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

Irgendwo muss noch die Rollen-Definition für den Server hin(Admin und User)

- 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 wird gesondert auf Sicherheitsaspekte in Verbindung mit RESTful-Architekturen eingegangen. Die hier beschriebene WebApi kann über die **URL!**[1] 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[2] als Träger-Protokoll genutzt, um eine einfache Interoperabilität zu gewährleisten.[3] Die dabei angeforderten Daten werden in der Regel im **XML!**[4]- oder **JSON!**[5]-Format übermittelt.

[1] **URL!**
[2] Hyper Text Transfer Protocol
[3] BOOTH et al.: Web Services Architecture
[4] **XML!**
[5] **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!**[6] 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[7]-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

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

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*.[8] Diese können mit verschiedenen HTTP-Verben kombiniert werden[9].

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

[8] KURTZ, Jamie/WORTMAN, Brian: ASP.NET Web API 2: Building a REST Service from Start to Finish. 2. Auflage. New York: Apress, 2014, ISBN 978–1–484–20109–1, S. 12ff..
[9] TILKOV, Stefan et al.: REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web. 3. Auflage. Heidelberg: dpunkt, 2013, ISBN 978–3–864–90120–1, S. 26ff..

Statuscode *200 (Ok)* geantwortet und bei Fehlern mit *400 (Bad Request)* oder *404 (Not Found)*[10].

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[11].

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.[12]

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!**[13] 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

[10] TILKOV et al.: REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web, S. 26ff..
[11] A. a. O.
[12] A. a. O.
[13] **MVVM!**

Grundgedanken des **Seperation of Concerns!**[14], klar zwischen den Models für die Datendank und den ViewModels, welche die WebApi benutzt, 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!**[15] gewählt. Dies hat den den Vorteil, dass das **Microsoft Entity Framework!**[16], welches sehr gut in für die Nutzung mit einer WebApi optimiert ist, als OR-Mapper[17] 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 Bespielen 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 public interface IEntity<T>
2 {
3   T Id { get; set; }
4 }
```

Quelltext 6.1: Basisinterface für DB-Repräsentationen

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

```
1    // Definiert einen Trainingsplan
2    public class Schedule: IEntity<int>
3    {
4        public Schedule(int id, string name = "", string userId = "",
             ICollection<Exercise> exercises = null)
5        {
```

[14] **Seperation of Concerns!**
[15] **MS SQL!**
[16] **Microsoft Entity Framework!**
[17] objekt-relationaler Mapper
[18] DYKSTRA: Getting Started with Entity Framework 6 Code First using MVC 5.

```
6           this.Id = id;
7           this.Name = name;
8           this.UserID = userId;
9           this.Exercises = exercises;
10      }
11
12      public Schedule(): this(-1){}
13
14      // DB ID
15      public int Id { get; set; }
16      // DisplayName des Trainingsplans
17      [Required]
18      public string Name { get; set; }
19      // 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 *Sche-*

[19] **einwertige Fremdschüssel!**
[20] **mehrwertigen!**

dule eine Verbundstruktur besitzt, welche *Exercises* heißt, wird implizit eine Verbindung zwischen den Relationen in der Datenbank angelegt.[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  // 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);
```

[21] DYKSTRA: Getting Started with Entity Framework 6 Code First using MVC 5.
[22] **ASP.NET Web API 2!**
[23] KURTZ/WORTMAN: ASP.NET Web API 2: Building a REST Service from Start to Finish, S. 2ff..

```
22      }
23
24      var datamodel = this.TheModelFactory.CreateModel(schedule);
25      await this.AppRepository.Schedules.AddAsync(datamodel);
26      var result = this.TheModelFactory.CreateViewModel(datamodel);
27      return CreatedAtRoute("GetScheduleById", new { id = schedule.Id },
            result);
28    }
29  }
```

Quelltext 6.3: POST-Methode zur Erstellung eines Trainingsplans

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

[24] WASSON: Attribute Routing in ASP.NET Web API 2.
[25] WASSON: Model Validation in ASP.NET Web API.
[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] SWAGGER: Swagger.
[29] **NuGet!**
[30] JOUDEH: ASP.NET Web API Documentation using Swagger.

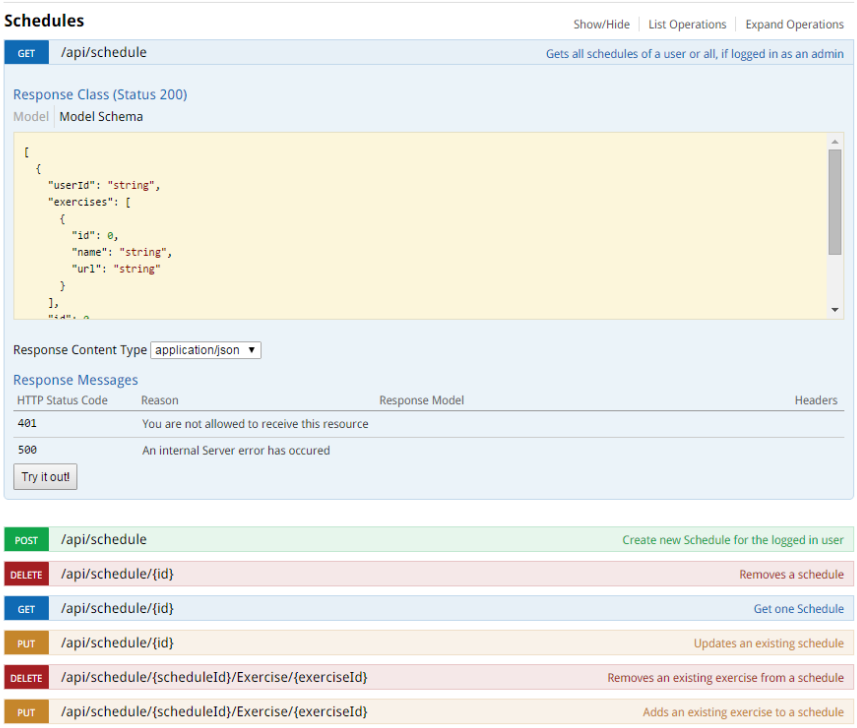

Abbildung 6.1.: Screenshot: Swagger UI der Web Api

6.3. Authentifizierung & Autorisierung

Wie bereits in Kapitel ((Da wo die Rollen definiert werden)) 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]

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 kommu-

[31] **Role-Based Access Models!**
[32] JOUDEH: Token Based Authentication using ASP.NET Web API 2, Owin, and Identity.

nizieren, nämlich Client, Resource Owner, Authorization Owner und Resource Server.[33]

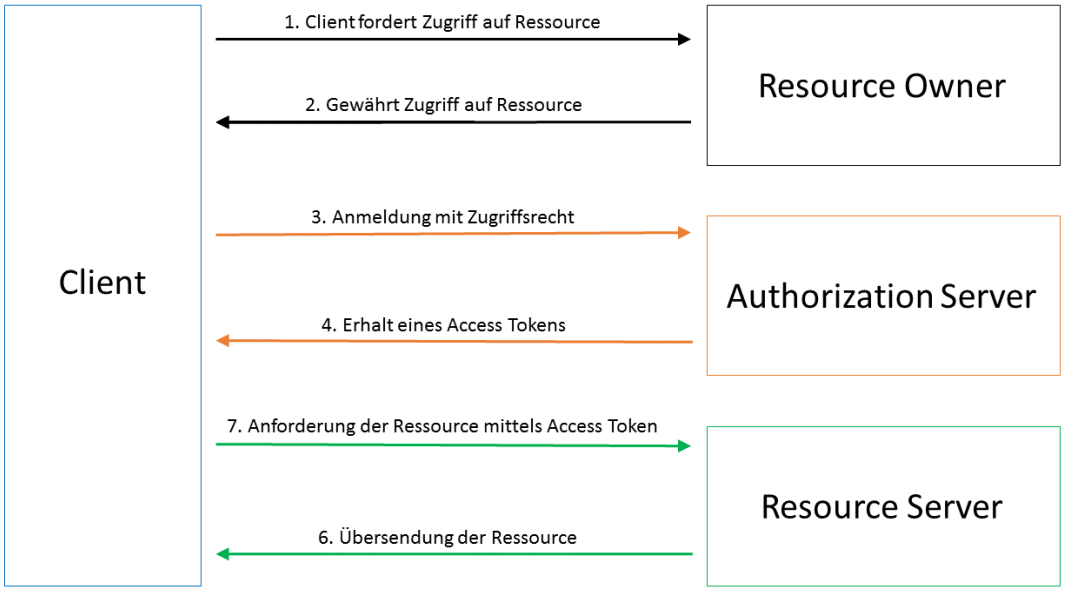

Abbildung 6.2.: Resourcenzugriff durch OAuth2

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 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

[33] STEYER, Manfred/SOFTIC, Vildan: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript. Köln: O'Reilly Germany, 2015, ISBN 978–3–955–61951–0, S. 286.

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

[34] **Grant!**
[35] JOUDEH: Implement OAuth JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1.
[36] JOUDEH: ASP.NET Identity 2.1 Roles Based Authorization with ASP.NET Web API.
[37] STEYER/SOFTIC: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, S. 287.
[38] **Single-Sign-On!**
[39] A. a. O., S. 294.

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]

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

[40] STEYER/SOFTIC: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, S. 289.
[41] **base64!**
[42] ATLASSIAN: Understanding JWT.

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 standardisierter 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.

[43] **HTTPS!**
[44] JOUDEH: Enable OAuth Refresh Tokens in AngularJS App using ASP .NET Web API 2, and Owin.
[45] Cross-origin resource sharing
[46] **Ajax!**
[47] **Same-Origin-Policy!**
[48] **JSONP!**
[49] STEYER/SOFTIC: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, S. 102.

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].

[50] **Preflight!**
[51] STEYER/SOFTIC: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript, S. 102.
[52] **Polyfills!**
[53] CANIUSE.COM: Can I Use: Cross-Origin Resource Sharing.

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!**-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] 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] 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. 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 Wei-

teren 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. 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

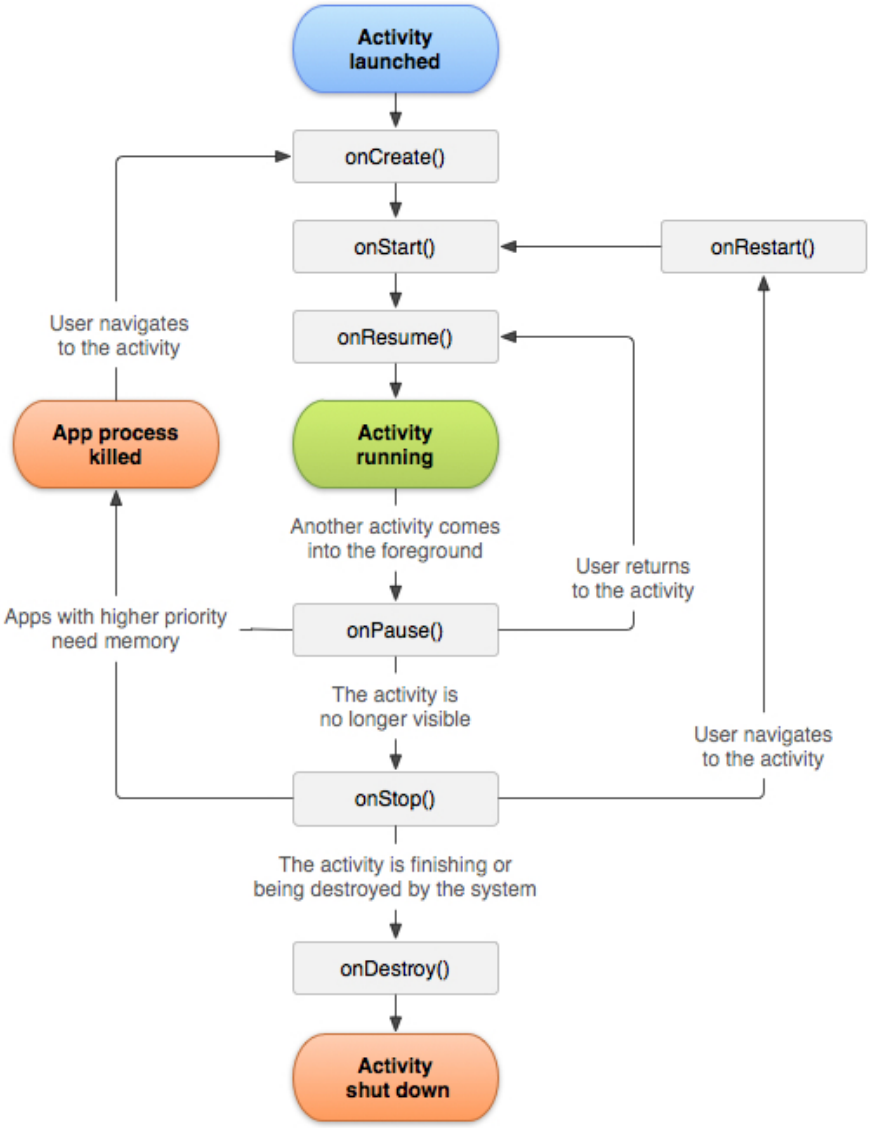

Abbildung 7.1.: Android Activity-Lifecycle

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

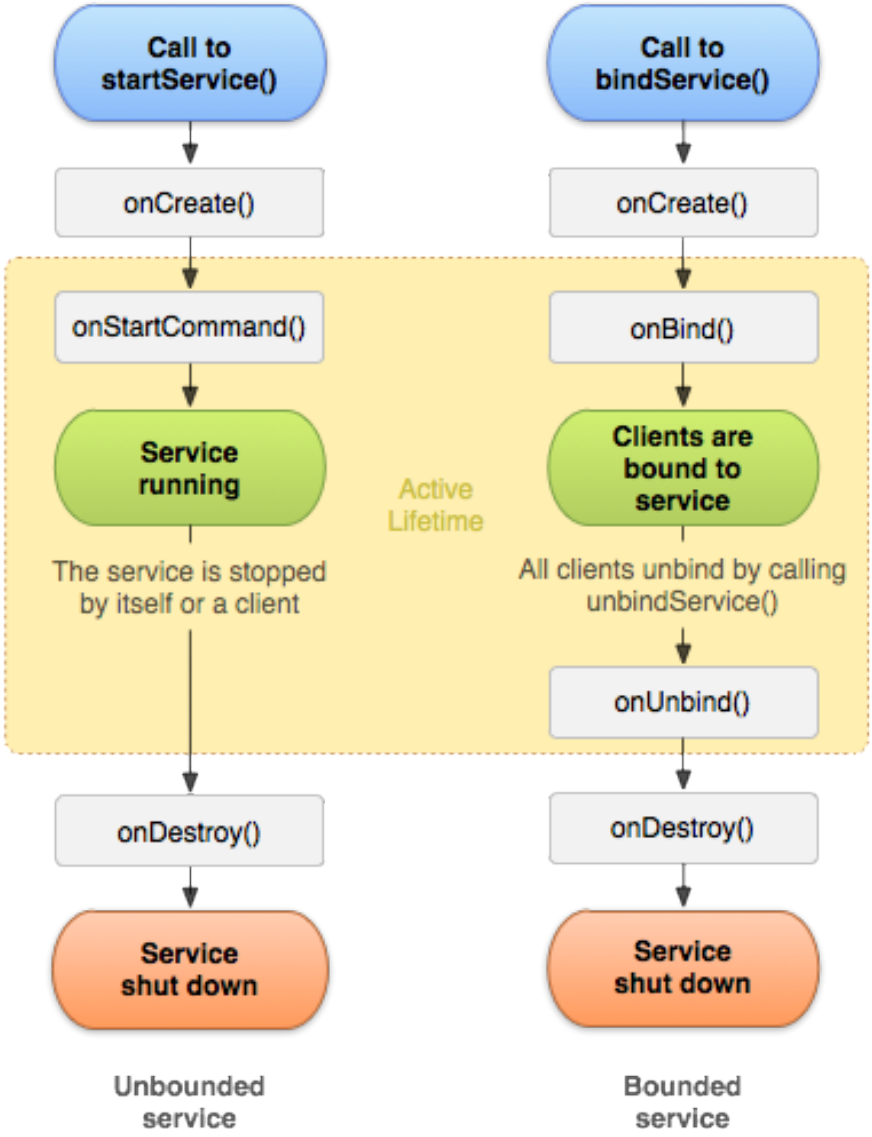

Abbildung 7.2.: Android Service-Lifecycle

Quelle: https://developer.android.com/guide/components/services.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
CORS Cross-origin resource sharing

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

ALLIANCE, Open Handset: Activities. August 2015 ⟨URL: https://developer.android.com/guide/components/activities.html⟩ – Zugriff am 2015-08-26

ALLIANCE, Open Handset: Application Fundamentals. August 2015 ⟨URL: https://developer.android.com/guide/components/fundamentals.html⟩ – Zugriff am 2015-08-26

ALLIANCE, Open Handset: Content Providers. August 2015 ⟨URL: https://developer.android.com/guide/topics/providers/content-providers.html⟩ – Zugriff am 2015-08-26

ALLIANCE, Open Handset: Intents and Intent Filters. August 2015 ⟨URL: https://developer.android.com/guide/components/intents-filters.html⟩ – Zugriff am 2015-08-26

ALLIANCE, Open Handset: Processes and Threads. August 2015 ⟨URL: https://developer.android.com/guide/components/processes-and-threads.html⟩ – Zugriff am 2015-08-26

ALLIANCE, Open Handset: Services. August 2015 ⟨URL: https://developer.android.com/guide/components/services.html⟩ – Zugriff am 2015-08-26

ALLIANCE, Open Handset: System Permissions. August 2015 ⟨URL: https://developer.android.com/guide/topics/security/permissions.html⟩ – Zugriff am 2015-08-26

ALLIANCE, Open Handset: User Interface. August 2015 ⟨URL: https://developer.android.com/guide/topics/ui/overview.html#Layout⟩ – Zugriff am 2015-08-26

ATLASSIAN: Understanding JWT. 2014 ⟨URL: https://developer.atlassian.com/static/connect/docs/latest/concepts/understanding-jwt.html⟩ – Zugriff am 27.08.2015

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

BLUESMOON: Flowchart showing Simple and Preflight XHR. August 2015 ⟨URL: https://upload.wikimedia.org/wikipedia/commons/c/ca/Flowchart_showing_Simple_and_Preflight_XHR.svg⟩ – Zugriff am 27.08.2015

BOOTH, David et al.: Web Services Architecture. Februar 2004 ⟨URL: http://www.w3.org/TR/ws-arch/⟩ – Zugriff am 2015-08-26

CANIUSE.COM: Can I Use: Cross-Origin Resource Sharing. 2015 ⟨URL: http://caniuse.com/#feat=cors⟩ – Zugriff am 27.08.2015

DYKSTRA, Tom: Getting Started with Entity Framework 6 Code First using MVC 5. 2015 ⟨URL: https://www.asp.net/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application⟩ – Zugriff am 27.08.2015

JOUDEH, Taiseer: ASP.NET Web API Documentation using Swagger. August 2014 ⟨URL: http://bitoftech.net/2014/08/25/asp-net-web-api-documentation-using-swagger/⟩ – Zugriff am 27.08.2015

JOUDEH, Taiseer: Enable OAuth Refresh Tokens in AngularJS App using ASP .NET Web API 2, and Owin. Juli 2014 ⟨URL: http://bitoftech.net/2014/07/16/enable-oauth-refresh-tokens-angularjs-app-using-asp-net-web-api-2-owin/⟩ – Zugriff am 27.08.2015

JOUDEH, Taiseer: ASP.NET Identity 2.1 Roles Based Authorization with ASP.NET Web API. März 2015 ⟨URL: http://bitoftech.net/2015/03/11/asp-net-identity-2-1-roles-based-authorization-authentication-asp-net-web-api/⟩ Zugriff am 27.08.2015

JOUDEH, Taiseer: Implement OAuth JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1. Februar 2015 ⟨URL: http://bitoftech.net/2015/02/16/

implement-oauth-json-web-tokens-authentication-in-asp-net-web-api-and-identity-2/⟩ – Zugriff am 27.08.2015

JOUDEH, Taiseer: Token Based Authentication using ASP.NET Web API 2, Owin, and Identity. Juli 2015 ⟨URL: http://bitoftech.net/2014/06/01/token-based-authentication-asp-net-web-api-2-owin-asp-net-identity/⟩ – Zugriff am 27.08.2015

KURTZ, Jamie/WORTMAN, Brian: ASP.NET Web API 2: Building a REST Service from Start to Finish. 2. Auflage. New York: Apress, 2014, ISBN 978–1–484–20109–1

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/⟩ – Zugriff am 2015-08-26

STEYER, Manfred/SOFTIC, Vildan: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript. Köln: O'Reilly Germany, 2015, ISBN 978–3–955–61951–0

SWAGGER: Swagger. 2015 ⟨URL: http://swagger.io/⟩ – Zugriff am 27.08.2015

TILKOV, Stefan et al.: REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web. 3. Auflage. Heidelberg: dpunkt, 2013, ISBN 978–3–864–90120–1

WASSON, Mike: Model Validation in ASP.NET Web API. 2012 ⟨URL: http://www.asp.net/web-api/overview/formats-and-model-binding/model-validation-in-aspnet-web-api⟩ – Zugriff am 27.08.2015

WASSON, Mike: Attribute Routing in ASP.NET Web API 2. Januar 2014 ⟨URL: http://www.asp.net/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2⟩ – Zugriff am 27.08.2015

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