Bachelorthesis

Verlässliche mobile Anwendungen

Untersuchungen am Beispiel einer Fitness-App

Am IT-Center Dortmund GmbH
Studiengang IT- und Softwaresysteme
erstellte Bachelorthesis
zur Erlangung des akademischen Grades
Bachelor of Science

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, 27. August 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 Applikationen (Apps) - kleine Programme für mobile Endgeräte - immer mehr in den Fokus. Apps haben sich im Laufe der Zeit im Alltag breit gemacht und sind mittlerweile für den Endnutzer unverzichtbar geworden. Sei es beim Online-Shopping, Chatten oder der Navigation. Überall finden Applikationen 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 nicht vollständig und ausreichend im ganzen Land verfügbar.

Auf Grund dessen ist es notwendig, dass die Applikationen 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 wurden, sollen auch aufgenommen und später zur Verfügung gestellt werden.

[1] SCHMIDT: Anzahl der Smartphone-Nutzer in Deutschland in den Jahren 2009 bis 2015 (in Millionen).

Zur Umsetzung dieser Idee bestehen zwei Möglichkeiten. Zum einen kann eine mobile Web- oder eine native Applikation genutzt werden. [Zitat eines Gurus]

1.2. Zielsetzung

Das Ziel dieser Arbeit soll es sein, die Architektur für eine verlässliche Applikation zu entwerfen. Zum einen wird die Verarbeitung und Umsetzung auf einem Windows-Server erläutert. Auf der anderen Seite werden parallel zwei Applikationen zum verlässlichen Zugriff entwickelt und anhand dessen beleuchtet, welche Umsetzung für den angegeben Sachverhalt angemessener erscheint. Für die Umsetzung der Webapplikation wird das ASP.Net-Framework verwendet. Die native Applikation wird aus technischen Gründen mit Hilfe von Xamarin für Android entwickelt. Die Auswahl des Android-Betriebssystems besteht darin, dass Tests auch ohne Komplikationen oder Beschränkungen des Hersteller auf eigenen Geräten problemlos durchgeführt werden können.
Die vorteilhaftere Möglichkeit wird zu einem Prototypen mit rudimentären Funktionen und Design weiterentwickelt. Dabei besteht dann die Möglichkeit einen Trainingsplan zu erstellen und auf die Trainingsdaten der letzten fünf Trainings - unabhängig von der Internetverbidnung - zuzugreifen.

1.3. Vorgehensweise

Nachdem nun die Notwendigkeit von verlässlichen Applikationen und das Ziel der Arbeit definiert wurden, befasst sich das folgende Kapitel 2 mit der Problemanalyse im Hinblick auf die Umsetzung mit den beiden herangezogenen Varianten nativer- und Webapplikation.

2. Problemanalyse

In diesem Kapitel wird das Gesamtproblem näher beleuchtet. Darauf aufbauend werden die grundsätzliche Komponenten und deren Funktionsweise beschrieben.

- konkrete Zeile
- Frühe Entscheidungen

Ziele:

- Datenaustausch zwischen Server und Client
- Authorisierung und Authetifizierung
- Ausfallsicherheit
  - Prüfung der Verfügbarkeit des Servers
  - Verhalten bei Nicht-Verfügbarkeit des Servers
- Synchronisation nach Ausfall

3. Grundlagen

- Wie funktioniert ein Cache?
- Welche Arten gibt es ?
  - Store Forward
  - Function Cache (klare Abgrenzung)
- Sequendiagramme Caches
- 80% zielführend
- 20% gefälliger Stil

4. Architektur

- Client-Server Architektur
- Use-Case-Diagramme
- ER-Diagramme
- 80% zielführend
- 20% gefälliger Stil

5. Aspekte der Realisierung

5.1. Entwicklungsumgebung

5.2. DB-System

5.3. Hosting-Plattform

5.4. Testing (evtl)

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 werden nochmal besonders auf Sicherheitsaspekte in Verbindung mit RESTful-Architekturen eingegangen.

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 als Träger-Protokoll genutzt um eine einfache Interoperabilität zu gewährleisten.[1] Die dabei angeforderten Daten werden in der Regel im XML- oder JSON-Format übermittelt.

6.1.1. RESTful Webservices

Da Webservices in der Regel HTTP[2] als Protokoll verwenden, wurde die Idee zur Implementierung eines Webservices erweitert, um die Möglichkeiten des Protokolls vollständig zu benutzen. Heraus kam das Programmierparadigma REST (*Representational State Transfer*). Mit einem REST-Server bzw. einem RESTful Webservice bezeichnet

[1] BOOTH et al.: Web Services Architecture
[2] Hyper Text Transfer Protocol

man einen Webservices, welcher die strikte Nutzung von HTTP als Programmierparadigma umsetzt. Dies meint, dass sich, wie im Internet üblich, hinter einer URI[3], genau eine einzige Ressource verbirgt. Bei einer Ressource geht man von Daten aus. Im Gegensatz zu anderen Webservice-Implementierungen stellen RESTful Webserver keine Methoden oder aufrufbare Funktionalitäten zu Verfügung. Diese Daten werden vom REST-Service über eine eindeutige URL bereitgestellt. Dies hat den Vorteil, dass die Schnittstele leicht und eindeutig beschrieben werden kann, da ein Aufruf einer URL an den REST-Service immer die gleichen Daten liefert.
In den meisten Fällen, wie auch in den Anwendungsfällen dieser Arbeit, soll der Webservice CRUD[4]-Funktionalitäten bereitstellen. Damit die Schnittstelle nicht durch unnötig viele unterschiedliche URLs aufgebläht wird, sieht der RESTful-Ansatz die Verwendung der verschiedenen HTTP-Verben vor. Hierbei 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 unterscheiden. 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 einer Element URI.[5]

Um den Rahmen der Arbeit nicht zu sprengen, 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 Element aufgerufen. Hierbei antwortet der Server dem HTTP-Standard folgend mit dem Status-Code 200 bei erfolgreicher Suche oder 404, wenn keine Ressource gefunden wurde.
Das POST wird zur Erstellung neuer Inhalte verwendet. Bei Nutzung von Element URIs

[3] Uniform Resource Identifier
[4] Create Read Update Delete
[5] **?**, .

wird versucht die ID für das neue Element zu benutzten. In der Regel wird das ID Mamagement aber auf dem Server implementiert, so dass in der Regel 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).

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, Sitzungen auf dem Server zu verwalten. Dadurch kann ein RESTful Webservice sehr leicht skaliert werden.

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.[6]

[6] **?**, .

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!**[7] aufgebaut. Hierbei dienen die Objekte, welche aus Tupeln der Datenbank erstellt werden als Model-Klassen. Bevor diese Daten dann über WebApi ausgespielt werden, werden sie vom Model in ein View-Model übertragen. Durch diese Kapselung kann, im Sinne des Design-Pattern Hierbei wird nach dem Grundgedanken des **Seperation of Concerns!**[8], klar zwischen den Models für die Datendank und denen für die WebApi unterschieden werden.

6.2.1. Aufbau der Datenbank

Da bei der Umsetzung des Projekts konsequent auf Produkte von Microsoft gesetzt wurde, wurde als Datenbanksystem MS SQL gewählt. Dies hat den den Vorteil, dass das **Microsoft Entity Framework!**[9] als OR-Mapper[10] 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.
Grundlage für Datenbankklassen ist das Interface IEntity(Quellcode **??**):

```
1  [U+FFFD]using System;
2  using System.Collections.Generic;
3
4  namespace fIT.WebApi.Repository.Interfaces.CRUD
5  {
6      public interface IEntity<T>
7      {
8          T Id { get; set; }
9      }
10 }
```

[7] **MVVM!**
[8] **Seperation of Concerns!**
[9] **Microsoft Entity Framework!**
[10] objekt-relationaler Mapper

Quelltext 6.1: Basisinterface für DB-Repräsentationen

Das Interface gewährleistet, dass jede Datenbank-Entität einen eindeutigen Schlüssel besitzt. Eine konkrete Implementierung für die Trainingspläne sieht man im Quellcode-Beispiel 6.2:

```
1  [U+FFFD]using System.Collections.Generic;
2  using System.ComponentModel.DataAnnotations;
3  using fIT.WebApi.Repository.Interfaces.CRUD;
4
5  namespace fIT.WebApi.Entities
6  {
7      /// <summary>
8      /// Definiert einen Trainingsplan
9      /// </summary>
10     public class Schedule: IEntity<int>
11     {
12         public Schedule(int id, string name = "", string userId = "",
               ICollection<Exercise> exercises = null)
13         {
14             this.Id = id;
15             this.Name = name;
16             this.UserID = userId;
17             this.Exercises = exercises;
18         }
19
20         public Schedule(): this(-1){}
21
22         /// <summary>
23         /// DB ID
24         /// </summary>
25         public int Id { get; set; }
26         /// <summary>
27         /// DisplayName des Trainingsplans
28         /// </summary>
29         [Required]
30         public string Name { get; set; }
31
```

```
32          /// <summary>
33          /// Fremdschluessel zum Nutzer (per Namenskonvention)
34          /// </summary>
35          public string UserID { get; set; }
36
37          /// <summary>
38          /// Uebungen (per Namenskonvention)
39          /// </summary>
40          public virtual ICollection<Exercise> Exercises { get; set; }
41      }
42  }
```

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 gut, wie das Entity Framework über Namenskonventionen Verbindungen zwischen Entitäten auflöst. Auf Grund des Aufbaus der Klasse *Schedule* wird eine Fremdschüssel-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

Die andere Seite dieser

6.2.2. Aufbau der WebApi

((HTTP-Verben POST)) Im vorliegend Server wurden nur Collection URIs verwendet, um einen neuen Eintrag zu einer Entität anzulegen. Der Server antwortet mit den Daten des neu erstellten Objekts. Dies enthält die ID und die URI zum neu erstellten Objekt.
Mit dem HTTP-Verb PUT wird eine vorhandene Ressource geändert. Wird das Management von IDs clientseitig implementiert, können Anfragen mit dem PUT-Verb auch zur Erstellung einer neuen Ressource benutzt werden, wenn die ID bisher nicht auf dem Server vorhanden ist. Da die Verwaltung der IDs im vorliegenden Projekt ausschließlich serverseitig durchgeführt wird, wurde das Anlegen von Objekten ausschließlich per POST-Anfrage realisiert.

Swagger

6.3. Authentifizierung & Autorisierung

6.3.1. OAuth2

6.3.2. JWT and Bearer Token

6.3.3. Zugriff per CORS

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 eingehalten

[1] **App!**
[2] **Android!**
[3] **monolithisch!**
[4] **Linux!**
[5] ALLIANCE: Application Fundamentals.
[6] **Sandbox!**

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!**[13]-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.[14]

[7] ALLIANCE: System Permissions.
[8] **Java!**
[9] **Compiler!**
[10] Virtuelle Maschine
[11] Android Package
[12] BECKER, Arndt/PANT, Marcus: Android - Grundlagen und Programmierung. 1. Auflage. Heidelberg: dpunkt.verlag GmbH, 2009, Seite 17-19.
[13] **XML!**
[14] ALLIANCE: User Interface.

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 *WindowManager* 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 sollten, 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. Ein weiterer Vorteil besteht 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.

7.1.4. Threading/Asynchronität

7.1.5. SQLite

7.2. Was ist XAMARIN?

7.2.1. Multiplattform-Unterstützung

7.2.2. Besonderheiten der Android-Umsetzung

Man muss z.B. Activities nicht manuell im Manifest eintragen -> macht XAMARIN für einen!

7.3. Eigene Umsetzung

7.3.1. Anlegen der Layouts

7.3.2. Konnektivität zum Server

7.3.3. Lokale Datenbank

7.3.4. Umsetzung des Caches

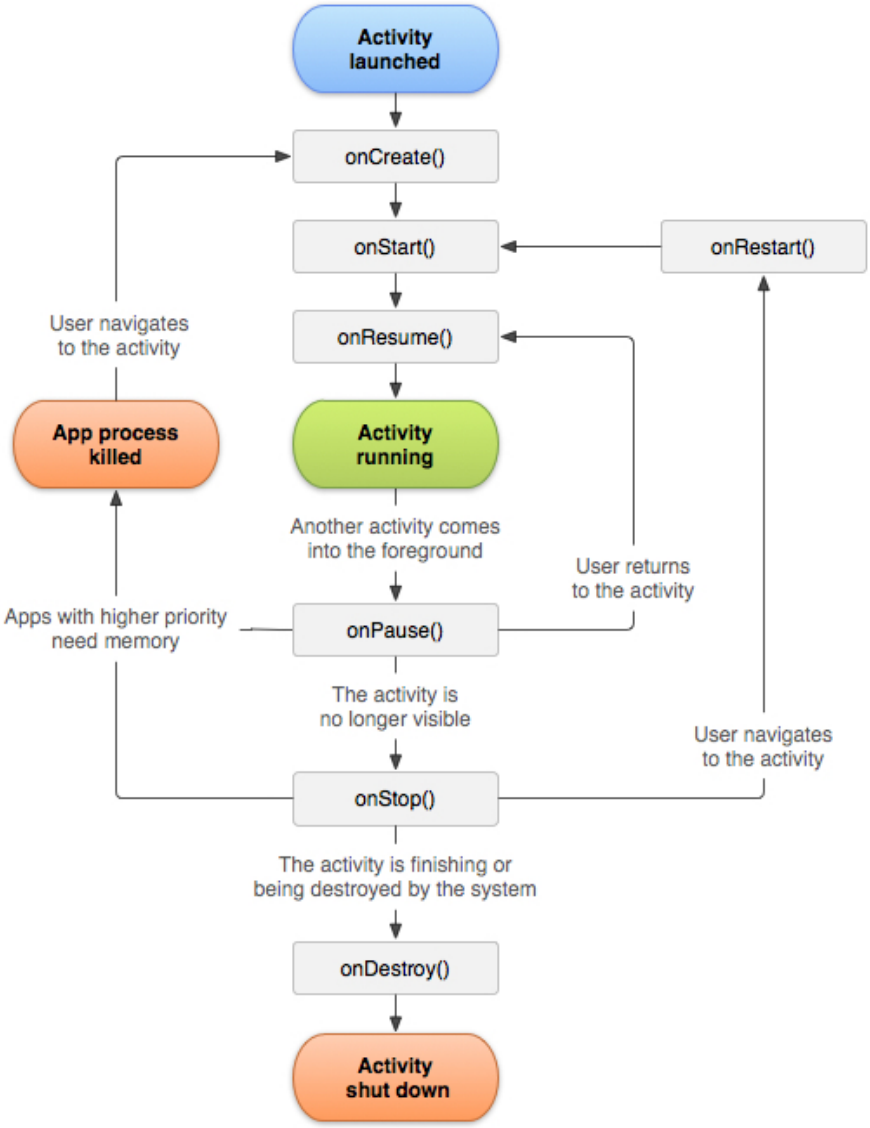

Abbildung 7.1.: Android Activity-Lifecycle

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

8. Realisierung der clientseitigen Implementierung als Webapplikation

8.1. Definition einer Single Page Application

8.2. AngularJs

8.2.1. MVC

8.2.2. Services

8.2.3. Promises

8.2.4. Routing

8.3. Umsetzung

8.3.1. Layout mit Twitter Bootstrap

- Was ist das
- Vorteile: responsives Verhalten

8.3.2. Online-Check

8.3.3. Herausforderung statusloses Protokol Http

- Login

8.4. CachedHttpService mit IndexedDB

8.4.1. Exkurs IndexedDB

8.4.2. Http-Verbs

Umsetzung von Caching auf basis der Http-Verben statt einer konkreten implementierung für jede entity

8.4.3. Synchronisation zwischen Server und SPA

8.5. Herausforderungen

- IndexedDB nicht voll implementiert
- Code vollständig einsehbar: Probleme mit sensiblen Daten

9. Gegenüberstellung der clientseitigen Implementierungen

10. Fazit

10.1. Ziele / Ergebnisse

10.2. Erkenntnisse

10.3. Ausblick

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

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

ALLIANCE, Open Handset: Activities. August 2015 ⟨URL: https://developer.android.com/guide/components/activities.html⟩

ALLIANCE, Open Handset: Application Fundamentals. August 2015 ⟨URL: https://developer.android.com/guide/components/fundamentals.html⟩

ALLIANCE, Open Handset: Content Providers. August 2015 ⟨URL: https://developer.android.com/guide/topics/providers/content-providers.html⟩

ALLIANCE, Open Handset: Intents and Intent Filters. August 2015 ⟨URL: https://developer.android.com/guide/components/intents-filters.html⟩

ALLIANCE, Open Handset: Processes and Threads. August 2015 ⟨URL: https://developer.android.com/guide/components/processes-and-threads.html⟩

ALLIANCE, Open Handset: Services. August 2015 ⟨URL: https://developer.android.com/guide/components/services.html⟩

ALLIANCE, Open Handset: System Permissions. August 2015 ⟨URL: https://developer.android.com/guide/topics/security/permissions.html⟩

ALLIANCE, Open Handset: User Interface. August 2015 ⟨URL: https://developer.android.com/guide/topics/ui/overview.html#Layout⟩

BECKER, Arndt/PANT, Marcus: Android - Grundlagen und Programmierung. 1. Auflage. Heidelberg: dpunkt.verlag GmbH, 2009

BOOTH, David et al.: Web Services Architecture. Februar 2004 ⟨URL: http://www.w3.org/TR/ws-arch/⟩

SCHMIDT, Holger: Anzahl der Smartphone-Nutzer in Deutschland in den Jahren 2009 bis 2015 (in Millionen). Juni 2015 ⟨URL: http://de.statista.com/statistik/daten/studie/198959/umfrage/anzahl-der-smartphonenutzer-in-deutschland-seit-2010/⟩

A. 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, 27. August 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, 27. August 2015 Kevin Schie / Stefan Suermann