das *Threading* (siehe Kapitel 7.2.4) und interne Aufrufe über *Services*, welche zum Beispiel zum Versenden von E-Mails verwendet werden können. Zudem können Daten persistent, auch über die Dauer einer *Session* hinaus, auf dem Gerät gespeichert werden (siehe Kapitel 7.4.3).

Unter Sicherheitsaspekten ist die native App einer Web-App vorzuziehen, da Daten nur mit guten technischen Kenntnissen ausgelesen werden können. Sind die Daten darüber hinaus lokal verschlüsselt, entsteht ein noch höherer Grad an Sicherheit.

Weiterhin ist die gesamte Logik der Applikation für den Endanwender nicht sichtbar. Dadurch ist eine Manipulation der Daten wesentlich schwieriger, als bei einer SPA. Diese sind nur über aufwendige programmatische Eingriffe möglich.

### 9.2.2. Nachteile

Die Umsetzung der nativen App beansprucht viel Zeit und ein grundlegendes Knowhow über die Funktionsweise des zu unterstützenden Betriebssystems.

Darin liegt auch noch ein weiteres Problem: Um eine große Marktabdeckung mit einer nativen App zu erreichen, benötigt man mindestens eine iOS- und eine Android-Applikation. Erst dann hat man Zugang zu über 95% der Smartphone-Nutzer in Deutschland.<sup>3</sup>

Die Entwicklung für zwei Systeme kann daraufhin in zwei Möglichkeiten umgesetzt werden: Zum einen könnten native Apps in den jeweiligen Programmiersprachen entwickelt werden. Zum anderen kann eine Multiplattform-Lösung, wie Xamarin-Platform es ist, eingesetzt werden. Dabei beschränkt sich der Funktionsumfang jedoch hauptsächlich auf die grundlegenden Funktionen. Auf Spezialfunktionen eines bestimmten Systems kann nur zugegriffen werden, wenn man eine Applikation exklusiv für eine spätere Endplattform erstellt. Weiterhin ist das Erstellen von Oberflächen aufwendiger als bei einer Single Page Application.

---

<sup>3</sup>IDC: Prognose zu den Marktanteilen der Betriebssysteme am Absatz vom Smartphones weltweit in den Jahren 2015 und 2019.

Zudem müssen native Apps direkt auf das Endgerät geladen und installiert werden und können nicht einfach über das Internet aufgerufen werden.

### 9.3. Fazit aus Meilenstein 1

Zusammenfassend kann festgehalten werden, dass die Nachteile der Single Page Application dahingehend überwiegen. Besonders die aktuell unzureichend implementierte Datenbank in einigen mobilen Browsern lässt es aktuell nicht zu, dass eine mobile Applikation für mehrere mobile Endgeräte entwickelt werden kann. Darüber hinaus kann die Sicherheit der Daten - besonders in Verbindung mit Vitaldaten - aktuell nicht vollständig gewährleistet werden kann. Diese Anforderung kann aktuell nur von einer nativen App zufriedenstellend geleistet werden. Ändert sich der Zustand der Implementierung, müsste diese Evaluation neu durchgeführt werden.

Somit wird im weiteren Vorgehen der Prototyp der nativen App zu einem Messeprototyp weiterentwickelt, da die Argumente gegen die Weiterentwicklung der Web-App bei einer nativen App nicht gegeben sind. Die Nachteile einer nativen Applikation sind darüber hinaus für die Weiterentwicklung zum Messeprototyp vernachlässigbar.

# 10. Weiterentwicklung eines Clients zu einem Messeprototyp

In diesem Kapitel wird die Weiterentwicklung der nativen App zu einem Messe-Prototyp beschrieben. Nach der Entscheidung für die Weiterentwicklung der nativen Android-App, begann die Planung der möglichen Erweiterungen. Dabei wurden besonders Performance- und Stabilitätsaspekte in den Vordergrund gestellt. Darüber hinaus sollte aber auch die Oberfläche einem Messeprototyp entsprechend verbessert werden und Funktionen, die aus den Kann-Kriterien des Pflichtenheftes entspringen, umgesetzt werden, um einen größeren Funktionsumfang präsentieren zu können.

## 10.1. Anpassungen der Ablauflogik

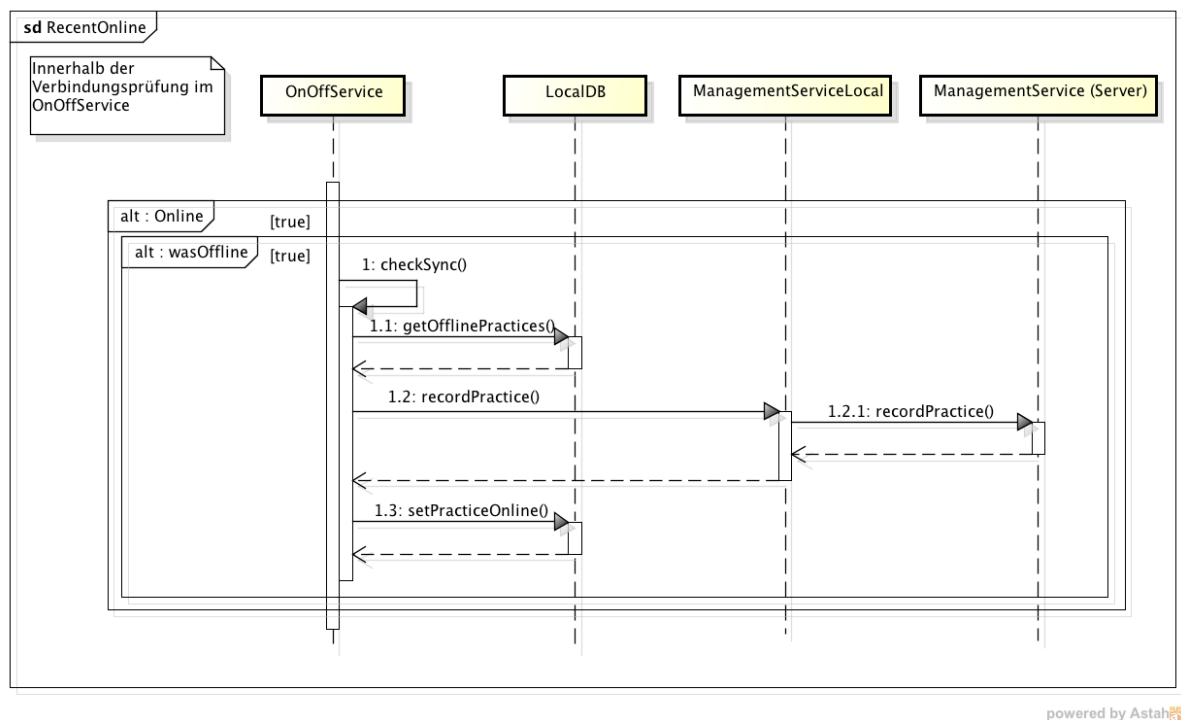
Die Ablauflogik der nativen App wurde weitestgehend beibehalten, da die Planung im Vorfeld schon eine komplette *User-Story* vorgesehen hat. So muss man sich zuerst anmelden, um anschließend durch die Trainingspläne und Übungen navigieren zu können und abschließend die Möglichkeit hat ein Training einzutragen. Demnach sind in diesem Sinne keine Verbesserungen oder Änderungen nötig.

Da die Anzeigelogik keiner großen Anpassung bedarf, wurde sich darauf konzentriert Aspekte der Stabilität und der Leistungsfähigkeit der App zu verbessern. So wurde die Synchronisation aus den Methoden der GET- und POST-Abfragen extrahiert und zentralisiert (siehe Kapitel 3.2.2).

Die Funktionalität des Caches wurde nur hinsichtlich der Synchronisation mit dem Server angepasst. Darüber hinaus wurden keine Änderungen vorgenommen. Das Synchronisieren erfolgt jeweils beim Abrufen von Daten vom -, sowie beim Übertragen

von Daten zum Server.

Die Abbildung 10.1 verdeutlicht den neuen Ablauf der Synchronisation.



**Abbildung 10.1.:** Sequenzdiagramm *RecentOnline*

Bedingung zum Start des Synchronisationsvorgangs ist, dass die Applikation aktuell eine Verbindung zum Server besitzt und im vorhergehenden Status noch offline war. Falls im Vorfeld Daten angelegt wurden, welche nur lokal gespeichert werden konnten, werden diese ausgelesen, auf dem Server gespeichert und in der lokalen Datenbank als synchronisiert gekennzeichnet.

Im zweiten Meilenstein wurde der Programmcode überarbeitet, um die Umsetzung der weiteren Funktionen, welche das Pflichtenheft für diesen Meilenstein vorsieht, zu erleichtern.

Im Folgenden ist es nunmehr nötig die Daten, die während der Zeit im Offline-Modus angelegt wurden, in der Methode `checkSync()` einzutragen. Unter der vorherigen Architektur hätte die Logik in jede Verbindungsoperation kopiert werden müssen. Dies hätte zu einer hohen Anzahl an Redundanzen im Code geführt. Diese sollten unbedingt umgangen werden. Deshalb ist diese zentralisierte Stelle zum Überprüfen der zu übertragenden Daten umgesetzt worden.

## 10.2. Anpassungen der Oberfläche

Die Oberflächen wurde dahingehend angepasst, dass ein durchgängiges Design, über die gesamte User Story hinweg, erkennbar ist. Auf diese wird im nächsten Abschnitt genauer eingegangen.

Die Oberfläche, insbesondere Schaltflächen und Dialoge wurden in diesem Schritt angepasst.<sup>1</sup> Dabei wurden zwei Designs umgesetzt: eine für den Login- und eine für den Registrieren-Button. Dazu war es nötig eine XML-Datei anzulegen, welche die Eigenschaften der Schaltfläche beschreibt (siehe Code-Beispiel 10.1):

```

1 <selector xmlns:android="http://schemas.android.com/apk/res/android">
2   <item android:state_pressed="false">
3     <layer-list>
4       <item android:right="5dp" android:top="5dp">
5         <shape>
6           <corners android:radius="2dp"/>
7           <solid android:color="#D6D6D6"/>
8         </shape>
9       </item>
10      <item android:bottom="2dp" android:left="2dp">
11        <shape>
12          <gradient android:angle="270" android:endColor="#4A6EA9" android:startColor="#4A6EA9"/>
13          <stroke android:width="1dp" android:color="#BABABA"/>
14          <corners android:radius="4dp"/>
15          <padding android:bottom="10dp" android:left="10dp" android:right="10dp" android:top="10dp"/>
16        </shape>
17      </item>
18    </layer-list>
19  </item>
20 </selector>

```

**Quelltext 10.1:** Design der Schaltfläche *Login*

Weiterhin wurden die Dialogfenster mit Animationen versehen, um eine zum Betriebssystem passende Verhalten zu erzeugen. Dazu musste, äquivalent zur Layout-Datei für die Schaltflächen, eine XML-Datei angelegt werden, welche die Bewegungen des Fensters beschreibt (siehe Code-Beispiel 10.2).

```

1 <?xml version="1.0" encoding="utf-8" ?>
2 <translate xmlns:android="http://schemas.android.com/apk/res/android"
3   android:fromXDelta="0"
4   android:toXDelta="100%p"
5   android:duration="500"/>

```

<sup>1</sup>ROCKET: Xamarin.Android Sample Projects.

### Quelltext 10.2: Dialog-Animation

Weiterhin wurden die Icons zur Anzeige des Verbindungsstatus ausgetauscht, um diese an das neue Design anzupassen. Als letztes wurde ein App-Icon hinzugefügt, welches auf dem Home-Screen des Endgerätes angezeigt wird.

## 10.3. Implementierung der Statistik

Um Fortschritte des Nutzers anzeigen zu können, wurde eine Übersicht mit den eingetragenen Trainingsleistungen implementiert. Diese ist für jede Übung über einen längeren Klick auf das Übungs-Feld erreichbar. In dem Balkendiagramm ist das Produkt aus dem Trainingsgewicht, der Wiederholungszahl und der Satzzahl dargestellt. Ein Screenshot der Oberfläche liegt dem Anhang bei (siehe Abbildung A.8)

Zur Darstellung wurde die *Xamarin-Extension BarChart* verwendet. Dieses Paket wurde in das Projekt eingebunden und kann durch einen einfachen Aufruf, mit der Übergabe eines Daten-Arrays, verwendet werden (siehe Code-Beispiel 10.3).

Zur Generierung der Diagramm-Daten waren Informationen über die Übung, den Trainingsplan und den *User* nötig. Diese Daten werden über einen Intent an die Activity *StatisticActivity* übergeben, in welcher daraufhin die benötigten Daten abgerufen werden (siehe Code-Beispiel 10.3).

```
1 protected async override void OnCreate(Bundle bundle)
2 {
3     try
4     {
5         base.OnCreate(bundle);
6         ooService = new OnOffService();
7
8         scheduleId = Intent.GetIntExtra("Schedule", 0);
9         exerciseId = Intent.GetIntExtra("Exercise", 0);
10        userId = Intent.GetStringExtra("User");
11        List<Practice> practices = new List<Practice>();
12    }
```



```

13     practices = await ooService.getAllPracticesAsync(userId, scheduleId,
14         exerciseId);
15
16     List<float> zahlen = new List<float>();
17     foreach (var item in practices)
18     {
19         zahlen.Add(Convert.ToSingle(item.Repetitions *
20             item.NumberOfRepetitions * item.Weight));
21     }
22
23     var data = zahlen.ToArray();
24     var chart = new BarChartView(this)
25     {
26         ItemsSource = Array.ConvertAll(data, v => new BarModel { Value = v
27             })
28     };
29     AddContentView(chart, new ViewGroup.LayoutParams(
30         ViewGroup.LayoutParams.FillParent,
31         ViewGroup.LayoutParams.FillParent));
32 }
33 catch (Exception ex){[...]}
```

**Quelltext 10.3:** Activity *StatisticActivity*

## 10.4. User Story der App

In der Abbildung 10.2 ist die Übersicht der Oberflächen zu erkennen. Größere Abbildungen sind im Anhang (siehe Kapitel A.3) zu finden. Um zu zeigen, wie der *Workflow* verläuft, sind ebenfalls die Verbindungen zwischen den Oberflächen eingezeichnet. Der Einstieg des Benutzers in die Android-App geschieht über die Startseite (siehe Abbildung A.2), welche die Auswahl zwischen Login und Registrierung darstellt. Diese Oberfläche wird nur für Nutzer angezeigt, welche sich noch nicht an der App angemeldet haben.

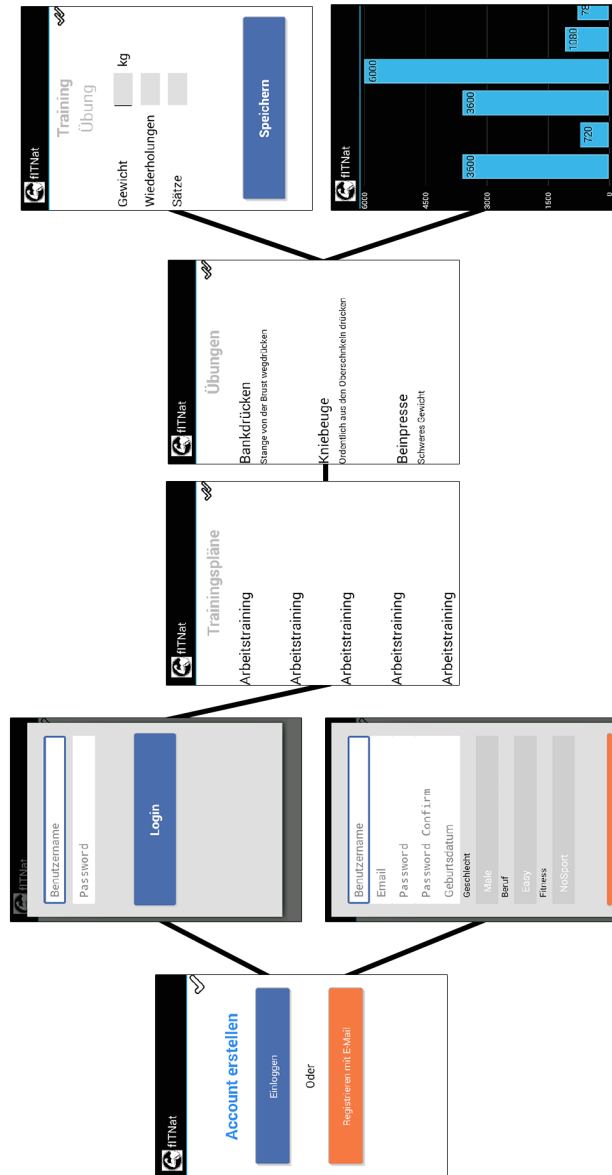


Abbildung 10.2.: User Story

Wählt der Benutzer die Registrierung, gelangt er zum Registrieren-Dialog (siehe Abbildung A.4). Um diese Funktion nutzen zu können, muss das Gerät eine Verbindung zum Server besitzen.

Wählt der Nutzer, statt der Registrierungsschaltfläche, die Schaltfläche zum Anmelden, erscheint der Login-Dialog (siehe Abbildung A.3). Um sich offline anmelden zu können, muss ein Nutzer sich einmal an der Applikation angemeldet haben, während diese eine Verbindung zum Web Service besessen hat. Dies ist nötig, da mit die App prüfen kann, ob die eingegebenen Daten serverseitig valide sind. Ist dies der Fall, werden sie lokal gespeichert, sodass sich ein Nutzer fortan auch lokal mit den gespeicherten Anmeldedaten autorisieren kann.

Nach einer erfolgreichen Anmeldung an der App, sei es offline oder online, werden die Trainingspläne des Benutzers als Liste angezeigt (siehe Abbildung A.5). Durch einen Klick auf den entsprechenden Plan, öffnet sich eine neue Seite mit den zu diesem Plan zugewiesenen Übungen (siehe Abbildung A.6). Auf dieser werden die Übungen mit Namen und Beschreibung aufgelistet. Der Benutzer hat nun zwei Möglichkeiten, um zu interagieren:

- Zum einen kann man durch das Auswählen einer Übung zu der Trainingsseite gelangen (siehe Abbildung A.7). Auf dieser Seite kann der Benutzer die Daten einer Trainingseinheit zu dieser Übung eintragen. Benötigt werden dazu das Gewicht, die Anzahl der Sätze und die Wiederholungszahl.
- Als zweite Möglichkeit kann, durch einen langen Klick auf eine Übung, eine Statistik zu den dazu eingetragenen Trainingseinheiten aufgerufen werden. Diese Statistik wird als Balkendiagramm angezeigt und gibt einen Überblick über die Leistungen des Benutzers. Der Index berechnet sich als Produkt der eingetragenen Werte aus Gewicht, Satzzahl und Wiederholungen.

Werden neue Daten eingetragen erscheinen diese, unabhängig vom Verbindungsstatus, sofort in der Statistik (siehe Abbildung A.8). Um in der Statistik alle Daten angezeigt zu bekommen, muss diese im Online-Modus aufgerufen werden, damit die Daten auf das Gerät geladen werden können. Wird die Statistik im Offline-Modus zum ersten Mal aufgerufen, werden nur die offline angelegten Trainingseinheiten angezeigt.

## 10.5. Fazit aus Meilenstein 2

Zusammenfassend kann festgehalten werden, dass der zweite Meilenstein zur Optimierung der nativen Android-Applikation beigetragen hat. Dies geschah sowohl in der Programmlogik, als auch durch die Anpassungen der Oberflächen.

So konnte, durch das Auslagern der Synchronisation, Last vom *UIThread* genommen werden. Die daraus folgenden Vorteile wurden bereits in Kapitel 7 vorgestellt.

Bei der Entwicklung eines Messeprototyps ist eine ansprechende Oberfläche von großer Bedeutung für die Akzeptanz durch den späteren Nutzer. Diese Anforderung wurde im ersten Meilenstein zurückgestellt, um einen Fokus auf den technischen Vergleich legen zu können. Nachdem nun im ersten Meilenstein eine für die Aufgabenstellung passende technische Grundlage entstanden ist, konnte sich in diesem Meilenstein auf die Optimierung des Aussehens fokussiert werden.

Die Statistik, ein Kann-Kriterium im Pflichtenheft (siehe Kapitel A.1), wurde zusätzlich umgesetzt, um die Nutzerfreundlichkeit weiter zu steigern. Als Nebeneffekt wurden weitere Kenntnisse im Umgang mit langen Klicks und der *BarChart-Extension* gewonnen.

Abschließend kann festgehalten werden, dass der zweite Meilenstein der nativen Applikation die letzten Verfeinerungen zum Prototypen gegeben hat.

Der Prototyp realisiert nun eine User Story, die alle benötigten Funktionen für die Entwicklung einer vollends ausgereiften App beinhaltet. So kann die Funktionalität des Speicherns einer Trainingseinheit auch auf das Speichern von Übungen und Trainingsplänen übertragen werden. Zudem kann das Abrufen von weiteren Daten, zum Beispiel für die Umsetzung von Administrator-Funktionen, mit den bestehenden Möglichkeiten umgesetzt werden.

# 11. Fazit

In diesem Kapitel wird eine Retrospektive des durchgeführten Projekts gegeben. Dabei wird zwischen dem Ergebnis bei der Erstellung der Applikationen und dem Erreichen der persönlichen Ziele unterschieden.

## 11.1. Ziele / Ergebnisse

Rückblickend kann das Projekt als ein Erfolg gesehen werden.

Aus dem Projekt ist eine umfassende Client-Server-Landschaft entstanden, welche die gesetzten Anforderungen an eine verlässliche mobile Applikation, sowohl in den Aspekten der Benutzerfreundlichkeit, als auch der Verlässlichkeit, erfüllt. Der Server sowie beide Clients haben in den jeweils erstellten Meilensteine alle Muss-Kriterien erfüllt. Teilweise konnten sogar optionale Kann-Kriterien umgesetzt werden. So ist es beispielsweise möglich, sich an der Web API zu registrieren oder in der nativen App eine Trainingsstatistik aufzubauen. Dies ist besonders deshalb bemerkenswert, da die Projektteilnehmenden einen Großteil der genutzten Techniken erst erlernen mussten. Alle im Projekt erzeugten Ressourcen können über das öffentliche Git-Repository unter <https://github.com/Fanuer/fIT> abgerufen werden.

## 11.2. Erkenntnisse

Der Erkenntnisgewinn dieser Arbeit ist beträchtlich.

Es wurden vorwiegend unbekannte und für die Autoren neue Technologien verwendet. Dies wurde gerade zu Anfang des Projekts jedoch auch als Risiko empfunden,

welches ein Problem in der Umsetzung hätte verursachen können. Diese Zweifel waren jedoch im nach hinein unbegründet. Die Einarbeitung geschah in den meisten Fällen reibungslos.

Die Tatsache, dass dieses Projekt von zwei Personen durchgeführt wurde, ergab sich als enormer Vorteil. Dies gestattete, punktuell Spezialwissen zu erwerben, was jeweils der anderen Person vermittelt werden musste. Dadurch wurde der Lerneffekt nochmals verstärkt und Zusammenhänge leichter vertieft.

Ebenfalls wurde von beiden Projektteilnehmer als sinnvoll erachtet, dass eine Komplettlösung für ein größeres Feld an Aufgaben, nämlich mobile Applikationen, geschaffen werden musste. Das meint, dass nicht nur eine Cache-Komponente für ein bestehendes System erstellt wurde, sondern sämtliche Komponente für die Realisierung der Systemlandschaft erstellt, verwaltet und veröffentlicht werden musste.

Somit konnte sich ein Einblick über alle anfallenden Arbeitsschritte zur Umsetzung einer mobilen Applikation geschaffen werden.

### 11.3. Ausblick

Die Grundlage für den Ausbau dieses Projektes zu einer marktreifen App ist gegeben:

Die Umsetzung aller wichtiger Funktionen wurde jeweils an mindestens einem Beispiel im Messeprototyp dargestellt und muss demnach nur noch auf die fehlenden Funktionalitäten übertragen werden.

Durch den nun tieferen Einblick in die Technologien sollte sich dieser Aufwand in Grenzen halten.

Auf Seiten des Servers müssen hauptsächlich Sicherheitsfunktionen weiterentwickelt werden, um die Web API öffentlich nutzbar zu machen. Zu diesen Funktionen zählen beispielsweise die Validierung von ClientIDs, die Einschränkung eingehender Anfragen durch CORS und die Signierung von Tokens.

Darüber hinaus kann eine Weiterentwicklung der App dazu führen, dass die Web API erweitert werden muss.

## **A. Anhang**

## A.1. Pflichtenheft

### Pflichtenheft

---

Zielsetzung des Projekts .....	2
Produkteinsatz .....	2
Anforderungsbeschreibung.....	2
Anforderungen an die <i>Proof of Concept</i> -Prototypen .....	2
Muss-Kriterien .....	2
Kann-Kriterien .....	3
Abgrenzungskriterien .....	3
Anforderungen an den Messe-Prototyp.....	3
Muss .....	3
Kann .....	3
Abgrenzungskriterien .....	3
Tests.....	3



### Zielsetzung des Projekts

Zielsetzung dieser Arbeit ist der Erkenntnisgewinn bei der Erstellung von zuverlässigen mobilen Anwendungen.

Um dieses Ziel zu erreichen, sollen die Unterschiede einer Entwicklung als native App zu der Entwicklung einer mobilen Web-Applikation geprüft werden. Als Forschungsobjekt dient dazu beispielhaft eine App, die es ermöglicht, erbrachte Leistungen beim Krafttraining festzuhalten.

Die festgehaltenen Daten sollen persistent gespeichert werden und jederzeit zur Verfügung stehen.

Dabei soll die Verlässlichkeit als Schwerpunkt dienen. Das meint, dass die Applikation weitestgehend unabhängig von äußeren Einflussfaktoren, wie beispielsweise einer vorhandenen Verbindung zu einem Server, funktioniert.

Nach der Gegenüberstellung der beiden Methoden, aus der jeweils ein rudimentärer Prototyp (*Proof of Concept*-Prototyp) hervorgehen soll, soll die günstigere der beiden Umsetzungen zu einem vollständigen Messe-Prototyp entwickelt werden.

Dieser soll eine User-Story durchspielen, welche beispielhaft die Umsetzung eines kompletten Anwendungsfalls zeigt.

### Produkteinsatz

Der Einsatz der erstellten Software beschränkt sich nur auf die Durchführung der Bachelor-Arbeit.

### Anforderungsbeschreibung

Da es sich bei dem Projektziel um ein zweistufiges Ziel handelt, werden auch die Anforderungen in zwei gesonderten Beschreibungen wiedergegeben.

#### Anforderungen an die *Proof of Concept*-Prototypen

In diesem Abschnitt sollen die Anforderungskriterien an die beiden *Proof of Concept*-Prototypen aufgezeigt werden.

#### Muss-Kriterien

- Ein Nutzer muss Daten einer Entität unabhängig von der bestehenden Serververbindung abrufen können
- Ein Nutzer muss sich unabhängig von der bestehenden Datenverbindung anmelden können
- Ein Nutzer kann Daten einer Entität unabhängig von der bestehenden Serververbindung bearbeiten
- Die Applikation muss es ermöglichen, lokal angelegte Daten mit denen des Servers zu synchronisieren
- Die Applikation muss ohne technische Kenntnisse bedienbar sein

### Kann-Kriterien

- Ein Nutzer kann neue Trainingsplan-Daten unabhängig von der bestehenden Serververbindung anlegen
- Der Benutzer kann sich über die Applikation registrieren
- Ein responsives Design soll eingebunden werden, um allen Benutzern eines Betriebssystems - unabhängig von dem Gerät - alle Funktionen zur Verfügung stellen zu können.
- Die Bedienung soll intuitiv sein

### Abgrenzungskriterien

- Der Nutzer kann keine Trainings oder Übungen anlegen, abrufen oder bearbeiten
- Aspekte der Sicherheit haben eine nachrangige Aufgabe, es soll im ersten Schritt die Möglichkeit der Umsetzung validiert werden

### Anforderungen an den Messe-Prototyp

In diesem Abschnitt sollen die Anforderungskriterien an den schlussendlichen Messe-Prototypen aufgezeigt werden.

### Muss

- Ein Nutzer muss Übungsdaten zu einem Trainingsplan abrufen können
- Ein Nutzer muss zu einer Übung Trainingsdaten abrufen
- Ein Nutzer muss zu einer Übung einen neuen Trainingsdatensatz anlegen können
- Die Anwendung muss alle Muss-Kriterien eines *Proof of Concept*-Prototypen erfüllen.

### Kann

- Für die letzten Datensätze eines Trainings stellt die Applikation eine grafische Statistik bereit
- Neu angelegte Trainingsdatensätze ändern eine Statistik, welche die letzten Daten grafisch aufbereitet
- Die Applikation unterstützt eine Benutzung von rollenbasierter Funktionszuweisungen
- Nutzer mit der Rolle *Administrator* haben die Möglichkeit, neue Übungen anzulegen.

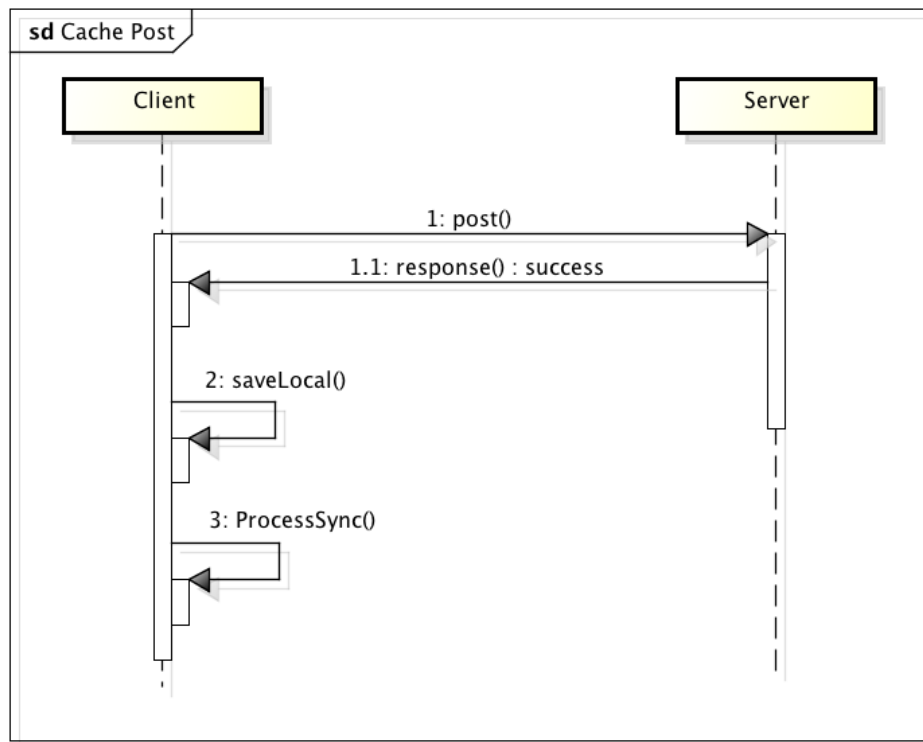
### Abgrenzungskriterien

- Nutzer dürfen nur Bereiche und Inhalte sehen, die für ihre Rolle relevant sind. Irrelevanter Inhalt darf nicht angezeigt werden.

### Tests

Da der Server als Kernkomponente eine besondere Aufgabe innehat, soll seine erwartete Funktionsweise gesondert neben den üblichen, projektbegleitenden Tests durch automatisierte Tests verifiziert werden. Für die Clients sind im Umfang dieser Projektarbeit Tests im Zuge der Implementierung ausreichend.

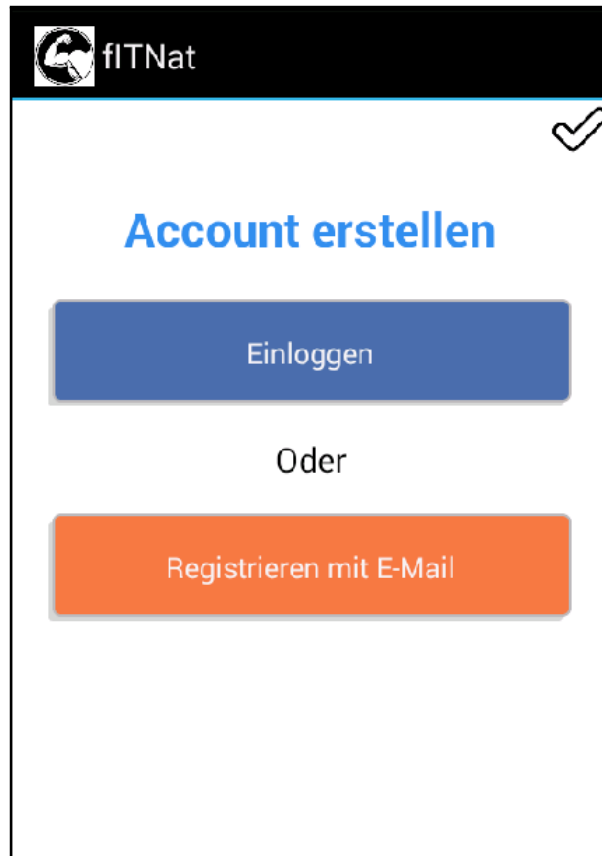
## A.2. Cache Post



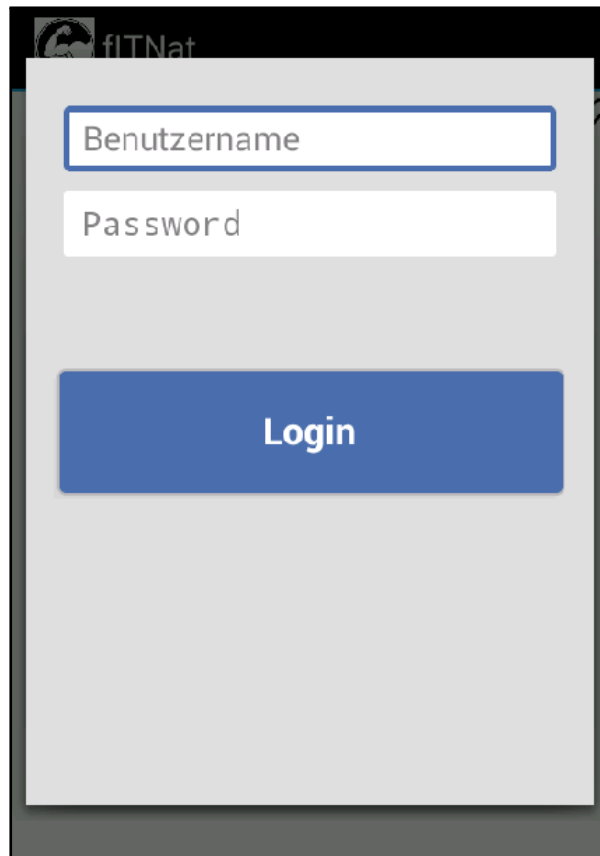
powered by Astah

Abbildung A.1.: Hochladen zum Server

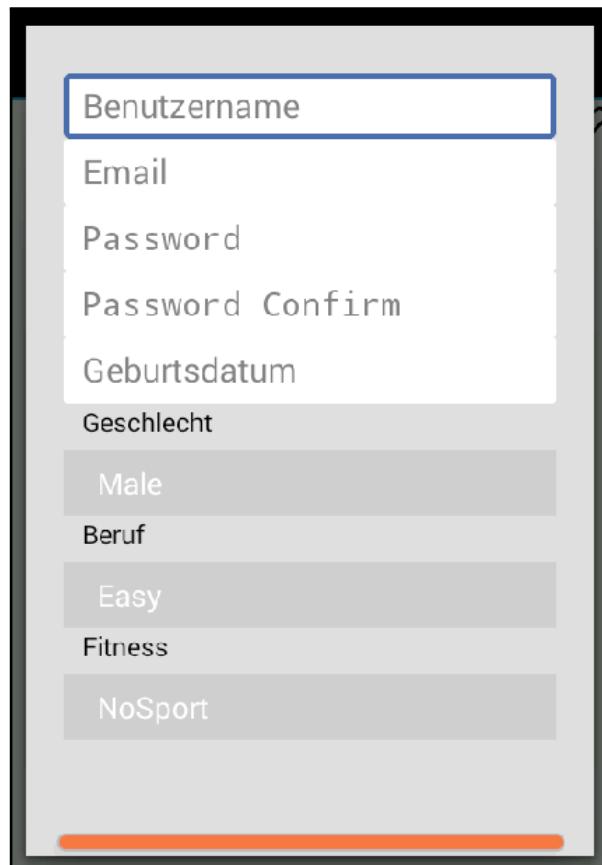
## A.3. User-Story in der nativen App



**Abbildung A.2.:** Startseite



**Abbildung A.3.:** Login



A registration form interface displayed on a mobile device. The form is contained within a white rectangular area with a thin blue border. It features several input fields and selection buttons. The fields are labeled: 'Benutzername' (highlighted with a blue border), 'Email', 'Password', 'Password Confirm', and 'Geburtsdatum'. Below these are three selection sections: 'Geschlecht' with a 'Male' button, 'Beruf' with an 'Easy' button, and 'Fitness' with a 'NoSport' button. An orange horizontal bar is visible at the bottom of the form area.

Benutzername

Email

Password

Password Confirm

Geburtsdatum

Geschlecht

Male

Beruf

Easy

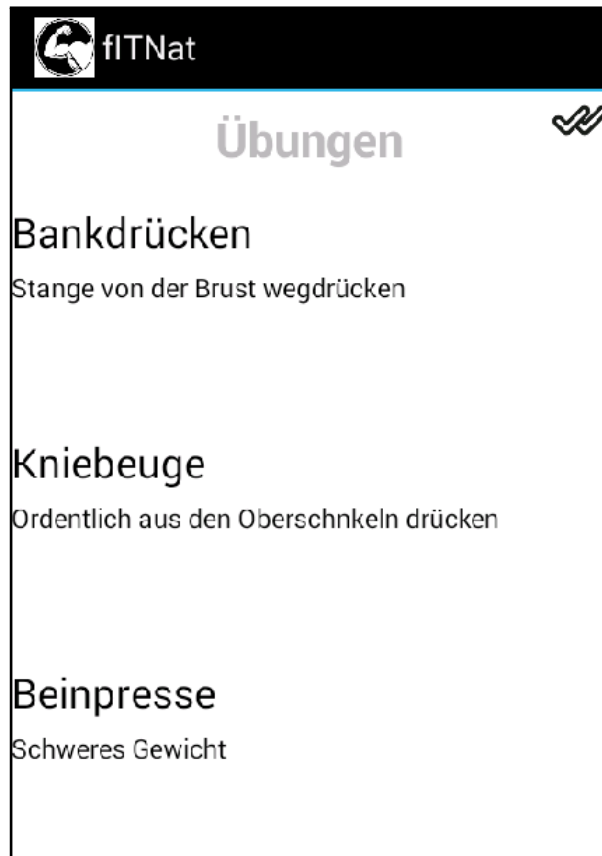
Fitness

NoSport

**Abbildung A.4.:** Registrierung



**Abbildung A.5.:** Übersicht der Trainingspläne



**Abbildung A.6.:** Übersicht der Übungen





fITNat

Training Übung

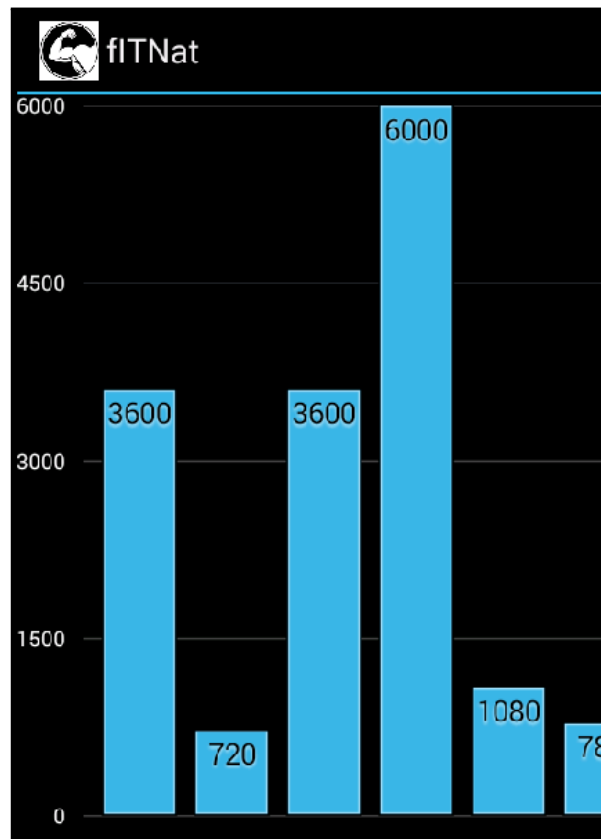
Gewicht  kg

Wiederholungen

Sätze

Speichern

Abbildung A.7.: Training



**Abbildung A.8.:** Statistik

# Glossar

## **.NET-Framwork**

Entwicklungs- und Laufzeitumgebung für Anwendungen unter Windows. Das Framework wurde von der Softwarefirma *Microsoft* entwickelt.

## **Android**

Mobiles Betriebssystem der Firma Google.

## **Android UI Toolkit**

Komponentensammlung zur Erstellung von Oberflächen für das mobile Betriebssystem Android.

## **App**

Kleine Programme/Applikationen für mobile Endgeräte.

## **Browser**

Ein Browser ist ein Programm zum Anzeigen von Internetseiten aus dem World Wide Web.

## **C#**

Eine von Microsoft entwickelte hohe Programmiersprache zur Umsetzung von Applikationen auf Grundlage des .Net-Frameworks.

## **CSS3**

Die aktuelle Version der Stylesheet-Sprache CSS.

## **Dirty Read**

Bezeichnet einen Zugriff auf eine Ressource, welche sich noch in Bearbeitung durch eine andere Komponente befindet. Somit wird ein altes Datum abgerufen..

## **DRAM**

Ein DRAM ist ein dynamischer RAM, der sehr langsam und kostengünstig ist.

## **Factory**

Das Factory-Entwurfsmuster ist ein Erzeugungsmuster. Eine Klasse, welche das Factory-Pattern benutzt, stellt Methoden zur Erzeugung von Objekten bereit. Diese Klasse ist daraufhin alleinig für die Erstellung der unterstützten Objekte verantwortlich. Der Vorteil liegt in der zentralen Anlaufstelle. Dadurch kann bestimmt werden, welche Programmstelle neue Objekte erzeugen soll und in welcher Art diese erzeugt werden.

## **Fat-Client**

Art eines Clients in einer Server-Client-Architektur. Dieser zeichnet sich dadurch aus, dass viele genutzte Funktionalitäten dezentral auf den Clients vorhanden sind. Dadurch sind weniger Anfragen an den Server nötig, was die Unabhängigkeit des Clients erhöht.

## **HTML5**

HTML5 meint die fünfte Version von HTML. Es bietet beispielsweise besondere Auszeichnungen für Kopf-, Fuß- und Navigationsbereiche.

## **iOS**

Mobiles Betriebssystem der Firma Apple.

## **Java**

Weit verbreitete objektorientierte Programmiersprache.

**Javascript**

Eine Programmierbrache, welche von Browsern für die clientseitige Ausführung von Code benutzt wird. Dadurch kein einer Webseite dynamisches Verhalten hinzugefügt werden..

**JSONP**

JSONP steht für JSON mit Padding. Es ist eine Möglichkeit zur Übertragung von JSON-Daten über Domaingrenzen hinweg.

**Linux**

Linux ist ein quelloffenes Betriebssystem, welches durch eine Community weiterentwickelt wird..

**Makro**

Kapselung von Code-Anweisungen zur Umsetzung einer Aufgabe.

**Markup**

Auszeichnung von Text, sodass zusätzliche Semantik geschaffen wird. Beispiele hierfür sind HTML und XML.

**Microsoft Entity Framework**

Ein OR-Mapper von *Microsoft*. Er ist besonders gut zur Kommunikation mit *MSSQL* geeignet.

**Mono**

plattformunabhängige Implementierung des .NET-Framwork.

**monolithisch**

Ein Betriebssystem wird als monolithisches bezeichnet, wenn es sich nicht nur um die Speicher- und Prozessverwaltung kümmert, sondern auch die benötigten Treiber zur Verfügung stellt..

## **NoSQL**

Steht für *Not only SQL*. NoSQL-Datenbanken besitzen keine klassischen relationalen Tabelleschemata. Dies erlaubt eine leichtere Speicherung von dynamisch strukturierten Daten. Es ist als ein strukturierter Datenspeicher zu verstehen, bei denen auf die Einträge per Index zugegriffen werden kann.

## **NuGet**

Eine Paketverwaltung für das .Net-Framework. Es erlaubt das Hinzufügen, Aktualisieren und Entfernen von Komponenten und deren Abhängigkeiten.

## **OR-Mapper**

Ein Framework zur Überführung von relationalen Tupeln in objekt-orientierte Objekte.

## **Polyfills**

Ein Codebaustein, welcher aktuelle Web Techniken in älteren Browsern verfügbar macht.

## **Reflection**

Die Fähigkeit eines Programm, seine eigene Struktur zu kennen und gegebenenfalls zu verändern.

## **Repository**

Das Repository-Entwurfsmuster kapselt die Datenschicht von der Applikationsschicht. Es bietet Schnittstellen an, mit dem die Applikationsschicht auf die Datenschicht zugreifen kann. Dadurch ist es später leichter, eine Datenschicht auszutauschen oder mehrere Datenquellen anzubinden.

## **Request**

Ein Request ist ein Aufruf an einen Knoten. Meist ein Aufruf an einen Server im Internet.

## **responsiv**

Eine Webseite, welche responsiv gestaltet wird, passt ihr aussehen abhängig vom aufrufenden Endgerät an. Diese Technik kommt besonders Geräten mit kleineren Displays wie Smartphones oder Tablets zu Gute.

## **Sandbox**

Isolierter Bereich, in dem Programm-Code ausgeführt werden kann, ohne dass dadurch die Außenwelt beeinflusst wird. Dies wird beispielsweise bei Virenprogrammen zur Prüfung von Dateien benutzt.

## **Separation of Concerns**

Bei diesem Design-Pattern wird darauf geachtet, dass sich Aufgabenbereiche nicht überschneiden. Jeder Teil eines Problems soll durch einen eigenen Teil gelöst werden.

## **Singleton**

Ein Entwurfsmuster, dessen Implementierung gewährleistet, dass ein Objekt einer Klasse nur genau ein Mal erzeugt wird. Wird dieses Objekt erneut benötigt, wird das bereits erzeugte Objekt zurückgegeben.

## **SRAM**

Ein SRAM ist ein statischer RAM, der eine sehr schnelle Zugriffszeit garantiert und Daten mit einem sehr geringen Stromaufwand speichern kann.

## **TCP**

TCP steht für Transmission Control Protocol. Dies ist ein verbindungsorientiertes Übertragungsprotokoll zum bidirektionalen Datentransport zwischen Computern.

## **Trigger**

Prozedur in einer Datenbank, welche beim Eintreten eines Ereignisses (z.B. neues Tupel angelegt) ausgeführt wird.

### **UserID**

Auch UID. Wird in einem Linux-basierten Betriebssystem zur eindeutigen Identifikation eines Systemnutzers benutzt.

### **User Story**

Ein Anwendungsfall einer Software. Hierbei ist es möglich, dass die Abarbeitung über mehrere Oberflächen und Aktionen umgesetzt werden kann.

### **XCode**

Eine von Apple entwickelte Entwicklungsumgebung zu Erstellung und Wartung von Anwendungen für die Betriebssysteme von Mac OS X, iOS und watchOS .



# Abbildungsverzeichnis

3.1. Abrufen vom Server . . . . .	22
4.1. Aufbau der Anwendung . . . . .	24
4.2. Anwendungsfälle des <i>Proof-of-Concept</i> -Prototyp . . . . .	25
4.3. Anwendungsfälle des Messeprototyp . . . . .	26
4.4. Datenbank-Entwurf . . . . .	27
5.1. Zuordnung von Produkten und Techniken zu Verantwortlichkeiten . . . . .	30
6.1. Swagger UI der Web Api . . . . .	44
6.2. Ressourcenzugriff durch OAuth2 . . . . .	45
6.3. Aufbau eines Testfalls . . . . .	50
7.1. Android Activity-Lifecycle . . . . .	56
7.2. Android Service-Lifecycle . . . . .	59
7.3. Xamarin Platform . . . . .	62
7.4. Sequenzdiagramm CRUD . . . . .	75
8.1. Veränderung der Statusanzeige . . . . .	85
8.2. Datenmodel zur Speicherung lokaler Daten . . . . .	87
10.1. Sequenzdiagramm <i>RecentOnline</i> . . . . .	96
10.2. User Story . . . . .	100
A.1. Hochladen zum Server . . . . .	109
A.2. Startseite . . . . .	110
A.3. Login . . . . .	111
A.4. Registrierung . . . . .	112

A.5. Übersicht der Trainingspläne . . . . .	113
A.6. Übersicht der Übungen . . . . .	114
A.7. Training . . . . .	115
A.8. Statistik . . . . .	116

# Tabellenverzeichnis

1.1. Arbeitsaufteilung . . . . .	10
2.1. Phasenplan . . . . .	14



# Quelltextverzeichnis

6.1. Basisinterface für DB-Repräsentationen . . . . .	39
6.2. Modelklasse für Trainingspläne . . . . .	39
6.3. POST-Methode zur Erstellung eines Trainingsplans . . . . .	41
6.4. Basis-Model-Klasse . . . . .	42
6.5. Implementierung des Tests 'Nutzer kann eigene Daten anpassen' . . . . .	50
7.1. Übertragen von Daten zwischen Activities . . . . .	66
7.2. Auslesen von Daten zwischen Activities . . . . .	67
7.3. Login über den <i>OnOffService</i> . . . . .	68
7.4. <i>UserModel</i> für die lokale Datenbank . . . . .	71
7.5. Login am Server . . . . .	72
7.6. Verbindungsüberprüfung . . . . .	73
7.7. Synchronisation der Offline-Daten . . . . .	75
8.1. Controller für die Navigationsleiste . . . . .	80
8.2. Navigation der Hauptseite erweitert um AngularJS-Markup . . . . .	81
8.3. Routing mit AngularJS . . . . .	83
8.4. Cache-Manifest-Datei . . . . .	89
10.1. Design der Schaltfläche <i>Login</i> . . . . .	97
10.2. Dialog-Animation . . . . .	97
10.3. Activity <i>StatisticActivity</i> . . . . .	98



# Literatur

- ANGULARJS: Online-Dokumentation: Controller, 2015, URL: <https://docs.angularjs.org/guide/controller> (besucht am 28.08.2015).
- DERS.: Online-Dokumentation: Dependency Injection, 2015, URL: <https://docs.angularjs.org/guide/di> (besucht am 28.08.2015).
- DERS.: Online-Dokumentation: Directiven, 2015, URL: <https://docs.angularjs.org/guide/directive#what-are-directives-> (besucht am 28.08.2015).
- DERS.: Online-Dokumentation: Interceptor, 2015, URL: [https://docs.angularjs.org/api/ng/service/\\$http#interceptors](https://docs.angularjs.org/api/ng/service/$http#interceptors).
- DERS.: Online-Dokumentation: Module \$q, 2015, URL: [https://docs.angularjs.org/api/ng/service/\\$q](https://docs.angularjs.org/api/ng/service/$q) (besucht am 28.08.2015).
- DERS.: Online-Dokumentation: Module \$route, 2015, URL: [https://docs.angularjs.org/api/ng/service/\\$route](https://docs.angularjs.org/api/ng/service/$route) (besucht am 28.08.2015).
- ANGULARJS: Online-Dokumentation: Modules, 2015, URL: <https://docs.angularjs.org/guide/module> (besucht am 28.08.2015).
- ANGULARJS: Online-Dokumentation: Scopes, 2015, URL: <https://docs.angularjs.org/guide/scope> (besucht am 28.08.2015).
- DERS.: Online-Dokumentation: Services, 2015, URL: <https://docs.angularjs.org/guide/services> (besucht am 28.08.2015).
- ANGULARJS: Testability Built-in, 2015, URL: <https://angularjs.org/#testability> (besucht am 08.09.2015).
- ARCITURA EDUCATION INC.: Service Data Forward Cache, 2015, URL: [http://soapatterns.org/candidate\\_patterns/service\\_data\\_forward\\_cache](http://soapatterns.org/candidate_patterns/service_data_forward_cache) (besucht am 03.09.2015).

- ATLASSIAN: Understanding JWT, 2014, URL: <https://developer.atlassian.com/static/connect/docs/latest/concepts/understanding-jwt.html> (besucht am 27.08.2015).
- BECKER, ARNDT und PANT, MARCUS: Android 2 - Grundlagen und Programmierung, 2. Aufl., Heidelberg 2010.
- BLUESMOON: Flowchart showing Simple and Preflight XHR, Aug. 2015, URL: [https://upload.wikimedia.org/wikipedia/commons/c/ca/Flowchart\\_showing\\_Simple\\_and\\_Preflight\\_XHR.svg](https://upload.wikimedia.org/wikipedia/commons/c/ca/Flowchart_showing_Simple_and_Preflight_XHR.svg) (besucht am 27.08.2015).
- BOOTH, DAVID u. a.: Web Services Architecture, Feb. 2004, URL: <http://www.w3.org/TR/ws-arch/> (besucht am 26.08.2015).
- BRAMSKI: Angular-IndexedDB, 2015, URL: <https://github.com/bramski/angular-indexedDB> (besucht am 31.08.2015).
- CANIUSE.COM: Can I Use: Cross-Origin Resource Sharing, 2015, URL: <http://caniuse.com/#feat=cors> (besucht am 27.08.2015).
- DERS.: Can I Use: IndexedDB, 2015, URL: <http://caniuse.com/#feat=indexeddb> (besucht am 31.08.2015).
- DERS.: Can I Use: Offline web applications, 2015, URL: <http://caniuse.com/#feat=offline-apps> (besucht am 31.08.2015).
- DATACOM BUCHVERLAG GMBH: Store-and-Forward-Verfahren, 2015, URL: <http://www.itwissen.info/definition/lexikon/Store-and-Forward-Verfahren-SF-store-and-forward.html> (besucht am 03.09.2015).
- DEUTSCHSPRACHIGE ANWENDERVEREINIGUNG TEX E.V.: Was ist LaTeX?, 2015, URL: <http://www.dante.de/tex/WasIstLaTeX.html> (besucht am 07.09.2015).
- DYKSTRA, TOM: Getting Started with Entity Framework 6 Code First using MVC 5, 2015, URL: <https://www.asp.net/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application> (besucht am 27.08.2015).
- GIT: Los geht's - Git Grundlagen, 2015, URL: <https://git-scm.com/book/de/v1/Los-geht%E2%80%99s-Git-Grundlagen> (besucht am 08.09.2015).
- GUMMER, HEINZ PETER und SOMMER, MANFRED: Einführung in die Informatik, 10. Aufl., München 2013.
- IDC: Prognose zu den Marktanteilen der Betriebssysteme am Absatz vom Smartphones weltweit in den Jahren 2015 und 2019, Sep. 2015, URL: <http://de.statista.com/statistik/daten/studie/182363/umfrage/prognostizierte-marktanteile-bei-smartphone-betriebssystemen/> (besucht am 01.09.2015).



- JOUDEH, TAISEER: AngularJS Token Authentication using ASP.NET Web API 2, Owin, and Identity, Juni 2014, URL: <http://bitoftech.net/2014/06/09/angularjs-token-authentication-using-asp-net-web-api-2-owin-asp-net-identity/> (besucht am 28.08.2015).
- DERS.: ASP.NET Identity 2.1 Roles Based Authorization with ASP.NET Web API, März 2015, URL: <http://bitoftech.net/2015/03/11/asp-net-identity-2-1-roles-based-authorization-authentication-asp-net-web-api/> (besucht am 27.08.2015).
- DERS.: ASP.NET Web API Documentation using Swagger, Aug. 2014, URL: <http://bitoftech.net/2014/08/25/asp-net-web-api-documentation-using-swagger/> (besucht am 27.08.2015).
- DERS.: Enable OAuth Refresh Tokens in AngularJS App using ASP .NET Web API 2, and Owin, Juli 2014, URL: <http://bitoftech.net/2014/07/16/enable-oauth-refresh-tokens-angularjs-app-using-asp-net-web-api-2-owin/>.
- DERS.: Implement OAuth JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1, Feb. 2015, URL: <http://bitoftech.net/2015/02/16/implement-oauth-json-web-tokens-authentication-in-asp-net-web-api-and-identity-2/> (besucht am 27.08.2015).
- DERS.: Token Based Authentication using ASP.NET Web API 2, Owin, and Identity, Juli 2015, URL: <http://bitoftech.net/2014/06/01/token-based-authentication-asp-net-web-api-2-owin-asp-net-identity/> (besucht am 27.08.2015).
- KANTOR, ILYA: JavaScript is single-threaded, 2011, URL: <http://javascript.info/tutorial/events-and-timing-depth#javascript-is-single-threaded> (besucht am 28.08.2015).
- KOWAL, KRIS: Documentation q, 2015, URL: <http://documentup.com/kriskowal/q/> (besucht am 28.08.2015).
- KURTZ, JAMIE und WORTMAN, BRIAN: ASP.NET Web API 2: Building a REST Service from Start to Finish, 2. Aufl., New York 2014.
- MICROSOFT: SQL Server 2014, 2015, URL: <http://www.microsoft.com/de-de/server-cloud/products/sql-server/> (besucht am 01.09.2015).
- DERS.: Visual Studio, 2015, URL: <https://www.visualstudio.com/> (besucht am 01.09.2015).
- DERS.: Was ist Azure?, 2015, URL: <http://azure.microsoft.com/de-de/overview/what-is-azure/> (besucht am 01.09.2015).

- OPEN HANDSET ALLIANCE: Activities, Aug. 2015, URL: <https://developer.android.com/guide/components/activities.html> (besucht am 26.08.2015).
- DERS.: Application Fundamentals, Aug. 2015, URL: <https://developer.android.com/guide/components/fundamentals.html> (besucht am 26.08.2015).
- DERS.: Content Providers, Aug. 2015, URL: <https://developer.android.com/guide/topics/providers/content-providers.html> (besucht am 26.08.2015).
- DERS.: Intents and Intent Filters, Aug. 2015, URL: <https://developer.android.com/guide/components/intents-filters.html> (besucht am 26.08.2015).
- DERS.: Processes and Threads, Aug. 2015, URL: <https://developer.android.com/guide/components/processes-and-threads.html> (besucht am 26.08.2015).
- DERS.: Services, Aug. 2015, URL: <https://developer.android.com/guide/components/services.html>.
- DERS.: System Permissions, Aug. 2015, URL: <https://developer.android.com/guide/topics/security/permissions.html> (besucht am 26.08.2015).
- DERS.: User Interface, Aug. 2015, URL: <https://developer.android.com/guide/topics/ui/overview.html#Layout> (besucht am 26.08.2015).
- ROCKET, JOE: Xamarin.Android Sample Projects, URL: <http://cforbeginners.com/XamarinProjects.html> (besucht am 06.09.2015).
- ROUSE, MARGARET: Cache (Computing) Definition, Apr. 2015, URL: <http://searchstorage.techtarget.com/definition/cache> (besucht am 02.09.2015).
- DERS.: Disk Cache Definition, Apr. 2015, URL: <http://searchstorage.techtarget.com/definition/disk-cache> (besucht am 02.09.2015).
- SAUMONT, PIERRE-YVES: Do it in Java 8: Automatic memoization, Sep. 2015, URL: <https://dzone.com/articles/java-8-automatic-memoization> (besucht am 03.09.2015).
- SCHACHERL, ROMAN: Xamarin 3: Plattformübergreifende App-Entwicklung, Aug. 2015, URL: <https://entwickler.de/online/xamarin-3-plattformuebergreifende-app-entwicklung-161845.html> (besucht am 28.08.2015).
- SCHMIDT, HOLGER: Anzahl der Smartphone-Nutzer in Deutschland in den Jahren 2009 bis 2015 (in Millionen), Juni 2015, URL: <http://de.statista.com/statistik/daten/studie/198959/umfrage/anzahl-der-smartphonenuutzer-in-deutschland-seit-2010/> (besucht am 26.08.2015).
- SQLITE-TEAM: About SQLite, Aug. 2015, URL: <http://www.sqlite.org/about.html> (besucht am 28.08.2015).

- STEYER, MANFRED und SOFTIC, VILDAN: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, Köln 2015.
- SWAGGER: Swagger, 2015, URL: <http://swagger.io/> (besucht am 27.08.2015).
- THE APACHE SOFTWARE FOUNDATION: HTTP Proxy Caching, 2014, URL: <https://docs.trafficserver.apache.org/en/5.3.x/admin/http-proxy-caching.en.html> (besucht am 02.09.2015).
- TILKOV, STEFAN u. a.: REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web, 3. akt. u. erw. Aufl., Heidelberg 2013.
- TWINCODERS: SQLite-Net Extensions, Juli 2015, URL: <https://bitbucket.org/twincoders/sqlite-net-extensions> (besucht am 06.09.2015).
- TWITTER: Bootstrap Online Documentation, 2015, URL: <http://getbootstrap.com/css/#grid> (besucht am 28.08.2015).
- W3-TECH: Usage statistics and market share of AngularJS for websites, 2015, URL: <http://w3techs.com/technologies/details/js-angularjs/all/all> (besucht am 28.08.2015).
- W3C: HTML5 Application Cache, 2015, URL: [http://www.w3schools.com/html/html5\\_app\\_cache.asp](http://www.w3schools.com/html/html5_app_cache.asp) (besucht am 31.08.2015).
- WASSON, MIKE: Attribute Routing in ASP.NET Web API 2, Jan. 2014, URL: <http://www.asp.net/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2> (besucht am 27.08.2015).
- DERS.: Model Validation in ASP.NET Web API, 2012, URL: <http://www.asp.net/web-api/overview/formats-and-model-binding/model-validation-in-aspnet-web-api> (besucht am 27.08.2015).
- XAMARIN INC.: Create native iOS, Android, Mac and Windows apps in C #, Aug. 2015, URL: <http://xamarin.com/platform> (besucht am 28.08.2015).
- DERS.: Cross-Platform - Application Fundamentals, Aug. 2015, URL: [http://developer.xamarin.com/guides/cross-platform/application\\_fundamentals/](http://developer.xamarin.com/guides/cross-platform/application_fundamentals/) (besucht am 28.08.2015).
- DERS.: Understanding Android API Levels, Aug. 2015, URL: [http://developer.xamarin.com/guides/android/application\\_fundamentals/understanding\\_android\\_api\\_levels/](http://developer.xamarin.com/guides/android/application_fundamentals/understanding_android_api_levels/) (besucht am 28.08.2015).
- DERS.: Using SQLite.NET ORM, 2015, URL: [https://developer.xamarin.com/guides/cross-platform/application\\_fundamentals/data/part\\_3\\_using\\_sqlite\\_orm/](https://developer.xamarin.com/guides/cross-platform/application_fundamentals/data/part_3_using_sqlite_orm/) (besucht am 06.09.2015).