Bachelorthesis

Verlässliche mobile Anwendunge

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, 4. September 2015

Inhaltsverzeichnis

Aufgabenstellung

Mobile Applikationen sind im täglichen Leben allgegenwärtig.

Eine Herausforderung bei diesen Anwendungen ist es, dass sie verlässlich funktionieren müssen, da ansonsten ein Schaden auftritt, welcher sogar lebensbedrohlich- oder zumindest finanziell sein 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. Übersicht wer was gemacht hat

1. Einleitung

In diesem Kapitel wird das grundlegende Problem und die daraus resultierende Aufgabenstellung erläutert.

1.1. Problemstellung

Momentan besitzen 57% der Deutschen ein Smartphone. Somit hat sich die Zahl der Smartphone-Nutzer seit Ende 2011 mehr als verdoppelt.[1]
Durch die verstärkte Nutzung geraten Apps immer mehr in den Fokus. Applikationen haben sich im Laufe der Zeit im Alltag ausgebreitet und sind mittlerweile für den Endnutzer unverzichtbar geworden. 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 WLANs in Deutschland abgeschlossen ist, benötigt man eine gute Verbindung über seinen Netzbetreiber. Diese ist aber noch nicht vollständig und ausreichend im ganzen Land verfügbar.
Deshalb ist es notwendig, dass die Apps versuchen Verbindungsabbrüche für den Benutzer zu überbrücken. Dabei besteht die Möglichkeit einer kurzzeitigen Zwischenspeicherung von Daten, die vom Benutzer eingesehen oder verwendet werden können, solange die Internetverbindung nicht bereitsteht. Änderungen, die in dieser Zeit gemacht werden, sollen auch aufgenommen und später zur Verfügung gestellt werden damit man auf all seinen Endgeräten einen einheitlichen Stand der Daten hat.

[1] **?**, .

Zur Umsetzung dieser Anforderungen können verschiedene Möglichkeiten genutzt werden. Die beiden verbreitetsten Methoden sind native oder Web-Apps.

1.2. Zielsetzung

Ziel dieser Arbeit soll es sein, zwei unterschiedliche verlässliche Applikationen zu entwerfen, umzusetzen und im Anschluss zu testen. Diese sollen es ermöglichen 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. Dafür soll ein Windows-Server implementiert (siehe Kapitel 6) und verwendet werden.
Bei den Applikationen wird während der Entwicklungsphase entschieden, welche der beiden Apps zu einem lauffähigen Messeprototypen weiterentwickelt wird. Diese Einschätzung kann jedoch erst getroffen werden, wenn verschiedene Umsetzungen in den einzelnen Applikationen getestet 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 Smartphone eine Verbindung zum Server hat oder nicht.

1.3. Vorgehensweise

Zum Erreichen der Ziele wird zuallererst ein Überblick über die Grundlagen der Umsetzung gegeben. Dabei werden dann schon die ersten Techniken vorgestellt, die für die Implementierung verwendet werden sollen. Folgend wird die allgemeine Architektur des Systems, bestehend aus den beiden Applikationen und dem Server, erläutert, um den Gesamtzusammenhang dieses Projektes in Gänze überblicken zu können. Darauf aufbauend wird jeweils detaillierter auf die Umsetzungen der Apps und des Servers, sowie die dabei verwendeten Technologien eingegangen. Daraufhin werden

die Applikationen verglichen und entschieden, welche der beiden zu einem Messeprototypen weiterentwickelt wird. Abschließend wird die Erweiterung zum Prototypen vorgestellt und ein Rückblick auf das gesamte Projekt gegeben.

2. Problemanalyse

In diesem Kapitel soll das vorher grob geschilderte Problem analysiert, konkrete Ziele definiert und Entscheidungen für das weitere Vorgehen bei der Umsetzung getroffen werden. Darauf aufbauend werden die grundsätzlichen Komponenten und deren Funktionsweise beschrieben.

Das Problem des Sachverhalts liegt darin, dass die Fitness-Anwendungen auch dann noch benutzbar sein sollen, wenn keine Verbindung zum Internet, speziell zum benötigten Server, besteht. Dafür müssen die Applikationen ausgelegt und vorbereitet werden. Sei es durch das Unterbinden von Funktionen oder das Speichern von bereits erhaltenen Daten, um diese dem Benutzer dann für die weitere Verwendung zur Verfügung stellen zu können.
Weiterhin gibt es Unterschiede in der Auswahl der lokal zu speichernden Daten. Auf der einen Seite können alle Daten, die interessant sind, automatisch von der Anwendung für den Benutzer hinterlegt werden. Zum anderen kann es die Möglichkeit für den Benutzer geben, bestimmte Daten offline verfügbar zu machen.
Zu beachten ist darüber hinaus, dass die Daten, die ohne Internetverbindung angelegt werden, wieder zum Server synchronisiert werden müssen, um Benutzereingaben zentral persistent speichern zu können. In diesem Anwendungsfall sollen Trainingsdaten erfasst und gespeichert werden.
Die Daten sollen für verschiedene Benutzer, die sich an dem Gerät anmelden, gespeichert werden. Des Weiteren sollen Benutzer nur Funktionen ausführen können, zu denen sie auch autorisiert sind.

Konkret kann daraus geschlossen werden, dass die Anwendungen mit einem Mechanismus ausgestattet sein müssen, der das lokale Zwischenspeichern von Informationen unterstützt. Damit soll das Abrufen von Daten im *Offline*-Modus ermöglicht werden. Des Weiteren soll es *offline* möglich sein, Daten anzulegen und diese sollen dann mit dem Server synchronisiert werden, wenn wieder eine Verbindung besteht.

Ziel soll es sein, zwei Fitness-Applikationen zu entwickeln, die mit einem selbst entwickelten Server kommunizieren (siehe Kapitel 6). Während der Kommunikation muss festgestellt werden, wann die Kommunikation abbricht und dementsprechend müssen die Applikationen das Verhalten vom *Online*- zum *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, können 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 Anzeigen. Deshalb werden erhaltene Datensätze auch lokal abgelegt. Wenn der Benutzer nun Daten anlegt, muss dies zum einen machbar sein, zum anderen müssen die Daten auch für die Applikation als *Offline*-Daten erkennbar in dem lokalen Speicher sein. Deshalb müssen Daten besonders gekennzeichnet werden.
Wenn die Verbindung zwischen Server und *Client* gerade wieder hergestellt ist, müssen lokal angelegte Daten zum Server übertragen werden. In diesem Schritt ist dann die lokale Kennzeichnung dieser Datensätze von Vorteil. Der Server muss diese Daten dann annehmen und selbst in einer Datenbank speichern, um denselben Stand zu haben wie die Applikation.

Zur Umsetzung werden zwei mobile Applikationen ausgewählt. Diese sind besonders im Einsatz der App (bspw. im Fitnessstudio) von Vorteil.
Zum einen wird eine *Single-Page-Application* und zum anderen eine Android-Applikation entwickelt.
Die **SPA!**[1] wird als *Homepage* im Browser umgesetzt. Dabei wird auf Responsivität

[1] **SPA!**

geachtet, um das kleine *Display* von mobilen Endgeräten nicht auszuschließen, da die beiden Applikationen vergleichbar auf mobilen Geräten funktionieren sollen.
Android wird als Plattform für die native App ausgewählt, um die Vorteile des offenen Systems nutzen zu können. So ist es beispielsweise möglich die entwickelten Apps ganz einfach auf einem Testsystem zu installieren, ohne - wie bei Apples iOS 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 (siehe (**?**,)) und die App kann einen größeren Anklang finden.

Eine der beiden Applikationen wird im Laufe der Arbeit ausgewählt und zu einem rudimentären Messe-Prototypen weiterentwickelt. Die Entscheidung in diesem Falle wird im Implementierungs-Prozess getroffen, um bis dahin die Vor- und Nachteile der beiden Möglichkeiten kennenzulernen. Die andere App wird jedoch trotzdem die oben genannten Grundfunktionen bereitstellen.

3. Grundlagen

In diesem Kapitel wird eine Übersicht über die verschiedenen Funktionsweisen gegeben. Darauf aufbauend wird dann herausgestellt, welche Art von Speicherung in diesem Projekt umgesetzt wird. Dabei wird auch auf die Unterschiedliche Auffassung des Begriffes *Cache* eingegangen und die Unterschiede werden erläutert.

3.1. Definition eines Caches

Ein *Cache* wird im Allgemeinen als eine Speicherregion oder Puffer verstanden, die besonders schnell erreichbar sind. Darin werden oft verwendete Daten gespeichert, um höheren Speicherverbrauch gegen einen Performancegewinn zu tauschen.
Caches werden in verschiedenen Umgebungen eingesetzt. Es wird zwischen *Memory Cache*, *Internet Browser Cache*, *Disk Caching* und *Server Caching* unterschieden.
Der *Memory Cache* wird in Computern verwendet, um den sehr schnellen SRAM[1] des Rechners auszunutzen. Diese Art macht es sich zunutze, dass Programme immer dieselben Daten oder Befehle ausführen. Diese Ergebnisse werden dann vom Betriebssystem in diesem *Cache* gespeichert, um zum Beispiel darauf aufbauend schnellere Berechnungen vollziehen zu können. Dieser Speicher wird dann dem dazu relativ langsamen DRAM[2] vorgezogen.[3]
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 man dann wieder zurück navigiert, kann der *Browser* viele Dateien

[1] Statischer RAM
[2] Dynamischer RAM
[3] **?**, Vgl..

wiederverwenden und muss nicht mehr die gesamte Seite nachladen. Diese Variante wird auch *Read Cache* genannt.[4]
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. Der Bereich kann sich jedoch auch im RAM[5] des Computer befinden.[6]
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 dann diese Seite aufruft, wird die Seite aus dem *Cache* zurückgegeben und der *Request* muss nicht wieder über das Internet geleitet werden, sondern wird direkt im internen Netzwerk beantwortet.[7]

3.1.1. Funktionsweisen von Caches

Caches können in zwei unterschiedliche Funktionsweisen unterteilt werden, die im Folgenden genauer vorgestellt werden.

3.1.1.1. Store Forward

Das *Store and Forward*-Prinzip ist eine spezielle Art des *Cachings*, das Netze überbrücken soll, welche Verzögerungen tolerieren. Gegensätzlich dazu sind die Techniken *Streaming* und Internettelefonie, die keine Verzögerungen tolerieren. Gleichzusetzen ist diese Technik mit der TCP[8]-Übertragung, die auch zwischengespeichert werden kann. Der Vorteil dieser Technik besteht darin, dass eine Zwischenstation die übertragenen Daten speichert, auf Integrität prüft und, wenn gewünscht, weiterleitet.[9]
Diese Technik kann auch in einem System untergebracht werden. So soll ein Datum auf der Festplatte gespeichert werden, wird dazu jedoch sicherheitshalber im Puffer zwischengespeichert. Direkt danach wird dieses Datum abgerufen, ist aber noch nicht auf die Festplatte geschrieben. Dann kann das Datum aus dem Puffer geladen

[4] **?**, .
[5] Random Access Memory
[6] **?**, .
[7] **?**, .
[8] Transmission Control Protocol
[9] **?**, .

werden und die geringe Geschwindigkeit der Festplatte wird dem Anwender nicht bewusst.[10]

3.1.1.2. 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.Cache-Memoization

3.1.2. Unsere Definition eines Caches

Im Folgenden wird der Begriff des *Caches* als eine abgewandelte Technik zur lokalen Zwischenspeicherung von Daten auf einem mobilen Gerät verstanden. Die allgemeine Definition geht bei einem *Cache* von einer Perfomrancesteigerung aus (siehe Kapitel 3.1), in diesem Projekt wird das Speichern von Daten jedoch dazu verwendet, um Daten auch offline zur Verfügung zu haben. 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, jedoch 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.

[10] **?**, .

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 4.2 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 des Caches

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 Daten dann auch 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. Funktionsweise des Caches

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 lokal direkt zu speichern. Daten, die abgerufen werden, auch lokal zu speichern. 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 folgenden Sequenzdiagramm ist der Ablauf für die Logik des *Caches* bei einem *Get*-Aufruf an den Server zu sehen.

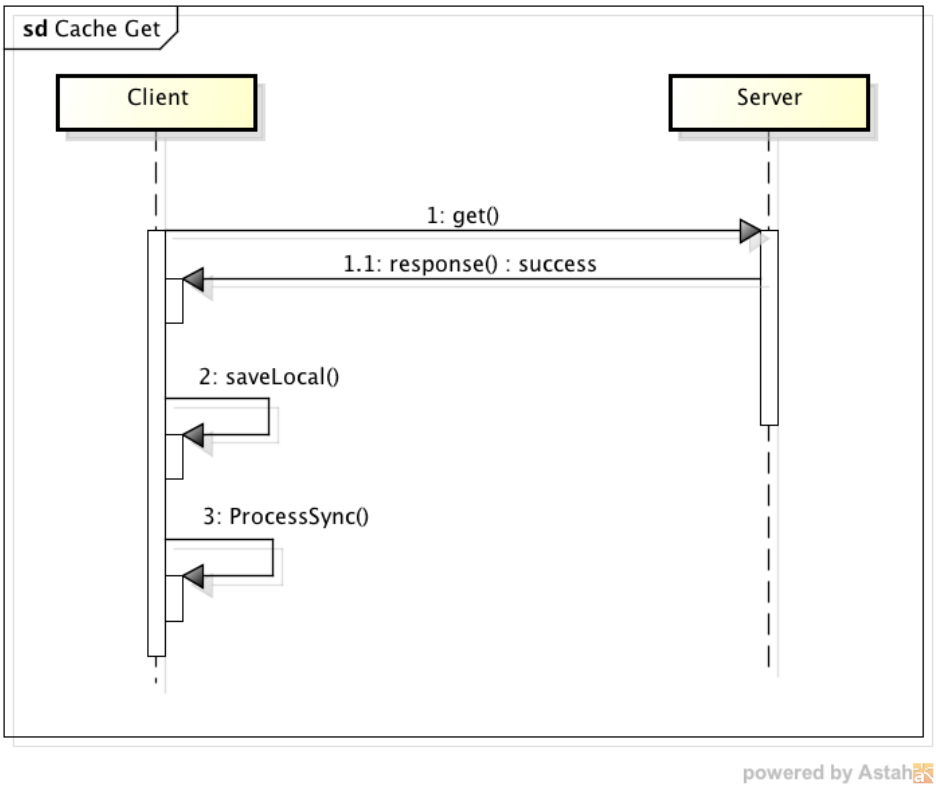

Abbildung 3.1.: Abrufen vom Server

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 **??**).

4. Architektur

In diesem Kapitel werden die architektonischen Randbedingungen für die Entwicklung der Applikation beschrieben. Hierzu zählt welche Anwendungsfälle in den späteren Prototypen und im schlussendlichen Messeprototyp gegeben sein muss. Daraus resultiert der Aufbau der Datenbank und die schlussendliche Systemarchitektur. Als Grundlage dient das Pflichtenheft. Dieses liegt dieser Arbeit gesondert im Anhang bei (siehe Anhang A.3).

4.1. Anwendungsfälle

Das Pflichtenheft sieht eine Unterteilung des Projekts in zwei aufeinanderfolgenden Meilensteine vor. Hierbei werden erst Proof-of-Concept-Prototypen entwickelt. Anschließend wird ein Prototyp zum Messe-Prototypen weiterentwickelt.
Für diese beiden Prototypen müssen andere bzw. erweiterte Anwendungsfälle implementiert werden. Darum werden nachfolgend für die beiden Implementierungsschritte die Anwendungsfälle einzeln aufgeschlüsselt.

4.1.1. Anwendungsfälle für Meilenstein 1 (Proof-of-Concept-Phase)

Aus dem Pflichtenheft ergeben sich folgende Anwendungsfälle für die erste Phase des Projekts:

- 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

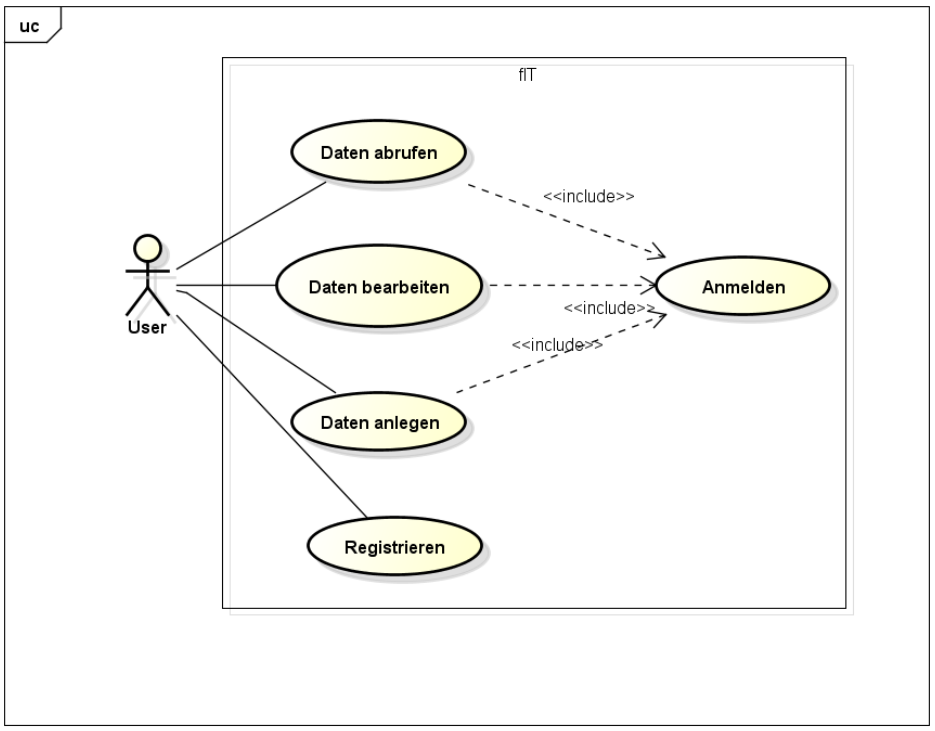

Abbildung 4.1.: Use-Cases Proof-of-Concept

4.1.2. Anwendungsfälle für Meilenstein 2 (Messe-Prototyp-Phase)

Für den Meilenstein 2 werden die bereits vorgestellten Anwendungsfälle weiter verfeinert. Daraus ergeben sich folgende Anwendungsfälle:

- 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 Trainings 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

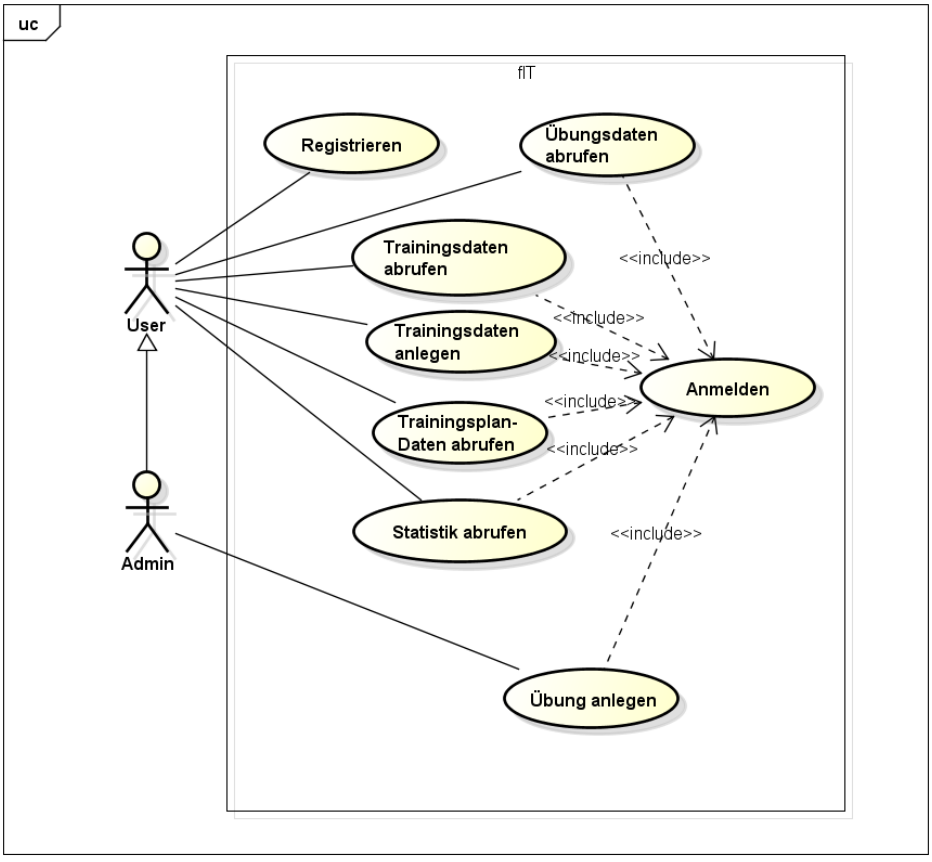

Abbildung 4.2.: Use-Cases Messeprototyp

4.2. Datenbank-Entwurf

Aus den definierten Anwendungsfälle ergibt sich die Struktur für die Datenbank. Als Grundlage werden die Anwendungsfälle des zweiten Meilensteins genutzt, um spätere Anpassungen nach Beendigung des ersten Meilensteins zu vermeiden.

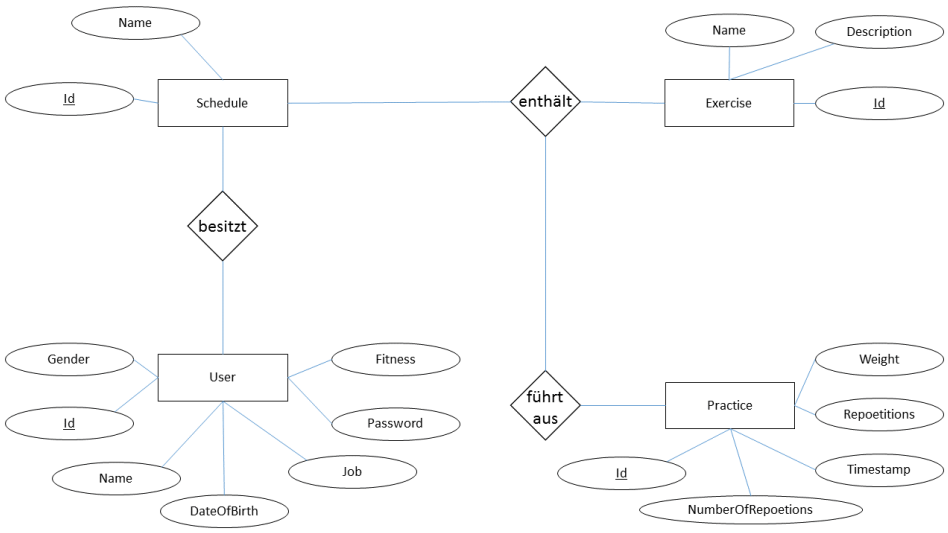

Abbildung 4.3.: Datenbank-Entwurf

4.3. Programmarchitektur

Da verschiedene Clients implementiert werden sollen, ist es sinnvoll das Projekt als Mehrschichtenarchitektur für verteilte Anwendung zu implementieren.
Der Server hält dabei die Funktionen zur Nutzung durch die Clients vor. Konkret greift der Server per OR-Mapper auf die Datenbank zu, bereitet die Daten in der Applikations-Schicht auf und reicht sie über eine Rest-Schnittstelle an die anfragenden Client weiter. Dabei muss gewährleistet sein, dass ein Nutzer nur die Daten abrufen darf, für die er autorisiert wurde.
Clientseitig werden Daten in einer Caching-Schicht zum Schutz vor eventuellen Verbindungsabbrüchen zwischengespeichert. Anschließend werden die erhaltenen Daten auf dem Endgerät für die Anzeige aufbereitet und angezeigt.
Abbildung 4.4 bildet diesen Aufbau grafisch ab:

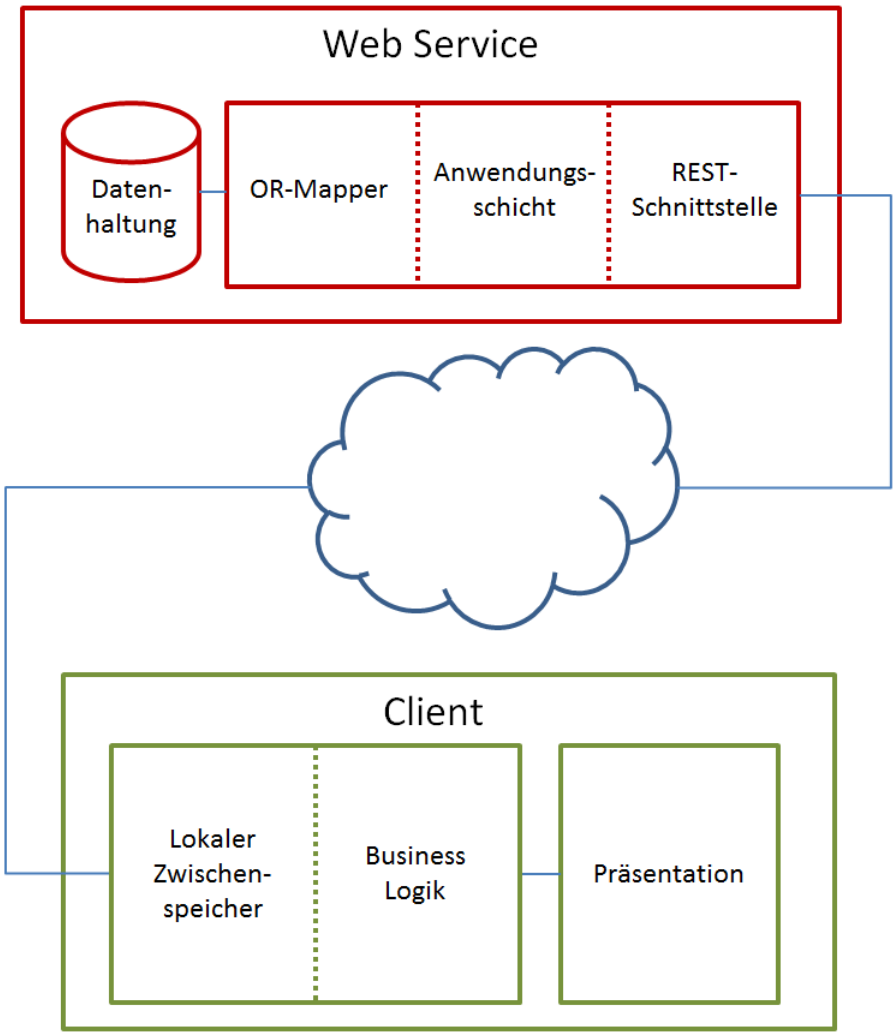

Abbildung 4.4.: Aufbau der Anwendung

4.4. Rollen-Konzept

Im Pflichtenheft wird für den zweiten Meilenstein eine Unterscheidung in den Berechtigungen gemacht. So dürfen beispielsweise nur Administrationen neue Übungen anlegen. Darum ist es nötig, ein Rollenkonzept zu entwickeln, der den Zugriff auf bestimmte Ressourcen reguliert.
Aus den Aussagen, die im Pflichtenheft getroffen wurden, geht hervor, das 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 Server authentifiziert. In diesem Status kann der Nutzer sich mit seinen Anmeldedaten einloggen oder als neuer Nutzer an der Web Applikation 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. Aspekte der Realisierung

In diesem Kapitel werden allgemeine Komponenten für die Umsetzung dieses Projektes beschrieben. Dabei werden jeweils nur diese Techniken vorgestellt, welche von mehreren Komponenten genutzt werden, sodass sie in den jeweils dafür vorgesehenen Kapiteln mehrfach genannt werden müssten.
Als generelle Aussage ist zu erwähnen, dass versucht wurde, möglichst viele Entwicklungswerkzeuge eines Unternehmens zu benutzen um mögliche positive Synergieeffekte in Form von leichter Kommunikation zwischen den gewählten Komponenten zu gewährleisten. Zur Umsetzung dieses Projekts wurden weitestgehend die Produkte der Softwarefirma *Microsoft* genutzt.

5.1. Entwicklungsumgebung

Für die Entwicklung sämtlicher Komponenten wurde *Microsoft Visual Studio 2015 Community Edition* (kurz *Visual Studio*) benutzt. Hierbei handelt es sich um die Standard-Entwicklungsplattform von *Microsoft*. Diese Entscheidung wurde aus Gründen getroffen:

- Die Clients sollen durch die Drittanbieter-Frameworks *Xamarin* und *AngularJs* umgesetzt werden. Beide Frameworks sind entweder bereits in die Entwicklungsumgebung integriert oder können leicht nachträglich zum Projekt hinzugefügt werden. Die hierfür benutzten Programmiersprachen *C#* und *Javascript* bzw. den kompletten **Web Stack!**[1] werden vollständig mit gängigen **IDE!**[2]-

[1] **Web Stack!**
[2] **IDE!**

Features wie Autovervollständigung, Syntax-Highlighting und Debugger unterstützt.

- Die Entwicklung von Web Anwendungen wird erheblich erleichtert, da *Visual Studio* mit einem integrierter Webserver ausgeliefert wird. Dadurch können entwickelte Applikationen direkt lokal getestet werden, ohne, dass ein zusätzlicher Webserver installiert oder die Anwendung auf einen Webserver deployt werden muss.
- Eine starke Integration von anderen *Microsoft* Produkten. Hierzu zählen die Hosting-Plattform *Microsoft Azure* und das Datenbank-System *Microsoft SQL Server*.
- Die Entwicklungsumgebung kann benötigte Komponenten und deren Abhängigkeiten durch den integrierten Paketmanager nachladen. Dadurch entfällt das nachträglichen Herunterladen von DLLs, wodurch Kompatibilitätsprobleme verringert werden.[3]

5.2. Datenbank-System

Als Datenbanksystem wurde ebenfalls die Lösung von *Microsoft* verwendet. Hierbei handelt es sich um *Microsoft SQL Server*. Durch die einheitliche Nutzung von Microsoft-Produkten kann für den Zugriff auf die Datenbank der OR-Mapper[4] genutzt werden. Dieser erlaubt es, direkt aus Modell-Klassen Datenbank-Entitäten zu entwickeln. Dieser Vorgang wird in Kapitel 6.2.1 näher erläutert[5].

5.3. Hosting-Plattform

Sowohl der Web Service als auch die Web-Application-Client müssen im Internet verfügbar gemacht werden, damit so von überall erreichbar sind. Hierzu biette *Microsoft* die Hosting-Plattform *Azure* an. Diese ermöglicht es, direkt aus *Visual Studio* heraus seine Webanwendung zu veröffentlichen. Gleichzeitig lässt sich eine Datenbank

[3] **?**, .
[4] objekt-relationaler Mapper
[5] **?**, .

hosten, welchen der Webservice direkt benutzen kann. Zusätzlich ist Azure sehr gut skalierbar, was den Einsatz als Hosting-Plattform für kleine Prototyp-Projekte optimal unterstützt[6].

[6] **?**, .

6. Realisierung der serverseitigen Implementierung

In diesem Kapitel wird näher auf die Implementierung des in Kapitel 4 besprochenen Webservices 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!**[1]-Architekturen eingegangen. Die hier beschriebene WebApi kann über die **URL!**[2] http://fit-bachelor.azurewebsites.net/ erreicht werden.

6.1. Was ist ein Webservice?

Um verteilte Systeme aufzubauen ist es nötig, eine Struktur zu implementieren, mit der Maschinen untereinander kommunizieren können. Diese Aufgabe übernehmen Webservices. Sie stellen innerhalb eines Netzwerkes Schnittstellen bereit, damit Maschinen plattformübergreifend Daten austauschen können. Hierbei wird meistens HTTP[3] als Träger-Protokoll genutzt, um eine einfache Interoperabilität zu gewährleisten.[4] Die dabei angeforderten Daten werden in der Regel im XML[5]- oder **JSON!**[6]-Format übermittelt.

[1] **RESTful!**
[2] **URL!**
[3] Hyper Text Transfer Protocol
[4] **?**, .
[5] Extensible Markup Language
[6] **JSON!**

6.1.1. RESTful Webservices

Da Webservices in der Regel HTTP als Protokoll verwenden, wurde die Idee zur Implementierung eines Webservices erweitert, um die Möglichkeiten des Protokolls besser zu benutzen. Heraus kam das Programmierparadigma REST (*Representational State Transfer*). Mit einem REST-Server bzw. einem RESTful Webservice bezeichnet man einen Webservices, welcher die strikte Nutzung von HTTP als Programmierparadigma umsetzt. Dies meint, dass sich, wie im Internet üblich, **URIs!**[7] zur eindeutigen Identifikation Ressource genutzt werden. Nachfolgend werden einige Prinzipien von REST näher beleuchtet.

Addressierbarkeit

Im Gegensatz zu anderen Webservice-Implementierungen stellen RESTful Webservices keine Methoden oder aufrufbare Funktionalitäten zu Verfügung, sondern ausschließlich Daten. 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 das Abhängigkeiten oder ein Kontext berücksichtigt werden müssen.
In den meisten Fällen, wie auch in den Anwendungsfällen dieser Arbeit, soll der Webservice CRUD[8]-Funktionalitäten bereitstellen. Damit die Schnittstelle nicht durch unnötig viele unterschiedliche URLs aufgebläht wird, sieht der RESTful-Ansatz die Verwendung der verschiedenen HTTP-Verben vor. Dazu werden zwei Arten von URLs unterschieden, um in Kombination mit HTTP-Verben verschiedene Aufgaben zu erfüllen. Zur Veranschaulichung sollen uns 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 einer Trainingsplans benutzt und mit einen konkreten Trainingsplan zu interagieren. Man spricht hier von

[7] **URIs!**
[8] Create Read Update Delete

einer *Element URI*.[9] Diese können mit verschiedenen HTTP-Verben kombiniert werden[10].

Nutzung von HTTP-Verben

Um den Rahmen der Arbeit nicht zu überspannen, wird sich hier nur auf die Vorstellung der vier meistverwendeten 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 wird zur Erstellung neuer Inhalte verwendet. Bei Nutzung von *Element URIs* wird versucht die ID für das neue Element zu benutzten. In der Regel wird das ID Management aber auf dem Server implementiert, so dass 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-conform wäre, eine Collection URI per PUT aufzurufen, wird dies selten implementiert, da der normale Anwendungsfall ist, dass ein einzelnes Objekt geändert werden soll. Stattdessen wird sich 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 hier 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 Erfolgsfall wird mit dem Statuscode *200 (Ok)* geantwortet und bei Fehlern mit *400 (Bad Request)* oder *404 (Not Found)*[11].

[9] **?**, .
[10] **?**, .
[11] **?**, .

Zustandslosigkeit

Da das statuslose Protokoll HTTP zum Datenaustausch genutzt wird, muss ein RESTful Webservice so implementiert werden, dass alle Informationen, welche für die Kommunikation benötigt werden, bei jeder Kommunikation mitgesendet werden. Was vordergründig als Nachteil erscheint ist ein wesentlicher Vorteil. Dadurch, dass jeder Request alle nötigen Informationen mitliefert, ist es nicht nötig, Kontext der Kommunikation über mehrere Request auf dem Server zu verwalten. Dadurch kann ein RESTful Webservice sehr leicht skaliert werden[12].

Daten sind unabhängig von der Präsentation

Das RESTful-Paradigma besagt, dass Daten losgelöst von einer Repräsentation bereit gestellt werden. Darum ist ein RESTful Webservice 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. Wird dieses nicht vom Server unterstützt, werden die angeforderten Daten in einem Standard-Format zurückgegeben.[13]

6.2. Aufbau der Komponenten

In diesem Kapitel wird beschrieben, wie der zuvor theoretisch beschriebene REST-Ansatz für das Projekt umgesetzt wurde.
Der Server besteht aus zwei Teilen: Der Datenbank und der WebApi, welche jeweils gesondert vorgestellt werden. Die WebApi wurde nach dem Design-Pattern **MVVM!**[14] aufgebaut. Hierbei werden die Objekte, welche aus Tupeln der Datenbank erstellt, aus präparierten Model-Klassen erzeugt. Bevor diese Daten dann über WebApi ausgespielt werden, werden sie vom Model in ein ViewModel übertragen. Hierbei wird, nach dem Grundgedanken des **Seperation of Concerns!**[15], klar zwischen den Models für die Datendank und den ViewModels, welche die WebApi benutzt, unterschieden werden.

[12] **?**, .
[13] **?**, .
[14] **MVVM!**
[15] **Seperation of Concerns!**

6.2.1. Aufbau der Datenbank

Da bei der Umsetzung des Projekts konsequent auf Produkte von Microsoft gesetzt wurde, wurde als Datenbanksystem **MS SQL!**[16] gewählt. Dies hat den den Vorteil, dass das **Microsoft Entity Framework!**[17], welches sehr gut in für die Nutzung mit einer WebApi 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 ((Richtiges Wort?!))in der Datenbank automatisch erzeugt wird.[18]
An den folgenden Beispielen wird exemplarisch beschrieben, wie die Model-Klassen aufgebaut wurden und wie sich daraus die Struktur der Datenbank ergibt. Grundlage für Model-Klassen ist das Interface *IEntity*(Quellcode 6.1):

```
1 [U+FFFD]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 [U+FFFD] // Definiert einen Trainingsplan
2    public class Schedule: IEntity<int>
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;
```

[16] **MS SQL!**
[17] **Microsoft Entity Framework!**
[18] **?**, .

```
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          // Fremdschluessel 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 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 Fremdschüssel!**[19]-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!**[20] 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.[21]

[19] **einwertige Fremdschüssel!**
[20] **mehrwertigen!**
[21] **?**, .

6.2.2. Aufbau der WebApi

Die Umsetzung der REST-Schnittstelle wurde mit Hilfe des Microsoft-Frameworks **ASP.NET Web API 2!**[22] realisiert. Dieses ermöglicht es, Controller-Methoden zu schreiben, welche über definierte Routen per HTTP aufgerufen werden können. Hierbei wurde wird die Umsetzung im Sinne des REST-Paradigmas durch vorhandene Funktionen unterstützt.[23]
Dies wird im Code-Beispiel 6.3 gezeigt:

```
1  [U+FFFD]// 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 occured")]
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       {
17          ModelState.AddModelError("UserId", "You can only create schedules for
                yourself");
18       }
19       if (!ModelState.IsValid)
20       {
21          return BadRequest(ModelState);
22       }
```

[22] **ASP.NET Web API 2!**
[23] **?**, .

```
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

Hierbei fällt sofort auf, dass das WebApi-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 WebApi-Framwork bietet hierfür pro Verb eine eigene Annotation. Im Codebeispiel 6.3 wird über die Annotation *HttpPost* (Zeile 12) ausgesagt, dass nur POST-Request durch diese Methode verarbeitet werden[24].
Das Framework versucht die empfangenen Daten in einem ViewModel-Objekt zu kapseln und anschließend zu validiert. Die dafür genutzten Validatoren werden direkt im View-Model als Annotationen angegeben.[25] Die Klasse *EntryModel* (Beispiel 6.4) zeigt die Möglichkeit in Zeile 7 und 11.
Schlägt die Validierung fehl, werden die Fehler mit dem passenden Statuscode zurückgegeben. Andernfalls werden die Daten per **Factory!**[26]-Klasse in ein Model konvertiert und per **Repository!**[27]-Klasse in der Datenbank persistiert. Anschließend wird dem ViewModel, im Sinne des REST-Gedankens, ein URL zur GET-Methode mit der ID des neu erstellten Objekts übergeben.

```
1  [U+FFFD]namespace fIT.WebApi.Models
2  {
3      // Defines one entry from the server
4      public class EntryModel<T>
5      {
```

[24] **?**, .
[25] **?**, .
[26] **Factory!**
[27] **Repository!**

```
6         // Id of an entity
7         [Required(ErrorMessageResourceName = "Error_Required",
              ErrorMessageResourceType = typeof(Resources))]
8         public T Id { get; set; }
9
10        // Name of an Entity
11        [Required(ErrorMessageResourceName = "Error_Required",
              ErrorMessageResourceType = typeof(Resources))]
12        public string Name { get; set; }
13
14        // Url to receive this entity
15        public string Url { get; set; }
16    }
17 }
```

Quelltext 6.4: Basis-Model-Klasse

Swagger

Da die WebApi parallel zu Clients entwickelt wurde, wurde schnell die Notwendigkeit einer Dokumentation des aktuellen Stands klar.
Aus diesem Grund wurde *Swagger* in die WebApi integriert. *Swagger* ist ein quelloffenes Framework zur Dokumentation von RESTful WebApis, welche von vielen großen Konzernen genutzt wird[28]. Durch Nutzung des **NuGet!**[29]-Packets *Swashbuckle* konnte durch hinzufügen von Kommentaren und Annotationen eine vollständige und übersichtliche Dokumentation erstellt werden[30].
Da das Authorisierungsprotokoll OAuth in Version 2 (kurz: **OAuth2**) zum Durchführungszeitpunkt des Projekts noch nicht von *Swagger* unterstützt wird, kann das Ausführen von API-Request 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.

[28] **?**, .
[29] **NuGet!**
[30] **?**, .

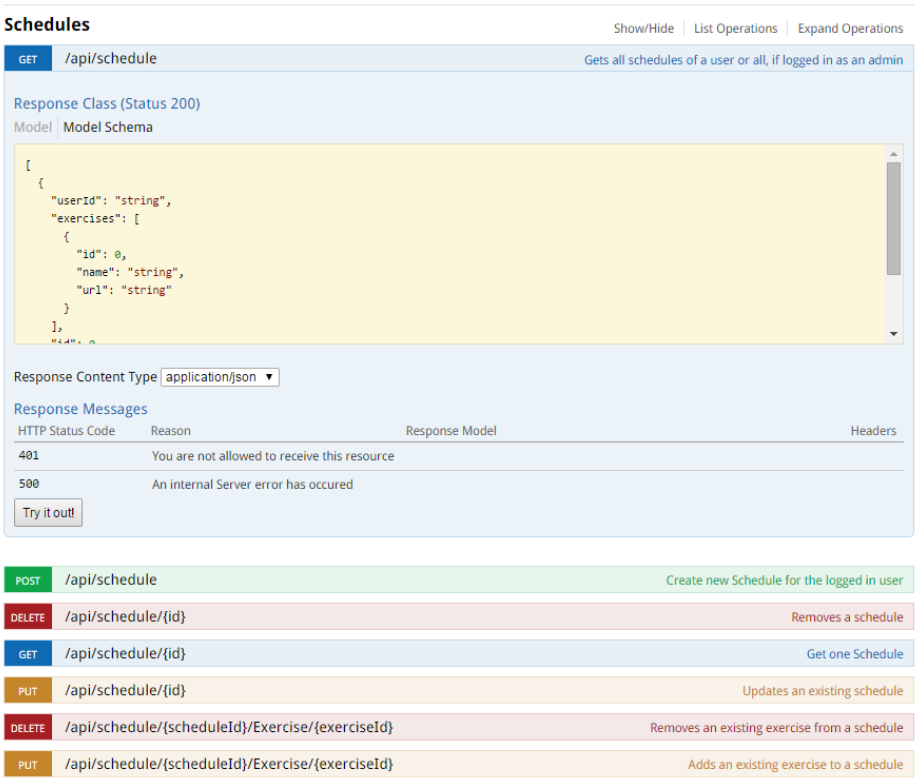

Abbildung 6.1.: Screenshot: Swagger UI der Web Api

6.3. Authentifizierung & Autorisierung

Wie bereits in Kapitel 4.4 beschrieben, darf nicht jeder Nutzer auf alle Daten zugreifen. Um dies zu bewerkstelligen, wurde ein Login-Mechanismus implementiert, welcher bekannte Nutzer authentifiziert. Da jedoch nicht alle authentifizierten Nutzer alle bereitgestellten WebApi-Methoden benutzen dürfen wurden auf Basis des **Role-Based Access Models!**[31] Rollen implementiert, welche den Nutzer zur Nutzung verschiedener Aufrufe autorisieren. Zur Umsetzung dieser Anforderungen wurde das Protokoll *OAuth2* implementiert.[32]

[31] **Role-Based Access Models!**
[32] **?**, .

6.3.1. OAuth2

OAuth2 ist ein Protokoll zur Authentifikation und zur Delegation von Zugriffsrollen. Die Struktur von OAuth2 kennt vier Instanzen, welche diesem Vorgang miteinander kommunizieren, nämlich Client, Resource Owner, Authorization Owner und Resource Server.[33]

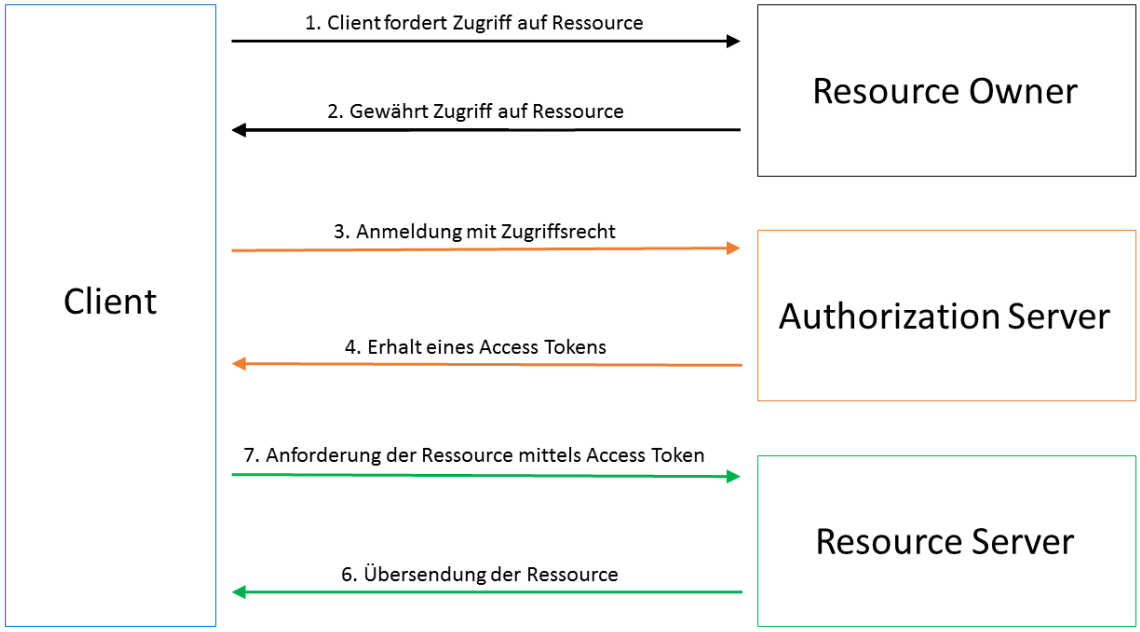

Abbildung 6.2.: Resourcenzugriff durch OAuth2

Client

Der *Client* ist ein Endpunkt, welcher eine Ressource (beispielsweise Trainingspläne) abrufen möchte. In unseren Fall ist das die Web- oder die native App. Diese kommunizieren jeweils mit den anderen Instanzen.

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

[33] **?**, .

einer Ressource.
Im diesem Projekt registriert sich der Nutzer an der WebApi. 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!**[34]) zur Anfrage am *Authorization Server*[35].

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[36]. Bei erfolgreicher Anmeldung erhält der Nutzer ein kurzlebiges *Access-Token*, dem 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 *Refreh-Token* wird benutzt, um ein neues *Access-Token* anzufordern. Die beiden Token-Arten werden nochmal genauer in Abschnitt 6.3.2 besprochen.[37]

Resource Server

Der *Resource Server* enthält die geforderten Ressourcen. Ab dieser Anfrage muss das Access-Token bei jeder Anfrage mitgesendet werden. Konkret passiert dies, indem im Header der Anfrage um den Schlüssel *authorization* erweitert wird.

Durch diese strikte Trennung dieser Instanzen ist es ohne weiteres möglich, dass unterschiedliche Systeme die jeweiligen Aufgaben übernehmen. Daraus hat sich in letzter Zeit etabliert, dass es immer häufiger **Single-Sign-On!**[38]-Szenarien implementiert werden. Dabei muss ich 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

[34] **Grant!**
[35] **?**, .
[36] **?**, .
[37] **?**, .
[38] **Single-Sign-On!**

registrieren muss, sondern jedes mal Zugriff über den Authorisierungsserver mithilfe seiner Credentials erhält.[39]

6.3.2. JWT and Bearer Token

Sowohl das Access-Token als auch das Refresh-Token sind JWTs (JSON Web Tokens). Das 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[40], welche jeweils als **base64!**[41]-String codiert wurden und mit einem Punkt getrennt sind. 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 Nutzername und Rollen, als auch Meta-Daten über das Token (z.B. das Ablauf-Datum).

- **Signatur**
  Damit gewährleistet ist, dass die Daten unverändert 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 Prototypen handelt, wurde die Implementierung der Client-Verwaltung nicht durchgeführt, da es für den Ablauf nicht zwingend benötigt wird. Der Server lässt alle gültigen Access-Token und alle bekannten Refresh-Tokens zu. Im produktiven Einsatz müsste diese Komponente dringend nachträglich implementiert werden, da sonst eine Sicherheitslücke entsteht.[42]

[39] **?**, .
[40] **?**, .
[41] **base64!**
[42] **?**, .

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 Request 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 von **HTTPS!**[43] erschwert.
Da das Refresh-Token nur 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[44]. Somit ist immer gewährleistet, das der Nutzer nur die Funktionalität nutzt, für die er auch autorisiert ist.

6.3.3. Zugriff per CORS

Im letzten Abschnitt wurden Maßnahmen beschrieben, damit nur autorisierte Nutzer an geschützte Daten herankommen. Mit CORS[45] wird ein weiterer Mechanismus vorgestellt, welcher den Zugriff auf die Daten per **Ajax!**[46] beschränkt.
Um den Nutzer davor zu schützen, dass eine Webseite im Hintergrund Daten von anderen Quellen nachläd, ist in jedem Browser eine **Same-Origin-Policy!**[47] implementiert. Diese besagt, dass nur Daten aus der gleichen Domäne, aus der der AJAX-Aufruf abgesetzt wurde, abgerufen werden dürfen.
Da es trotzdem häufig nötig ist, auf fremden Domains zuzugreifen, wurden schnell Workarounds wie das Vorgehensmodell **JSONP!**[48] eingeführt. Da diese jedoch von vielen Entwicklern als unelegant empfunden wurden[49], wurde mit CORS ein stan-

[43] **HTTPS!**
[44] **?**, .
[45] Cross-origin resource sharing
[46] **Ajax!**
[47] **Same-Origin-Policy!**
[48] **JSONP!**
[49] **?**, .

dardisierter Weg entwickelt, um Daten von fremden Domains abzurufen. Hierbei wird beim Server eine Liste an gültigen Domains für eine Cross-Domain-Anfrage hinterlegt.
Soll nun vom Browser eine Anfrage an den Server gesendet werden, wird über das HTTP Verb unterschieden, ob durch diese Anfrage eine Server-Datum 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 Daten-Änderung im Sinne von CORS durch den Aufruf angestoßen wurde oder wenn der Aufruf zusätzliche Schlüssel im Header enthält, wird vor der Ausführung ein **Preflight!**[50] gesendet. Dies ist eine OPTIONS-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[51]. Dadurch ist gewährleistet, dass nur berechtigte Clients anfragen an den Server senden. Es wurde auf weitere Implementierung von **Polyfills!**[52] verzichtet, da CORS bereits in allen modernen Browsern genutzt werden kann[53].

6.4. Testen der Funktionalität

Die erwartete Funktionsweise des Servers ist Grundvoraussetzung für die Umsetzung der Clients. Um diese zu bewerkstelligen wurde die *ManagementApi* entwickelt. Dies ist ein 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. Hierbei wurde darauf geachtet, dass alle Methode asynchron aufrufbar sind. Die so erzeugten Methoden iterative entwickelt und direkt getestet. Hierbei wurden zur Verifikation der Funktionalität ausschließlich Positivtests erstellt. So war sichergestellt, dass jede Methode der *ManagementApi* die gewünschten Änderungen auf dem Webservice

[50] **Preflight!**
[51] **?**, .
[52] **Polyfills!**
[53] **?**, .

durchführt.
Der nachfolgende Codeausschnitt zeigt beispielhaft die Entwicklung eines Testfalls:

```
1  [U+FFFD][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,
          PASSWORD).Result)
8     {
9        var data = session.Users.GetUserDataAsync().Result;
10       Assert.AreEqual(USERNAME, data.UserName);
11
12       try
13       {
14          data.UserName = NEWNAME;
15          session.Users.UpdateUserDataAsync(data).Wait();
16
17          data = session.Users.GetUserDataAsync().Result;
18          Assert.AreEqual(NEWNAME, data.UserName);
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 sowohl der ManagementService als auch die 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 der clientseitigen Implementierung 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!**[1] gegeben, die nun verfeinert wird. Dabei werden folgend die verwendeten Komponenten und Techniken erläutert und die Zusammenhänge zwischen den Techniken 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!**[2] handelt es sich um eine Art eines **monolithisch!**[3]en Multiuser-**Linux!**[4]-Systems.[5] Dieses Betriebssystem stellt die Hardwaretreiber zur Verfügung und führt die Prozessorganisation, sowie die Benutzer- und Speicherverwaltung durch. Jede Applikation wird in einem eigenen Prozess gestartet. In diesem Prozess befindet sich eine **Sandbox!**[6], die eine virtuelle Maschine mit der Applikation ausführt. Die Kommunikation aus der Sandbox heraus kann nur über Schnittstellen des Betriebssystems geschehen. Diese Einschränkung sorgt für Sicherheit im System, da ein Prinzip der minimalen Rechte

[1] **App!**
[2] **Android!**
[3] **monolithisch!**
[4] **Linux!**
[5] **?**, .
[6] **Sandbox!**

eingehalten wird. Demnach 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 allokierte Prozess beendet und das Betriebssystem bleibt von diesem Problem unberührt.[7] Android-Applikationen werden in der Programmiersprache **Java!**[8] geschrieben, mit einem Java-**Compiler!**[9] kompiliert und dann von einem Cross-Assembler für die entsprechende VM[10] aufbereitet. Das Produkt ist ein ausführbares .apk[11]-Paket.[12] Im Folgenden werden die Android-Komponenten, die für die Umsetzung relevant sind, genauer betrachtet.

7.1.1. User Interfaces

User Interfaces sind die Bildschirmseiten der 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 Checkboxen. Als zweites werden *View Groups* verwendet, um *Views* sowie andere *View Groups* anzuordnen. 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 *UIs* werden unter *res/layout* im Android-Betriebssystem hinterlegt. Des Weiteren können Ressourcen in den *UIs* verwendet werden. Unter Ressourcen versteht man Elemente, die zum Verzieren von Oberflächen verwendet werden können. Darunter fallen bespielsweise Grafiken oder *Style-Sheets*, die über den jeweiligen Ressourcen-Schlüssel aufgerufen und verwendet werden.[13]

[7] **?**, .
[8] **Java!**
[9] **Compiler!**
[10] Virtuelle Maschine
[11] Android Package
[12] **?**, .
[13] **?**, .

7.1.2. 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 eigentlichen Aufgaben liegen in der Bereitstellung eines Fensters, das dann auf den Screen, der für die App vom Betriebssystem bereitgestellt wird, gelegt wird. Das Fenster ist im Anschluss für die Annahme von Benutzerinteraktionen bereit. Das Fenster wird mit Hilfe des Aufrufs *SetContentView()* aufgerufen. Zur Benutzerinteraktion werden dann die bereits vorgestellten *View*-Elemente verwendet. Die *Activity* ist folgend für die Verarbeitung und Auswertung der Eingaben verantwortlich.
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 aus dem Menü des Betriebssystems zu setzen. Dabei ist empfehlenswert, dass dieselbe *Activity* sowohl das Main- als auch Launcher-Attribut erhält.
Diese Festlegungen müssen im Manifest hinterlegt werden. Das Manifest liegt im Root-Ordner der App und stellt dem Betriebssystem wichtige Informationen der Applikation zur Verfügung. Dieses Manifest wird vor Ausführung der App analysiert und ausgewertet. Darin kann beispielweise festgelegt werden, welche Komponenten oder anderen Applikationen auf entsprechende *Activities* zugreifen dürfen. Wenn eine *Activity* nicht von außerhalb der App erreicht werden soll, sollte kein Intent-Filter gesetzt werden, da demnach der genaue Name der *Activity* zum Start bekannt sein muss. Diese Informationen sind jedoch nur in der gegenwärtigen App vorhanden.
Da eine App normalerweise aus mehreren *Activities* besteht, müssen diese *Activities* 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. Ein Intent ist ein Nachrichtenobjekt innerhalb von Android, welches zur Kommunikation zwischen App-Komponenten verwendet wird. In diesem Fall zwischen zwei *Activities*. Zur Erstellung 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 Betriebssystem ausführen zu lassen. Beispielsweise kann man ein *Intent* mit der Aktion

zum Starten des Email-Programms übergeben und die entsprechend im Betriebssystem hinterlegte Applikation zum schreiben von Emails wird geöffnet.
Eine *Activity* kann drei Stati in einem *Lifecycle* einnehmen. Zum einen kann die *Activity* im Status *Resumed* - oft auch *Running* genannt - sein und damit momentan im User-Fokus stehen, also im Vordergrund der App sein und die Interaktionen entgegennehmen. Des Weiteren kann eine *Activity* pausieren, wenn eine andere im User-Fokus steht. Dabei ist der *View* der betrachteten *Activity* jedoch immer noch teilweise sichtbar, da der darüberliegende *View* zu Beispiel nicht den gesamten Bildschirm in Anspruch nimmt. Anders verhält es sich, wenn der *View* der betrachteten *Activity* komplett überdeckt ist. Dann befindet sich die *Activity* nämlich im Status *Stopped*. Sowohl im Status *Stopped* als auch im Status *Paused* lebt die *Activity* noch. Das bedeutet, dass das *Activity*-Objekt zusammen mit allen Objekt-Stati und Memberinformationen im Arbeitsspeicher liegt. Der einzige Unterschied dieser beiden Stati liegt darin, dass eine *Activity* im Status *Paused* noch eine Verbindung zum *Window-Manager* besitzt, die im Status *Stopped* nicht mehr vorhanden ist. Gemeinsam haben diese beiden Stati jedoch noch, dass sie bei mangelndem Arbeitsspeicher vom Betriebssystem zerstört werden können.

Die Ausführung der internen Methoden einer *Activity* ist abhängig von den Eingaben des Benutzers. Dabei durchläuft jede *Activity* ihren *Lifecycle*, der in Abbildung 7.1 dargestellt ist. Darin ist zu erkennen, dass zuerst die *OnCreate()*-Methode aufgerufen wird. Darin werden alle essentiellen Initialisierungen gemacht und der *View* aufgerufen. Nachfolgend werden *OnStart()* und *OnResume()* durchlaufen bis die *Activity* den User-Fokus wieder verliert, jedoch der *View* noch sichtbar ist. In dem Moment wird die *OnPause()*-Methode ausgeführt, um Benutzereingaben gegebenenfalls speichern zu können, denn in diesem Zustand ist es in seltenen Fällen möglich, dass der Status - wie oben erklärt - durch das Betriebssystem zerstört wird. Kehrt der Benutzer zurück, wird *OnResume()* wieder aufgerufen, sonst *OnStop()*, um auch dort aufgenommene Daten persitieren zu können. Von dort gibt es zwei verschiedene Rücksprung-Möglichkeiten. Zum einen könnte der Fall eintreten, dass die Daten der *Activity* aus dem Arbeitsspeicher gelöscht wurden, die *Activity* jedoch noch einmal aufgerufen wird. In diesem Fall startet die *Activity* wieder von vorn. Eine weitere Möglichkeit ist die Rückkehr des Benutzers zu der *Activity*. Dabei werden dann die Methoden *OnRestart()* und *OnStart()* aufgerufen.
Zusammenfassend lässt sich daraus ableiten, dass die Persistierung von Eingaben in den Methoden *OnPause()*, *OnStop()* und *OnDestroy()* durchgeführt werden soll-

ten, da diese Zustände zerstört werden können. Die weiteren Methoden sollten aus Performancegründen jedoch minimal und agil gehalten werden.

7.1.3. 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 keinen zugehörigen *View*, sondern werden von anderen App-Komponenten, wie beispielsweise einer *Activity* gestartet. Sie laufen im *Main-Thread* des Prozesses der aufrufenden Komponente. Ein *Service* erstellt keinen eigenen *Thread*, noch einen eigenen Prozess zur Abarbeitung der Aufgaben. Diese Eigenschaft der *Services* muss vom Entwickler bedacht werden. Denn daraus kann man ableiten, dass rechenintensive Aufgaben in einem explizit gestarteten *Thread* arbeiten sollten, um Fehler der Art *Application Not Rsponding* (ANR) zu vermeiden und die Benutzeroberfläche nicht unnötig zu verlangsamen. Ein Vorteil besteht jedoch darin, dass *Services* Aufgaben auch dann noch ausführen können, 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.
Bei Android wird grundsätzlich zwischen zewi 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 selbstständig. Auf der anderen Seite gibt es *Bound-Services*, die durch einen Aufruf von *BindServcice()* 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 *IBind* und sorgt für den Austausch von *Request* und *Results*. Des Weiteren verläuft eine mögliche Interprozess-Kommunikation zwischen Komponente und *Service* über dieses *Interface*. Die größte 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. Natürlich gibt es Mischformen dieser beiden *Service*-Arten, die abhängig von der zu leistenden Aufgabe gewählt werden sollten.

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.
Started-Services werden die Methode *OnCreate()* nach dem Start durch eine Komponente ausführen, wenn der *Service* noch nicht läuft. Darin sollten dann die Initialisierungen und einmaligen Aufgaben zum Start des *Services* durchgeführt werden. *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 *Devices* oder das Beenden durch eine Komponente oder den *Service* selbst, wird *OnDestroy()* ausgeführt, um abschließende Aufgaben durchzuführen. Dazu zählen beispielsweise das Beenden von Datenbankverbindungen oder *Threads*.
Bound-Services werden, wie oben genannt, über *BindService()* von einer Komponente gestartet und führen dann, genauso wie die *Started-Services*, die *OnCreate()*-Methode zum Initialisieren aus. Gefolgt vom aktiven Status, in dem anfangs *OnBind()* aufgerufen wird und die von den Komponenten verlangten Aufgaben ausgeführt werden. Anschließend lösen sich die Komponenten wieder vom *Service*. Haben sich alle Komponenten gelöst, wird auch beim *Bound-Service OnDestroy()* ausgeführt.

7.1.4. 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 7.1.3), und die App wird vom Benutzer erneut gestartet, so wird diese Komponente in dem noch bestehenden Prozess und *Thread* eingepflegt.
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 ***user ID!***[14] besitzen.
Prozesse können aber auch durch das Betriebssystem zerstört werden, wenn die Geräte-Ressourcen sich zum Beispiel dem Ende neigen 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!***[15] beim Starten einer App, ist der Hauptakteur für die Kommunikation mit dem Betriebssystem. So werden alle Aufrufe an die Komponenten des *Android UI toolkits* 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 lästige Wartezeiten zu ermöglichen. Einzige Einschränkung dabei ist, dass niemals von einem anderen *Thread* als dem *UI-Thread* auf ***Android UI toolkits!***[16] zugegriffen werden darf. Diese Limitierung muss bei der Implementierung beachtet werden.
Zum Umgehen dieser Problematik können asynchrone ***Tasks!***[17] verwendet werden, die Aufgaben außerhalb des *UI-Threads* ausführen. Auf das Ergebnis dieser Ausführungen kann dann wieder zugegriffen werden. Diese Umsetzung bietet einen leichteren Umgang mit *Multithreading* für den Entwickler und genießt deshalb immer größere Beliebtheit.

[14] **user ID!**
[15] **UI-Thread!**
[16] **Android UI toolkits!**
[17] **Tasks!**

7.1.5. SQLite

SQLite ist eine in sich geschlossene und serverlose **SQL!**[18]-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!***[19] 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 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.

7.2. Was ist Xamarin Platform?

Xamarin Platform 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!**[20]. Mono seinerseits hat mehrere Vorteile:

- **Popularität**
  Es kann auf die Erfahrung von Millionen C# -Entwicklern zurückgegriffen werden.

- **Höhere Programmiersprache**
  Es können die Vorteile von höheren Programmiersprachen verwendet werden. Zu nennen sind dabei besonders *Threading*, automatische Speicherverwaltung und ***Reflection!***[21].

- **Klassenbibliotheken**
  Die Verwendung von bestehenden Klassenbibliotheken erleichtern das Umsetzen komplexer Aufgaben.

[18] **SQL!**
[19] **Trigger!**
[20] **Mono!**
[21] **Reflection!**

- ***Cross-Platform***
  Die fertiggestellte Software kann auf fast allen Systemen verwendet werden.

Damit können plattfomunabhängige Programme in C# programmiert werden. Beliebtheit erlangte Mono mit dem Wunsch von Entwicklern Apps auf verschiedenen mobilen Betriebssystemen bereitstellen zu können und dabei Änderungen und die Entwicklung größtmöglich zu vereinen. Genau diese Wünsche werden mit *Xamarin Platform* erfüllt. Vorher war es immer nötig drei Applikationen für die verbreitetsten mobilen Betriebssysteme zu entwickeln. Dies bedeutete, dass die *Guidlines* der jeweiligen Systeme iOS, Android und Windows Phone analysiert und in den jeweiligen Programmiersprachen umgesetzt werden mussten.

7.2.1. Multiplattform-Unterstützung

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.[22] So ist der Umgang mit asynchronen Funktionen in dieser Sprache zum heutigen Stand am besten gelöst. Des Weiteren können *Shared Projects* zur Entwicklung verwendet werden, sowie PCL[23]s und ***NuGet!***-Pakete eingebunden werden, um den Funktionsumfang schnell und einfach erweitern zu können.[24]

7.2.2. 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 freien und starken Entwicklungsumgebung Visual Studio in Kombination mit C#, zum anderen kann man

[22] **?**, .
[23] Portable Class Library
[24] **?**, .

alle Besonderheiten der Android-Entwicklung einbeziehen und verwenden.
So muss man am Anfang der Entwicklung auswählen, welche Android-API[25] als Minimalvoraussetzung verwendet werden soll und welche Version vorrangig unterstützt werden soll. Zweifellos ist es möglich die von dem *Device* 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 sehr große 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 gute Beihilfe.

7.2.3. 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- und *Target*-Anforderung gewählt. Dies steht für Android 4.4 mit dem Namen Kitkat. Damit sollte eine große Abdeckung von Android-Geräten bewerkstelligt werden.[26]

7.3. Eigene Umsetzung

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 gesetzt und die Entwicklung darauf abgestimmt.

[25] Application Programming Interface
[26] **?**, .

7.3.1. Anlegen der Layouts

Der Aufbau der Layout-Seiten ist auf Grund einer vorher abgestimmten *User-Guideline* vorgenommen worden. So entsteht eine Übersichtsseite, auf der der Benutzer entscheiden kann, ob er sich einloggt oder registriert. Zu Beginn des Anlegens dieses *Main-Layouts* wurde ein *ViewGroup* des Typs *LinearLayout* gewählt, um die *Widget*-Elemente darauf anzuordnen. Hinzugefügt wurden demnach zwei *Buttons* für die genannten Funktionen, ein Text für die Überschrift, sowie eine *ImageView*, die für die Anzeige der Konnektivität zum Server verwendet wird.
Bei einem *Click* auf den Registrieren-*Button* erscheint ein Dialog im Vordergrund, der den Benutzer dazu auffordert die benötigten Daten zur Registrierung einzugeben. Die Oberfläche orientiert sich demnach an einem *RelativeLayout*, um die *Widgets* darauf anzulegen. Zur Vereinfachung der Validierung wird für das Eingabefeld der Email-Adresse ein *EditText-Widget* mit dem *InputType textEmailAddress* verwendet. Somit wird die Benutzereingabe in diesem Feld vom Betriebssystem auf die Eigenschaften einer Email-Adresse überprüft. Intern wird dazu eine *Regular Expression* zur Validierung verwendet und die Überprüfung kann vernachlässigt werden.
Zur Weitergabe der Eingabedaten muss eine von *EventArgs* erbende Klasse erstellt werden, die alle Daten zur Registrierung hält. Die Eingabe des Passwortes und der Wiederholung des Passwortes wird in einem speziellen Passwort-Feld entgegengenommen. Dadurch wird die für Passwörter bekannte Eingabe vom Betriebssystem verwendet. Die Buchstaben werden als Punkte dargestellt und sind somit nicht so leicht für Dritte bei der Eingabe einsehbar.
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 parst. 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. Nach der erfolgreichen Registrierung verschwindet der

Dialog wieder 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.
Beim Login besteht das Dialogfenster aus den beiden Eingabefeldern für Benutzername und Passwort, sowie einem Login-Knopf. Beim Betätigen des Knopfes wird die Kombination *Username* und Passwort beim *OnOffServiceLocal* abgefragt. 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 ist erforderlich. Aus sicherheitstechnischen Gründen wird nicht angegeben, ob der Benutzername schon nicht in der Datenbank vorhanden ist, oder, ob nur die Kombination fehlerhaft ist. Nach einer validen Eingabe verschwindet der Login-Dialog wieder und die Startseite ist für drei Sekunden sichtbar, um einen Blick auf die Konnektivitätsanzeige werfen zu können. Danach folgt eine Weiterleitung zu den Trainingsplänen des nun eingeloggten Benutzers.
Die Darstellung der Trainingspläne und der zugehörigen Übungen auf einer folgenden Seite sind technologisch gesehen gleich. Einzig die Beziehung der zu ladenden Daten und die übergebenen Werte ändern sich. Als herausfordernd hat sich 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 herauszufinden. Aber zuerst wurde die Erstellung der *ListView* gelöst. 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.
Zur Übergabe der *UserId* und der *ScheduleId* an die *ExerciseActivity* wurde zuerst vergeblich versucht diese im Aufruf der *ExerciseActivity* mit zu übergeben. Diese Möglichkeit hat sich im Nachhinein als Irrtum herausgestellt und eine weitere Einarbeitung in die vorherig erläuterten *Intents* (siehe Kapitel 7.1.2) durchgeführt. Danach setzte sich die Umsetzung dahingehend weiter, dass die Funktion *PutExtra()*

des *Intents* dazu genutzt wurde, um die 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.

```
1  /// <summary>
2  /// Clickevent 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 =
           exercises[e.Position].Description.ToString();
12
13     var practiceActivity = new Intent(this, typeof(PracticeActivity));
14     practiceActivity.PutExtra("Exercise", exerciseId);
15     practiceActivity.PutExtra("Schedule", scheduleId);
16     practiceActivity.PutExtra("User", userId);
17     StartActivity(practiceActivity);
18 }
```

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 werden und kann nicht einfach aus der aktuellen *UserSession* des Benutzers gelesen werden, da man davon ausgehen muss, dass sich der Benutzer auch offline hätte einloggen können. Demnach hat man im Online-Modus zwei Möglichkeiten die Id des *Users* 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 =
               Java.Lang.Integer.ParseInt(txtNumberOfRepetitions.Text);
11
12         bool result = await ooService.createPracticeAsync(scheduleId,
               exerciseId, userId, DateTime.Now, weight, repetitions,
               numberOfRepetitions);
13         if(result)
14         {
15             //Zurueck zu der Uebungsseite
16             OnBackPressed();
17         }
18         else
19         {
20             new AlertDialog.Builder(this)
21                 .SetMessage("Anlegen ist schiefgegangen")
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 ersichtlich kann man die übergebenen Informationen zwischen *Activities* aus dem *Intent* auslesen. Zudem kann man erkennen, dass man dank Xa-

marin das Parsen einer *Integer*-Zahl über eine *Java*-Funktion durchführen kann.
Im folgenden Kapitel wird dann auf den noch unbekannten Aufruf des *ooService*-Objekts eingegangen.

7.3.2. OnOffService

Dieser *OnOffService* 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 *Device* gerade online oder offline ist und dementsprechend die Interaktion mit der lokalen Datenbank (siehe Kapitel 7.3.3) oder dem Server durchgeführt. Zudem wird 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 nachgucken
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*

In diesem Codebeispiel kann man die Umsetzung dieser Aufgabe an Hand 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 alle in diesem Schritt behandelt, um die aufrufende *Activity* mit der originalen Fehlermeldung des Servers versorgen zu können 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. Es würde die Performancevorteile verspielen, wenn man diese asynchronen Methoden dann beim Serverabruf synchron verwenden würde. Auf der anderen Seite sollten dadurch die Performancevorteile der Asynchronität in diese App übernommen werden. 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 großteils Rückgabewerte liefern, die für die Weiterverarbeitung essentiell

sind. Möglich wäre diese 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.

7.3.3. Lokale Datenbank

Technologisch wird eine SQLite-Datenbank aus den bereits in Kapitel 7.1.5 erläuterten Vorteilen genutzt.
Die lokale Datenbank dieser App wird für die Umsetzung des *Caches* (siehe Kapitel 7.3.6) benötigt. Darin werden die lokalen Daten gespeichert und mit dem Server abgeglichen. Die Erstellung der Datenbank findet beim Start des *OnOffServices* statt. Ist die Datenbank schon vorhanden, wird keine weitere Aktion ausgeführt. Zur Verbesserung der Leistung wird die Erstellung in einem separaten Thread durchgeführt, da kein Rückgabewert erwartet 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 schenken und die Anzahl der direkten SQL-Befehle minimieren würde. Diese wurde testweise eingepflegt, funktionierte aber nicht erwartungsgemäß.

```
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},
                Password={3}]", LocalId, UserId, Username, Password);
13      }
14  }
```

Quelltext 7.4: *UserModel* für die lokale Datenbank

Der lokale *User* 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 diese Überlegung 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. Einfache Such- oder Einfüge-Operationen werden vom *Framework* bereitgestellt, wohingegen Abfragen über die erstellten Beziehungstabellen selbst umgesetzt wurden.

7.3.4. Lokaler ManagementService

Die Verbindung zum Server geschieht über das bereitgestellte *Package* fIT.WebApi.Client.Porta
Alle benötigten Funktionen werden darin bereitgestellt. Darüber 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 zum Server den *ManagementLocalService*. Dieser *Service* arbeitet mit dem Server direkt zusammen und ist ausschließlich für das Abrufen von Daten vom Server verantwortlich. Rückgabewerte werden meist einfach weitergereicht.
Zur Veranschaulichung ein kleiner 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){[...]}
15     return result;
16 }
```

Quelltext 7.5: Login am Server

7.3.5. Verbindungsprüfung zum Server

Die Verbindungsprüfung zum Server geschieht im *OnOffService*. Dafür wird eine unendliche Schleife in einem eigenen *Thread* gestartet, um in einem festen Intervall (alle 10 Sekunden) einen *Ping* zum Server zu schicken und damit die Erreichbarkeit des Servers zu überprüfen. 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              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 Pings 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 unwahrscheinlich eingestuft wurde, wurde diese Umsetzung niedriger priorisiert.

7.3.6. Umsetzung des Caches

Der *Cache* 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 *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 geupdatet.
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. 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 *Username* 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.

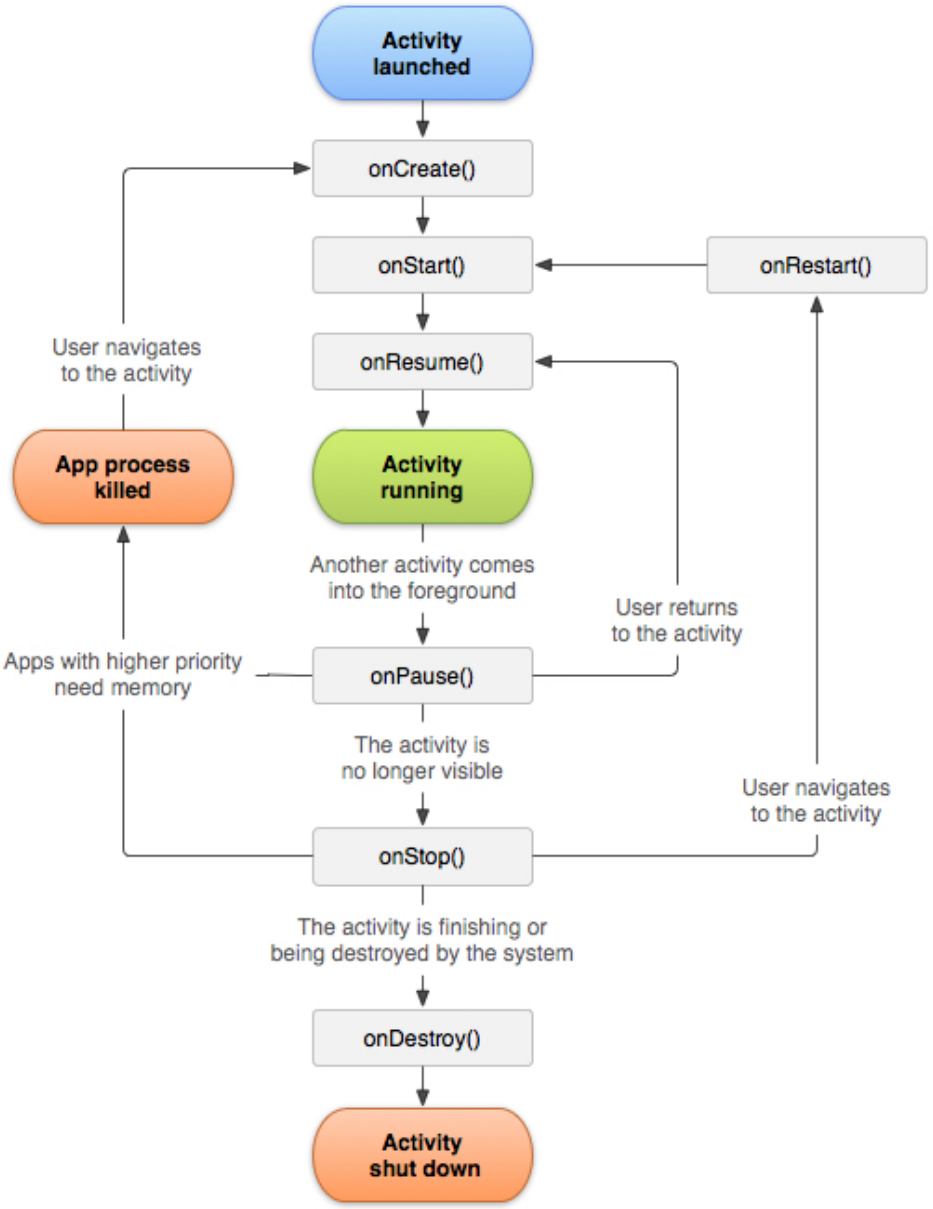

Abbildung 7.1.: Android Activity-Lifecycle

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

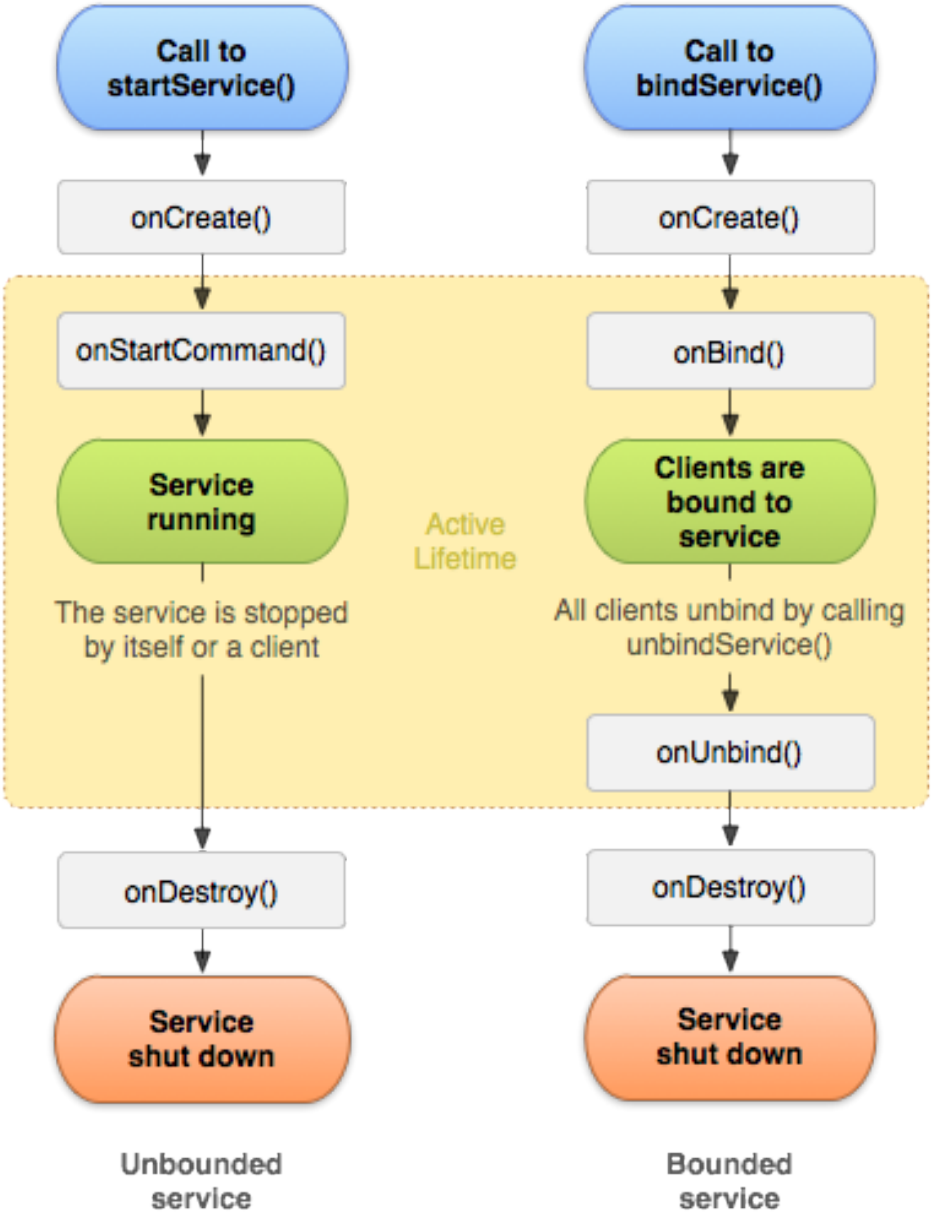

Abbildung 7.2.: Android Service-Lifecycle

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

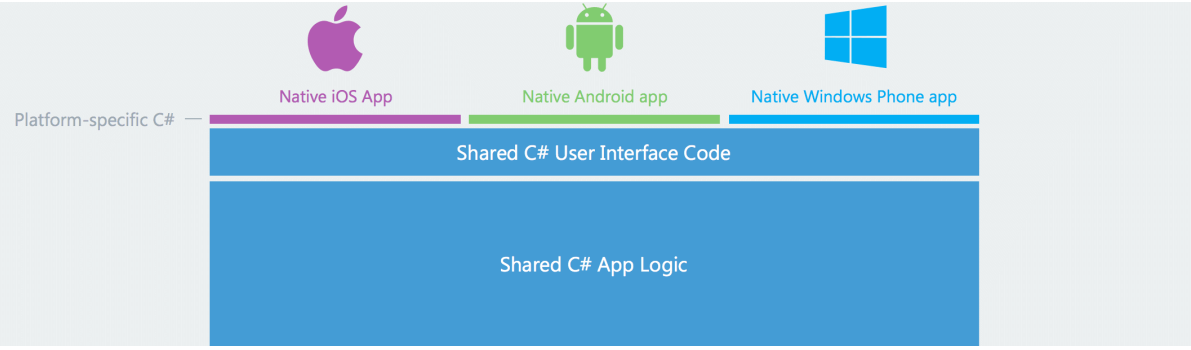

Abbildung 7.3.: Xamarin Platform
Quelle: http://xamarin.com/platform

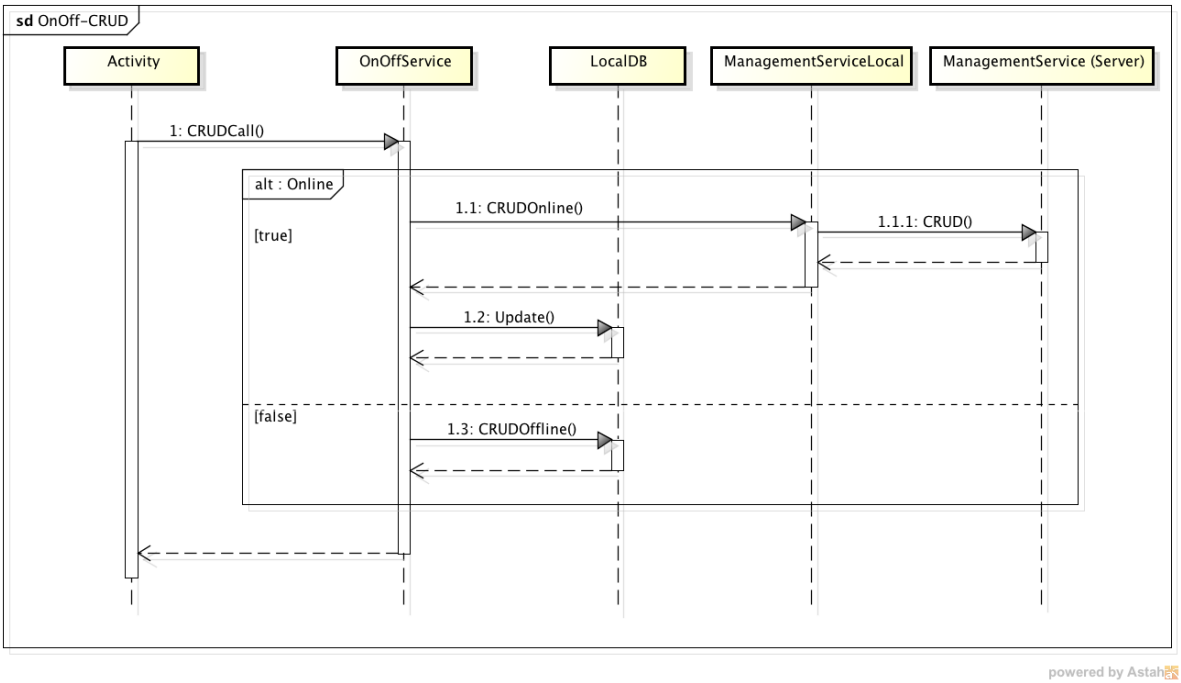

Abbildung 7.4.: Sequenzdiagramm CRUD

8. Realisierung der clientseitigen Implementierung als Webapplikation

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

8.1. Definition einer Single Page Application

Die Web-Applikation wurde als **Single Page-Application!**[1] (SPA) implementiert.
Bei diese Art der Webanwendung wird die Verarbeitung von Anfragen vom Server auf die Client 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 HTML-Struktur übernommen, wobei der Grad der Autonomie des Clients vom genutzten Framework abhängt. Diese Auslagerung der Verarbeitung begünstigt die Nutzung des RESTful Webservices (siehe Kapitel 6), welcher die Daten liefert, die die SPA dann verarbeiten kann. Daraus ergibt sich, dass die Verbindung zum Server nur lose vorhanden ist. Es müssen nur die wenigen statischen Seiten und deren eingebundenen Skripten vom Server abgerufen werden. Die sonstige Kommunikation besteht nur noch zwischen der SPA und dem Webservice. Im Laufe diese

[1] **Single Page-Application!**

Kapitels wird noch darauf eingegangen, wie auch diese beiden Verbindungen durch geeignete Techniken bis auf ein Mindestmaß reduziert wurden.[2]

8.2. AngularJs

Zur Umsetzung der SPA wurde das quelloffene Framework AngularJs from Google gewählt, welches sich immer größerer Popularität erfreut[3][4]. Es bietet alle Möglichkeiten, um einen Endgerät als weitestgehend autonomen **Fat-Client!**[5] zu implementieren. Dadurch wird die Möglichkeit geschaffen, eine Applikation zu entwickelte, welche selbst dann akzeptabel reagiert, wenn ein aufrufendes Endgerät keine Verbindung zum Internet besitzt. Hierzu stellt es zusätzliches **MarkUp!**[6] bereit, welches zu Laufzeit interpretiert und ausgeführt wird. Die dazu nötigen Komponenten von AngularJs und deren Einsatz werden nachfolgend genauer erläutert.

8.2.1. Begriff: Komponente

Wenn in diesem Kapitel der Begriff Komponente benutzt 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[7].

8.2.2. Dependency Injection

AngularJs wurde so konstruiert, dass zu jeden Zeitpunkt eine gute Testbarkeit gewährleistet ist. Aus diesem Grund setzt AngularJs für die Verbindung von verschiedenen Funktionen auf *Dependency Injection*.
Hierbei wird beim Ausführen einer Funktion die genutzten Parametern geprüft. Gibt

[2] **?**, .
[3] **?**, .
[4] **?**, .
[5] **Fat-Client!**
[6] **MarkUp!**
[7] **?**, .

es eine bekannte Komponente, welche den gleichen Namen, wie der geforderte Parameter besitzt, erzeugt AngularJs ein Objekt diese Komponente und übergibt dieses an die Funktion. Hierbei wird bei der Erzeugung der Objekte für die Parameter ebenfalls Dependency Injection angewandt.[8]

8.2.3. Services

Services sind abgeschlossene Komponenten, welche bestimmte Funktionalitäten kapselt und bereitstellen. AngularJs bietet neben der Möglichkeit, eigene Services zu erstellen eine große Auswahl an vorhandenen Services, um verschiedene wiederkehrende Aufgaben durchzuführen. Dabei sind die Services von AngularJs immer mit dem Präfix *$* versehen. Einer der wichtigsten Serviceses 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!**[9]-Muster). Fordert eine weitere Komponente ein Objekt des Services, wird das bereits erstellt Objekt zurückgegeben.[10]

8.2.4. Promises

Im letzten Abschnitt wurde der Service *$http* angesprochen (siehe 8.2.3), welches Methoden zur Kommunikation mit einem RESTful Webservice 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[11]. Deshalb wurde das Prinzip der *Promises* eingeführt. Dies erlaubt die Abarbeitung von asynchronem Code, indem beim Aufrufen einer asynchron abzuarbeitenden Methode ein *Promise*-Objekt zurückgegeben wird. Die repräsentiert ein Versprechen über einen späteres Ergebnis. Auf dieses Ergebnis kann mit den Methoden *then* und *catch* reagiert werden. *Then* wird nach erfolgreicher Abarbeitung der asynchronen

[8] **?**, .
[9] **Singleton!**
[10] **?**, .
[11] **?**, .

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 verketten der Aufrufe möglich. Die Nutzung von Promisses ist im *Module $q* gekapselt, welches sich an dem Projekt *q* orientiert.[12][13][14].

8.2.5. MVC

AngularJS nutzt das Architektur-Muster **MVC!**[15] um Datenbeschaffung bzw. -haltung, Datenverarbeitung und Datenpräsentation strikt zu trennen.
Zur Beschaffung werden entweder Services, welche per Dependency Injection hinzugeladen werden, benutzt oder Model-Funktionen 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 bereit stellt[16].

```
1  [U+FFFD]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;
10   $scope.authentication = authFactory.authentication;
11 };
```

Quelltext 8.1: Controller für die Navigationsleiste

[12] **?**, .
[13] **?**, .
[14] **?**, .
[15] **MVC!**
[16] **?**, .

Hierbei ist zu sehen, wie mittels *Dependency Injection* die benötigten Komponenten für die Funktion bereitgestellt werden (Zeile 1). Interessant ist dabei besonders die Komponente *$scope*. Diese wird verwendet, um Daten zwischen dem Controller und der View (in dem Fall der statischen HTML-Seite) auszutauschen (Zeile 8ff.)[17]. Dabei stellt AngularJs Funktionen bereit, um diesen Austausch bidirektional durchzuführen. Somit können beispielsweise Daten, welche ein Nutzer in ein Textfeld eingibt, im Controller weiter verarbeitet werden.
Die Benutzung der durch den *$scope* bereitgestellten Variablen wird im Beispiel 8.2 gezeigt.

```
1  [U+FFFD]<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.isAuth">
17           <a href="#/login">Log in</a>
18         </li>
19         <li data-ng-show="!authentication.isAuth">
20           <a href="#/register">Register</a>
21         </li>
22         <li data-ng-show="authentication.isAuth">
23           <a href="#">Hallo {{authentication.userName}}</a>
24         </li>
25         <li data-ng-show="authentication.isAuth">
26           <a href="#/schedules">Schedules</a>
```

[17] **?**, .

```
27          </li>
28          <li data-ng-show="authentication.isAuth">
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 Attribute *data-ng-controller* (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 benutzten:

- **Nutzung in Direktiven**
  AngularJs stellt Direktiven bereit, welche die Darstellung der Webseite beeinflussen. Dies zeigt sich in Zeile 6. Das Direktiv *ng-hide* wird benutzt, um dynamisch HTML-Element auszublenden. Um zu entscheiden, 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[18].

- **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 geschweifte Klammern umgeben sein.

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

[18] **?**, .

8.2.6. Routing

Damit das MarkUp nicht schnell durch die Nutzung von zusätzliche Direktiven überladen wird, bietet AngularJs das module *$route* zur Implementierug von Routing an. Hierbei wird durch die Direktive *ng-view* ein Block als View-Container definiert. Dieser wird abhängig von der aufgerufenen URL mit unterschiedlichen Inhalten befüllt. Das Beispiel 8.3 zeigt solch eine Routing-Konfiguration.

```
1  [U+FFFD]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, innerhalb des View-Containers gerendert werden. Diese MarkUp-Schnipsel liegen ebenfalls auf dem Server.

8.3. Umsetzung

Durch die Nutzung der vorgestellten Komponenten war es möglich, einen Prototyp, welcher die grundlegenden Anforderungen, die in Kapitel 4 definiert wurden, erfüllt.[19] 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 bewehrte, quelloffene 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. Dies liegt daran, dass *Bootstrap* es erlaubt, unter Nutzung eines integrierten Grid-Systems eine hoch-responsive Applikation zu erstellen.[20]
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. Herausforderung statusloses Protokol Http

Da HTTP ein statusloses Protokol ist, ist es ohne Weiteres nicht möglich, ein einmal abgerufenes Access-Token wiederzuverwenden. Um dies dennoch zu erreichen, wurde Service *authFactory* entwickelt, welcher sich um das ein- und ausloggen kümmert. 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 benutzt, nämlich der *Interceptor*. Dies ist ein spezieller Service, welcher eng mit dem *$http*-Service verbunden ist. Mit dem Interceptor ist es möglich, eine Request kurz vor- und eine Response direkt nach Erhalt einzusehen und gegebenenfalls darauf zu reagieren. Dies wird verwendet, um vor

[19] **?**, .
[20] **?**, .

dem Absenden eines Request den *authorizsation*-Header zu setzen, falls ein Access-Token vorhanden ist.
Der Respond 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[21].

8.3.3. Online-Check

Der Nutzer soll eine visuelle Rückmeldung darüber bekommen, ob die Applikation gerade eine Verbindung zu Web Service aufbauen kann oder nicht. Dafür wurde der im Beispiel 8.1 gezeigte Controller um einen Online-Check erweitert. Ruft die Anwendungen in einem 5 Sekunden-Intervall die URI *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*.

Abbildung 8.1.: Screenshot: Veränderung der Statusanzeige, wenn keine Verbindung zum Internet besteht

8.4. Erweiterung um Offline-Nutzung

Die bisher vorstellten Komponenten um Umsetzungen führten zum Prototyp einer funktionierenden Single Page Applikation. Diese benötigt aber zur Nutzung noch eine Verbindung zum Webserver, auf der die Applikation gehostet wird und eine Verbindung zum Web Service, zur Durchführung von Interaktionen. In den folgenden Abschnitten wird beschrieben, wie Techniken eingesetzt wurde, um den Zugriff auf diese Ressourcen auf ein Minimum zu reduzieren.

[21] **?**, .

8.4.1. Implementierung des CachedHttpServices

Zur Reduzierung der Bindung an den Webservice wurde ein Cache implementiert. Hierbei wurde von der Planung des Caches aus Kapitel ((CACHE BESCHREIBUNG)) 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, Objekt zur Laufzeit beliebig zu verändern und zu erweitern. Darum kam die Idee auf, die Serverdaten, welche noch nicht an den Server gesendet wurde um Meta-Daten 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!**[22]-Datenbank anzulegen und zu verwalten. Sie wird von den meisten Browsern unterstützt[23]. Da *NoSQL*-Datenbanken keine festen Schema kennen, sondern beliebige Datenstrukturen per Index oder Schlüssel bestimmt, ist die Ablage eines dynamisch erstellten Objekts ohne weiteren Aufwand möglich. Zur Nutzung der IndexedDB wurde ein externes AngularJs-Module verwendet.[24]

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, welche wie folgt aussieht.

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.

[22] **NoSQL!**
[23] **?**, .
[24] **?**, .

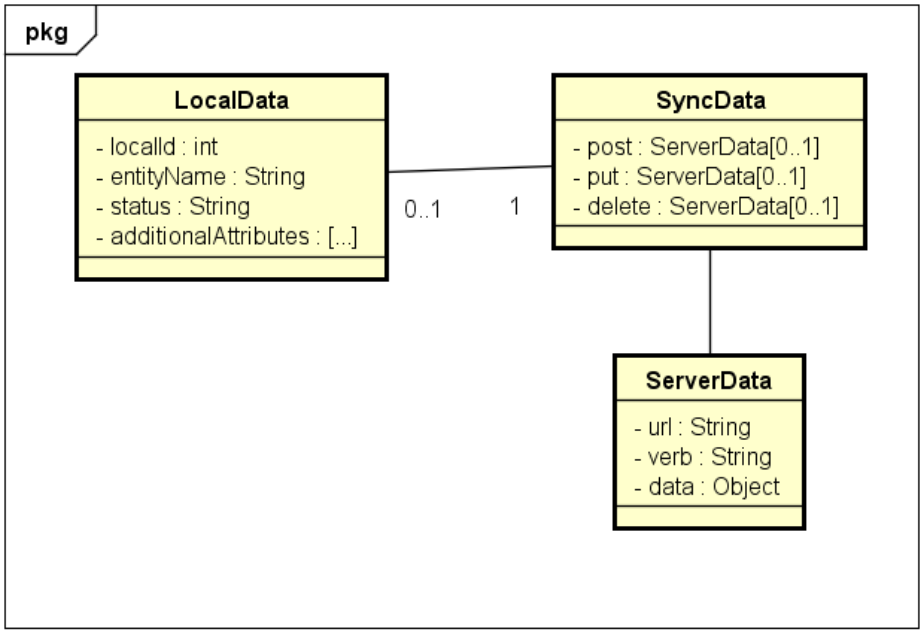

Abbildung 8.2.: Screenshot: Datenmodel zur Speicherung lokaler Daten

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* zwischen gespeichert. 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 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 einen offene Delete-Befehl, wird dieser als erstes durchgeführt. Alle weiteren offenen Synchronisationsprozesse werden im Zuge der Löschung dieses Datensatzes ebenfalls mit gelöscht. Dadurch wird die Anzahl der benötigten Requests verringert.
  Kommt es bei einem Objekt zu einem Create/Post- und Delete-Request, ohne, dass zwischenzeitlich eine Synchronisation durchgeführt wurde, wird der Datensatz direkt lokal gelöscht, 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. Dieser wird dann durchgeführt. Mit dem Ergebnis wird die ID des lokalen Datensatzes angepasst, damit eventuell anstehende Update/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 Synchronisationsprozesses, wird dieser aus dem Synchronisationsobjekt entfernt. Wenn dieses daraufhin keine offenen Prozesse 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 Webservice optional.

8.4.2. Das AppCache-Manifest

Der Einsatz des *cachedHttpService* erlaubt eine Nutzung der *SPA* auch ohne, dass eine Verbindung zum Web Service besteht. Doch noch braucht die *SPA* eine Verbindung zu ihrem Host, damit statische Dateien wie *View*-Templates und *Script*-Dateien nachgeladen werden können. Diese Verbindungen können durch die Nutzung eines AppCaches stark reduziert werden.
Der *AppCache* wurde im Zuge von *HTML5* implementiert und wird von allen gängigen

Browsern in der aktuellen Version unterstützt[25][26]. 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 Beispiel 8.4 ab Zeile 5ff.).

```
1  [U+FFFD]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
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 *
```

[25] **?**, .
[26] **?**, .

```
35
36 SETTINGS:
37 prefer-online
```

Quelltext 8.4: Cache-Manifest-Datei

Sind diese Dateien erstmal auf dem Client gespeichert, werde sie genutzt, wenn keine Verbindung zu 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, darum gibt es einen Weg, 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 zu Server aufbaut, wird die neue Manifest-Datei heruntergeladen. Dies für dazu, dass der Client alle lokalen Dateien invalidiert und sich die Dateien, welche das neue Manifest-Datei definiert, erneut herunterläd. 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 Anforderungen aus für den Prototyp umgesetzt hat. Hierbei wurde der Funktionsumfang mit dem Chrome der Version 44.0.2403.157 getestet.

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, so dass 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 SPA

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 Web-Techniken und unter Nutzung geeigneter Frameworks lässt sich sehr leicht ein einheitliches Aussehen schaffen, welche für verschiedene Anzeigegrößen optimiert wurde. Hierbei ist man nicht nur auf mobile Endgeräte beschränkt sondern erhält quasi nebenbei eine Webseite, die bequem eine Desktop-Anwendung ersetzen kann. Auch die Umsetzung der Business-Logik konnte ohne großen Einarbeitung-Aufwand bewerkstelligt werden. Dabei fällt auf, dass durch das Voranschreiten von HTML5 viele Funktionen, welche vor einigen Jahren nur durch

Desktop Applikationen umgesetzt werden, 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, aber noch nicht alle Techniken für den Browser umgesetzt. So ist die Umsetzung der *IndexedDB* für iOS und Microsoft-Geräte noch sehr fehleranfällig[1]. In diesem Punkt spiegelt sich auch das größte Problem jeder Web-Umsetzung wieder: Unterschiedliche Browser implementieren einige Apis anders oder teilweise auch gar nicht, sodass vieles der Entwicklungszeit für das Anpassen der Funktionen und Oberflächen für die verschiedenen Browser genutzt werden muss. Wenn es nun so ist, dass Kern-Komponenten wie in unserem Fall die IndexedDB in einigen wichtigen Browsern (iOS Safari-Nutzung bei 7.33% (v. 8.1-8.4 Stand 31.08.2015[2]) nicht ausreichen unterstützt werden, ist die Umsetzung dieses Teilaspekts für den produktiven Einsatz fast unmöglich.
Ein weiterer Nachteil ergibt sich aus der Nutzung von AngularJs. Da die gesamte Datenaufbereitung mit Authentifizierung und dem Routing auf Seiten des Clients passiert, können die lokal gespeicherten Daten mit Hilfe der Entwicklungswerkzeuge des Browsers einfach ausgelesen werden. Darum wäre es unter Sicherheitsaspekten fahrlässig, die hier vorgestellte Implementierung der Authentifizierung (siehe Kapitel 8.3.2) ohne weitere Sicherheitsmaßnahmen produktiv zu stellen.

9.2. Umsetzung als native App

Des Weiteren sollen auch die Vor- und Nachteile der Umsetzung als native Applikation vorgestellt werden.

[1] **?**, .
[2] **?**, .

9.2.1. Vorteile

Die Vorteile einer nativen Applikation liegen besonders in dem umfangreichen Funktionsumfang. Dieser kann alle bereitgestellten Funktionen des Betriebssystems ausnutzen. Dazu zählen das *Threading* (siehe Kapitel 7.1.4) und interne Aufrufe über *Services*, die dann zum Beispiel zum Versenden vom Emails verwendet werden können. Zudem können Daten persistent, auch über die Dauer einer *Session* hinaus, auf dem Gerät gespeichert werden (siehe Lokale Datenbank in Kapitel 7.3.3).
Eine hohe Sicherheit kann in dem Zuge einer nativen App ebenfalls bereitgestellt werden, da die Daten nur mit guten technischen Kenntnissen ausgelesen werden können. Sind die Daten darüber hinaus noch lokal verschlüsselt, so sind diese sicher. Weiterhin ist die gesamte Logik der Applikation nicht sichtbar für den Endanwender und von Manipulationen bei Datenabrufen kann man ausschließen. 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 unterstützende Betriebssystems.
Darin liegt auch noch ein weiteres Problem. Um eine große Markt-Abdeckung mit einer nativen App zu erreichen, benötigt man mindestens eine iOS- und eine Android-Applikation. Dann besitzt den Zugang zu über 95% der Smartphone-Nutzer in Deutschland.[3]
Die Entwicklung für zwei Systeme kann daraufhin in zwei Möglichkeiten umgesetzt werden. Zum einen könnten native Apps in den jeweiligen Sprachen entwickelt werden. Zum anderen kann eine Multiplattform-Lösung, wie Xamarin-Platform es ist, eingesetzt werden. Dabei beschränkt sich der Funktionsumfang dann aber auf die grundlegenden Funktionen, wenn bei den verschiedenen Funktionen nicht noch zwischen den Systemen unterschieden wird.
Weiterhin ist das Erstellen von Oberflächen aufwendiger als bei einer *Single Page Application*.
Zudem müssen native Apps direkt auf das *Device* geladen werden und können nicht einfach über das Internet aufgerufen werden.

[3] **?**, .

9.3. Resultat: Weiterentwicklung als native App

Zusammenfassend kann festgehalten werden, dass die Nachteile der *Single Page Application* dahingehend überwiegen, dass die Sicherheit der Daten - besonders in Verbindung mit Vitaldaten - eine höhere Priorität einnimmt. Diese Anforderung kann nur von einer nativen App zufriedenstellend geleistet werden. Hierbei zeigt sich die bisher unzureichende Implementierung der Datenbank in einigen Browsern zum Zeitpunkt der Projektdurchführung als unzureichend für die Anforderungen einer Anwendung, welche auf verschiedenen gängigen Mobilgeräten laufen soll. Ändert sich der Zustand der Implementierung müsste diese Evaluation neu durchgeführt werden.

10. Weiterentwicklung eines Clients zu einem Messeprototyp

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 Messeprototypen 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 an 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 dann 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 sinnvoll.
Aufgrund dessen wurden Anpassungen vorgenommen, die nach Außen nicht sichtbar sind, die Stabilität und die Leistungsfähigkeit der App aber zu einem sehr großen Teil verbessert haben. So wurde die Synchronisation aus den Methoden der *Get*- und *Post*-Abfragen extrahiert und zentralisiert (Vergleich dazu in Kapitel 3.2.2).
Die Funktionalität des *Caches* wurde nur hinsichtlich der Synchronisation mit dem Server angepasst. Darüber hinaus wurden keine Änderungen vorgenommen und das Eintragen von Daten erfolgt jeweils beim Abrufen von Server-Daten, sowie beim Übertragen von Daten zum Server. Die Abbildung 10.1 verdeutlicht den neuen Ablauf des

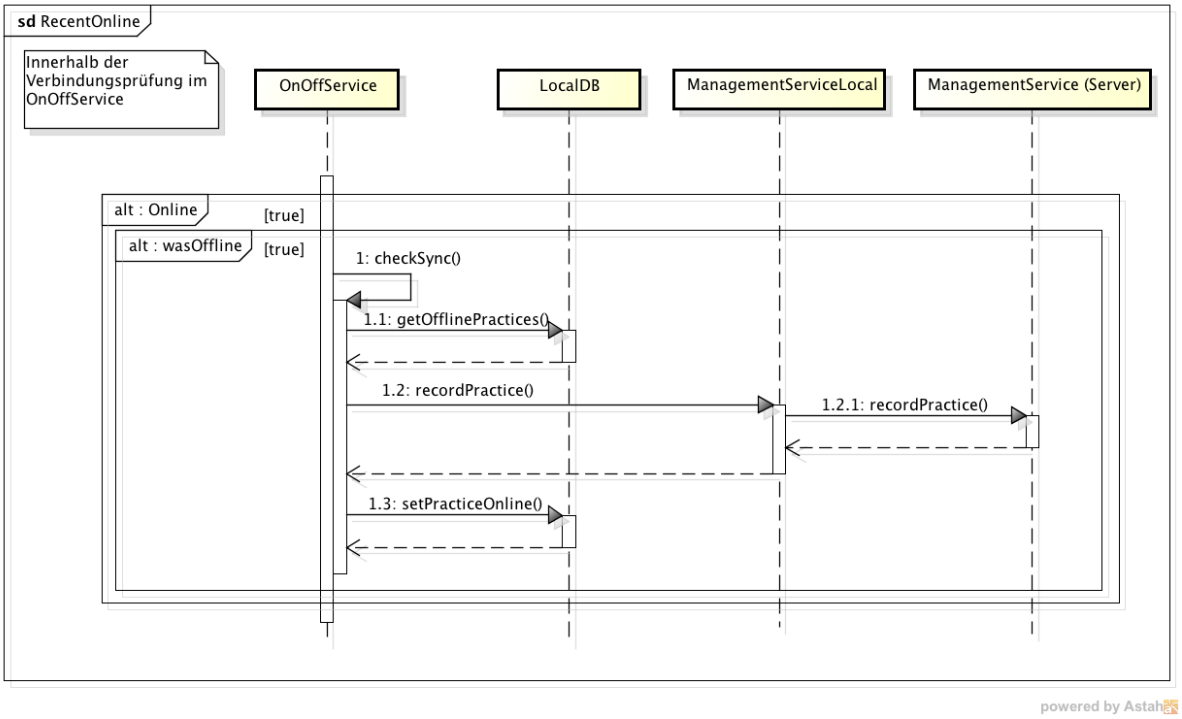

Abbildung 10.1.: Squenzdiagramm *RecentOnline*

Synchronisierens. Bedingung zum Start des Synchronisierens ist die Tatsache, dass die Applikation aktuell eine Verbindung zum Server besitzt und im vorhergehenden Status noch *offline* war. Falls dann im Vorfeld Daten angelegt wurden, die nur lokal gespeichert werden konnten, werden diese im Falle des Synchronisierens ausgelesen, auf dem Server gespeichert und in der lokalen Datenbank wieder als ßum Server synchronisiert"gekennzeichnet.
In diesem 2. Meilenstein wurde der Programmcode refaktorisiert, um die Umsetzung weiterer Funktionen 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 Verbidnungs-Operation kopiert werden müssen und hätte somit zu einer großen *Code*-Redundanz geführt. Diese Redundanz sollte unbedingt umgangen werden und deshalb ist diese zentralisierte Stelle zum Überprüfen der zu übetragenden Daten umgesetzt worden.

10.2. Anpassungen an der Oberfläche

Die Oberfläche wurde soweit angepasst, dass ein durchgängige *Corporate Design* erkennbar ist. Die Oberfläche, insbesondere *Buttons* und Dialoge wurden in diesem Schritt angepasst.
Dabei wurden zwei *Designs*, jeweils für den Login- und den Registrieren-*Button*, umgesetzt. Dazu war es nötig eine XML-Datei anzulegen, die dann die Eigenschaften der Schaltfläche zu beschreiben hat (Quellcode 10.1):

```
1  [U+FFFD]<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 des Login-*Buttons*

Weiterhin wurden die Dialogfenster mit Animationen versehen, um eine zum Betriebssystem passende *Usability* gewährleisten zu können. Dazu musste ähnlich zum *Button* eine XML-Datei angelegt werden und darin dann die Bewegungen des Fensters beschrieben werden (Quellcode 10.2).

```
1  [U+FFFD]<?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"/>
```

Quelltext 10.2: Dialog-Animation

Weiterhin wurden die *Icons* zur Kenntlichmachung des Verbindungsstatus gegen zum Design passende ausgetauscht und ein App-*Icon* eingefügt.

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. Zur Darstellung wird die *Xamarin-Extension BarChart* verwendet. Das Paket wird in das Projekt eingebunden und kann durch einen einfachen Aufruf mit der Übergabe eines Daten-*Arrays* verwendet werden (Quellcode: 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 *StatisticActivity* übergeben, in welcher dann die benötigten Daten abgerufen werden.

```
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,
               exerciseId);
14
15         List<float> zahlen = new List<float>();
16         foreach (var item in practices)
17         {
```

```
18              zahlen.Add(Convert.ToSingle(item.Repetitions *
                    item.NumberOfRepetitions * item.Weight));
19          }
20
21          var data = zahlen.ToArray();
22          var chart = new BarChartView(this)
23          {
24              ItemsSource = Array.ConvertAll(data, v => new BarModel { Value = v
                    })
25          };
26          AddContentView(chart, new ViewGroup.LayoutParams(
27            ViewGroup.LayoutParams.FillParent,
                  ViewGroup.LayoutParams.FillParent));
28      }
29      catch(Exception ex){[...]}
30  }
```

Quelltext 10.3: Statistik *Activity*

10.4. Fazit aus Meilenstein 2

Zusammenfassend kann festgehalten werden, dass der zweite Meilenstein zur Optimierung der nativen Adnroid-Applikation beigetragen hat. So konnte durch das Auslagern der Synchronisation Last vom dem *UIThread* genommen werden. Die daraus folgenden Vorteile wurden bereits in Kapitel 7 vorgestellt.
Das es sich am Ende der Entwicklung um einen Messeprototypen handeln soll, ist das Design von besonderem Interesse. Diese Anforderung wurde im ersten Meilenstein zurückgestellt, um zuallererst einen Fokus auf den technischen Vergleich legen zu können. Nach der Sondierung der besseren Möglichkeit sollte diese dann weiter ausgebaut werden. Deshalb hat diese App eine Verbesserung der Oberflächen erhalten.
Die Statistik (ein Kann-Kriterium aus A.3) wurde zusätzlich umgesetzt, da das Interesse an der Umsetzung eines langen Klicks und der Möglichkeiten der *BarChart-Extension* groß waren.

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

11. Fazit

Anfangs wurde ein Überblick über die Aufgabe gegeben und die Systemarchitektur vorgestellt. Darauf aufbauend wurden die verwendeten Technologien erörtert und die Umsetzungen der beiden Applikationen bis zu einem festgelegten Punkt dokumentiert. Die Implementierung des Servers wurde zusätzlich betrachtet. Die anschließende Gegenüberstellung der Apps hat ergeben, dass die Weiterentwicklung der Android-App in Hinsicht auf Sicherheit als einzige Lösung gesehen werden kann.
Deshalb wurde für diese Applikation ein zweiter Meilenstein begonnen, um die Anforderungen an den umzusetzenden Prototypen zu implementieren.

11.1. Ziele / Ergebnisse

Rückblickend kann das Projekt als ein großer Erfolg gesehen werden. Die eingangs formulierten Ziele konnten vollständig umgesetzt werden. Zudem wurden teilweise noch Funktionen umgesetzt, die über das formulierte Ziel hinaus gehen.
Zusammenfassend ist es nun mit dem Prototypen möglich den kompletten Funktionsumfang der nativen Android-App auch im *Offline*-Modus nutzen zu können.

11.2. Erkenntnisse

Der Erkenntnisgewinn dieser Arbeit ist beträchtlich. Es wurden ausschließlich unbekannte und für die Autoren neue Technologien verwendet. Die Einarbeitung geschah in den meisten Fällen reibungslos, war jedoch auch ein vorher unbekanntes Risiko, welches ein Problem in der Umsetzung hätte verursachen können.

Des Weiteren ist augenscheinlich, dass der Umgang mit den mobilen System, sei es Android über Xamarin oder Webapplikation über AngularJS, vertieft wurde. Darüber hinaus wurde mit dem dahinter fungierenden Server eine Einheit geschaffen, die die Erweiterung des Prototypen auch für eine größere Menge Benutzer ermöglicht.

11.3. Ausblick

Die Grundlage für den Ausbau dieses Projektes zu einer marktreifen App ist allemal gegeben. Der dahinter liegende Server ist stark genug, um eine größere Last an Anfragen zu bewältigen. Zum Ausbau dieses Projekts muss die Android-App weiterentwickelt werden. Die Umsetzung aller wichtiger Funktionen wurde jeweils an mindestens einem Beispiel im Messeprototypen 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.

Abkürzungsverzeichnis

ACL Access Control Lists
URI Uniform Resource Identifier
AES Advanced Encryption Standard
HTTP Hyper Text Transfer Protocol
CRUD Create Read Update Delete
OR-Mapper objekt-relationaler Mapper
.apk Android Package
VM Virtuelle Maschine
CORS Cross-origin resource sharing
PCL Portable Class Library
API Application Programming Interface
RAM Random Access Memory
SRAM Statischer RAM
DRAM Dynamischer RAM
TCP Transmission Control Protocol
XML Extensible Markup Language

Glossar

App

Kleine Programme/Applikationen für mobile Endgeräte. 3

OR-Mapper

Ein Framework zur Überführung von relationalen Tupeln in objekt-orientierte Objekte. 27

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

A. Anhang

A.1. Pflichtenheft

Pflichtenheft

A.2. Pflichtenheft

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

A.3. Pflichtenheft

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.4. Cache Post

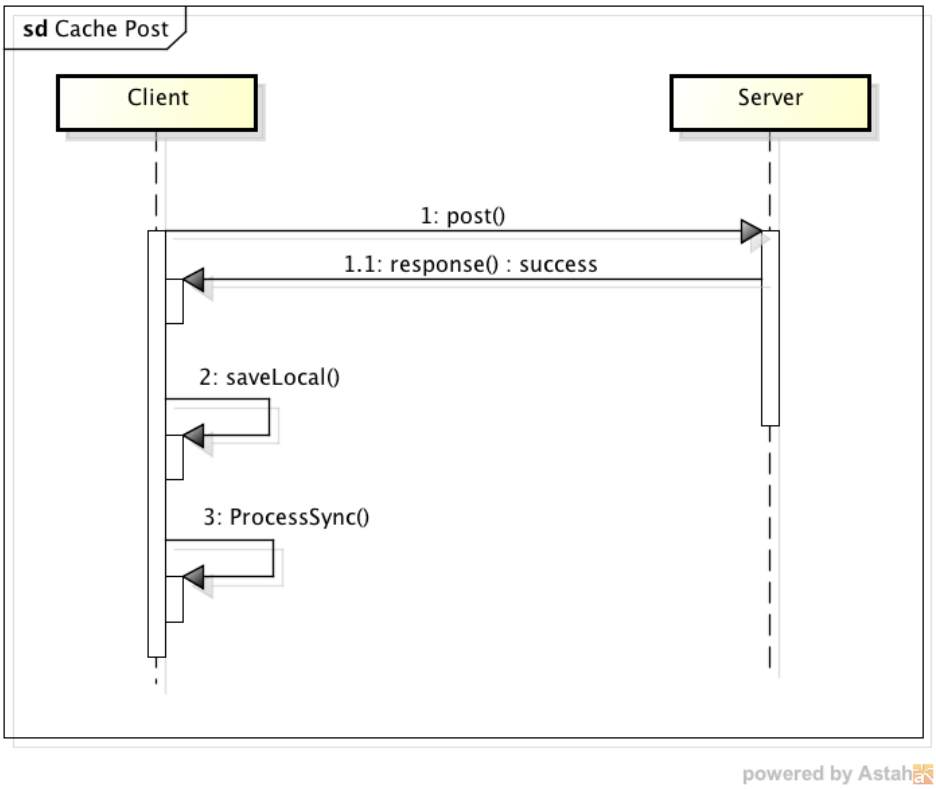

Abbildung A.1.: Hochladen zum Server

B. 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, 4. 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, 4. September 2015 Kevin Schie / Stefan Suermann