Praktikum

Mobile Computing

Winter 2015/2016

REST - Tutorial

1 Ziel

Das *Client-Server-Modell* ist das Standardkonzept für die Verteilung von Aufgaben innerhalb eines Netzwerks. Ein *Server* ist dabei ein Programm, das eine bestimmte Software-Funktionalität bereitstellt. Im Rahmen des Client-Server-Konzepts können andere Programme, die *Clients*, diese Funktionalität nutzen. Dabei können sich Clients und Server durchaus auf unterschiedlichen Rechnern befinden – sie müssen es aber nicht unbedingt! Das Programmieren und Testen von einfachen Client-Server-Anwendungen kann daher auch relativ komfortabel auf einem einzigen Rechner erfolgen und nach Fertigstellung auf die Zielrechner verteilt werden. Dieses Tutorial zeigt ein einfaches Beispiel für eine Client-Server-Beziehung zwischen zwei Anwendungen mittels eines **REST-Services**.

2 REST

***Re**presentational **S**tate **T**ransfer* bezeichnet ein Programmierparadigma für verteilte Systeme, insbesondere für Webservices. REST ist eine Abstraktion der Struktur und des Verhaltens des World Wide Webs. REST hat das Ziel einen Architekturstil zu schaffen, der den Anforderungen des modernen Webs besser entspricht. Dabei unterscheidet sich REST vor allem in der Forderung nach einer einheitlichen Schnittstelle von anderen Architekturstilen.

2.1 JSON

Die ***JavaS**cript **O**bject **N**otation*, ist ein kompaktes Datenformat in einer einfach lesbaren Textform zum Zweck des Datenaustauschs zwischen Anwendungen. Jedes gültige JSON-Dokument soll ein gültiges JavaScript sein und per `eval()` interpretiert werden können. JSON ist unabhängig von der Programmiersprache. Parser existieren in praktisch allen verbreiteten Sprachen.

Insbesondere bei Webanwendungen und mobilen Apps wird es in Verbindung mit JavaScript, Ajax oder WebSockets zum Transfer von Daten zwischen dem Client und dem Server häufig genutzt.

Im Gegensatz zu XML, welches als Alternative auch häufig zur Datenübertragung genutzt wird, ist JSON meist wesentlich kompakter.

JSON (226 Byte)	**XML** (279 Byte)

`{`
`"Herausgeber": "Xema",`
`"Nummer": "1234-5678-9012-3456",`
`"Deckung": 2e+6,`
`"Waehrung": "EURO",`
`"Inhaber": {`
`  "Name": "Mustermann",`
`  "Vorname": "Max",`
`  "maennlich": true,`
`  "Hobbys": [ "Reiten", "Golfen", "Lesen" ],`
`  "Alter": 42,`
`  "Kinder": [],`
`  "Partner": null`
`}`
`}`
	`<Kreditkarte`
	`  Herausgeber="Xema"`
	`  Nummer="1234-5678-9012-3456"`
	`  Deckung="2e+6"`
	`  Waehrung="EURO">`
	`  <Inhaber`
	`  Name="Mustermann"`
	`  Vorname="Max"`
	`  maennlich="true"`
	`  Alter="42"`
	`  Partner="null">`
	`  <Hobbys>`
	`  <Hobby>Reiten</Hobby>`
	`  <Hobby>Golfen</Hobby>`
	`  <Hobby>Lesen</Hobby>`
	`  </Hobbys>`
	`  <Kinder />`
	`  </Inhaber>`
	`</Kreditkarte>`

3 Ein einfacher REST-Server mit Spark

Spark[1] ist ein minimalistisches Framework zur Erstellung von Web-Servern in Java (Java 8 wird benötigt). Durch die vergleichsweise einfache Handhabung können REST-Server schnell erstellt und benutzt werden.

Spark bietet Unterstützung für alle benötigten HTTP-Requests über statische Methoden, wie `Spark.get(…)` und `Spark.post(…)`.

Bei REST-Servern hat jeder HTTP-Request eine spezielle Bedeutung:

GET	Anfordern von Daten
POST	Erstellen von Daten
PUT	Erstellen und Ändern von Daten
DELETE	Löschen von Daten

Anmerkung: PUT wird im Gegensatz zu POST meist idempotent realisiert, d.h. wiederholte Aufrufe mit denselben Parametern führen zum selben Ergebnis.

3.1 Erste Schritte

Für die Entwicklung eines REST-Servers erstellen Sie zuerst ein neues Modul in Android Studio vom Typ „Java-Library" mit einem geeigneten Namen, wie z.B. „RestServer".

Um die benötigten Bibliotheken automatisch zu importieren werden diese in das Build-Skript von Android Studio eingefügt. Dazu öffnen Sie die `build.gradle`-Datei des RestServer-Moduls und ergänzen folgende Zeilen in die *„dependencies"*, damit es etwa wie folgt aussieht:

```
apply plugin: 'java'

dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    compile 'com.sparkjava:spark-core:2.3' //hinzufügen
    compile 'com.google.code.gson:gson:2.5' //hinzufügen
}
```

Nach dem Speichern der Datei wird eine Warnmeldung (Gradle files have changed since last project sync) erscheinen, welche mit „Sync Now" am rechten Bildschirmrand bestätigt werden muss. Sollte diese Meldung nicht erschienen sein muss der Prozess manuell über „Tools → Android → Sync Project with Gradle Files" gestartet werden. Nun werden alle benötigten Bibliotheken automatisch von Android Studio heruntergeladen und hinzugefügt. Um dies zu überprüfen müssen Sie im Project-Navigator auf der linken Seite auf „Project" umschalten und den Eintrag „External Libraries" aufklappen. Hier sollten nun diverse Einträge vorhanden sein, wie etwa: gson, jetty-…, slf4j, spark-core, websocket-… .

Erstellen Sie eine Hauptklasse (z.B. *MyRestServer.java*) mit *main*-Methode und fügen sie eine Run-Configuration für eben diese hinzu.

[1] Spark Website: http://sparkjava.com/

In die *main*-Methode werden wir nun einen Handler für einen HTTP-Request einfügen:

```
Spark.get("/", new Route()
{
      public Object handle(Request req, Response res)
      {
          return "Hallo Welt!";
      }
});
```

Starten Sie nun Ihren REST-Server über den Play-Button und laden folgenden Link im Browser:

http://localhost:4567/

Im Browser-Fenster erscheint nun die von Ihnen angegebene Ausgabe. Sollte der Start des Servers durch einen `UnsupportedClassVersionError` verhindert werden, stellen Sie sicher, dass sie mit **Java 8** arbeiten.

3.1.1 Lambdas

Alle benutzten Methoden in diesem Tutorial, die sich auf Spark beziehen, können elegant durch Lambdas ersetzt werden. Dies macht den Code deutlich kompakter. Im Tutorial wird weiterhin die komplette Schreibweise vorgestellt. Das obige Beispiel sieht mit Lambdas wie folgt aus:

```
Spark.get("/", (req, res) -> "Hallo Welt!");
```

Mehr Informationen zu Lambdas oder Spark allgemein können beispielsweise in der offiziellen Dokumentation gefunden werden:

http://sparkjava.com/

3.2 Liste der User

Falls bisher nicht geschehen legen Sie eigenständig eine `User`-Klasse an, die die Datenbankeinträge (siehe Tutorial Kapitel 4) repräsentieren.

Fügen Sie einen neuen HTTP-Request-Handler für eine Abfrage der User aus der Datenbank hinzu. Idealerweise soll die angesteuerte Methode eine Liste von Usern zurückgeben (List<User>).

```
Spark.get("/users", new Route()
{
      public Object handle(Request req, Response res)
      {
          return queryUsersFromDatabase(); //returns List<User>
      }
});
```

Starten Sie den REST-Server erneut und öffnen Sie den Link um alle User abzufragen:

http://localhost:4567/users

Die Ausgabe sollte in etwa so aussehen:

```
[de.restserver.User@181b35c2, de.restserver.User@16dcc8b1]
```

Diese Ausgabe ist verständlicherweise nicht geeignet um sie am Client zu verarbeiten. Das Problem hieran ist, dass Spark auf jedem zurückzugebenen Objekt die *toString()*-Methode aufruft. Die Java Implementierung der *toString()*-Methode besteht standardmäßig aus dem Paket- und Klassennamen sowie dem HashCode in Hexadezimal.

3.3 JSON-Objekte erstellen

Um nicht manuell jede *toString()*-Methode so umzubauen, dass sie einen JSON-String zurückgibt, wird als nächstes ein *ResponseTransformer* implementiert. Dieser wandelt alle Objekte in JSON-Strings um. Die JSON-Bibliothek „Gson" bietet hierfür geeignete Methoden.

Erstellen Sie eine *JSONTransformer*-Klasse, die das Interface ResponseTransfromer implementiert, und benutzen Sie die Gson-Konvertierung.

```
public class JsonTransformer implements ResponseTransformer
{
    private Gson gson = new Gson();
    public String render(Object model) throws Exception {
        return gson.toJson(model);
    }
}
```

Fügen Sie nun noch ihrem bisherigen HTTP-Request-Handler den angelegten ResponseTransformer hinzu und testen den Aufruf erneut im Browser.

```
Spark.get(String path, Route route, ResponseTransformer transformer);
```

Die Ausgabe sollte nun wie folgt aussehen:

```
[{"vorname":"user1v","nachname":"user1n"},
{"vorname":"user2v","nachname":"user2n"}]
```

3.4 REST-Konformität (optional)

Wenn ein REST-Server sich konform zum de-facto-Standard verhalten soll, muss auch der Content-Type der HTTP-Anfragen/Antworten beachtet werden. Dies können Sie hier vernachlässigen, da hier nur die Grundlagen behandelt werden.

Um den Content-Type der Server-Antworten dennoch richtig einzustellen, können Sie mit der *before*- oder *after*-Methode von Spark den Content-Type verändern:

```
Spark.after(new Filter()
{
    public void handle(Request req, Response res) throws Exception
    {
        res.type("application/json");
    }
});
```

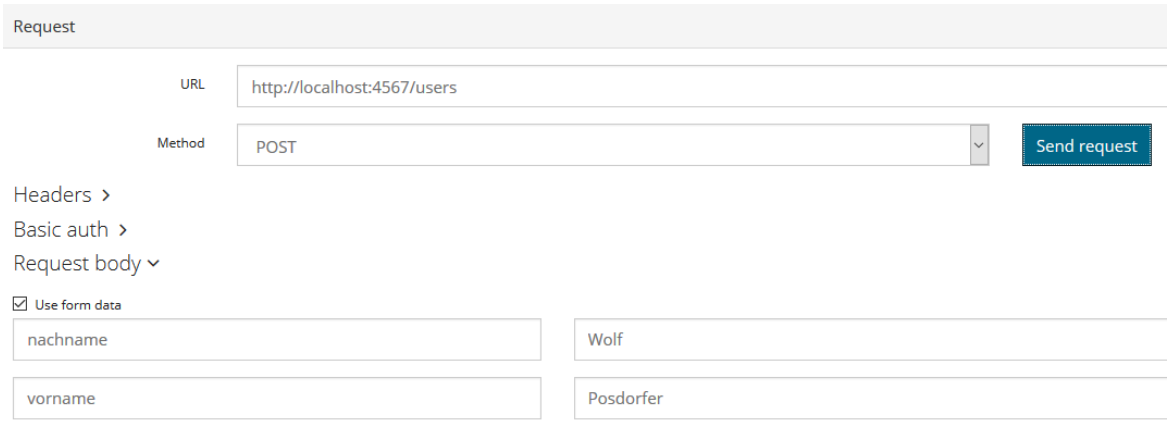
Nun wird nach jedem HTTP-Request-Handler der Content-Type der Ausgabe von `text/html` auf `application/json` geändert.

3.5 Daten hinzufügen

Das Hinzufügen neuer Daten wird in REST-Servern meist über den POST-Request realisiert. Erstellen Sie einen neuen POST-Request-Handler der auf *„/users"* Anfragen reagiert. Über die Methoden *queryParams(String p)* und *queryParams()* können Sie auf dem Request-Objekt die einzelnen Parameter abfragen oder alle vorhandenen Parameter auflisten.

Um POST-Anfragen im Browser zu testen, benötigen Sie ein Browser-Plugin wie *„Postman"*[2] in Chrome oder *„Rested"*[3] in Firefox.

Ein POST-Request um einen neuen User anzulegen sieht in Rested (nach Start über das „</>"-Icon in der Toolbar) so aus:



Legen sie einen POST-Request-Handler an, um neue User zu erstellen. Geben Sie als Antwort erstmal nur „success" oder „error" zurück. Verbesserte Antwortmöglichkeiten kommen in einem späteren Abschnitt.

[2] https://chrome.google.com/webstore/detail/postman-rest-client/fhbjgbiflinjbdggehcddcbncdddomop
[3] https://addons.mozilla.org/de/firefox/addon/rested/

3.6 User aktualisieren

Um Daten zu aktualisieren wird ein PUT-Request verwendet. Um viele Parameter zu vermeiden, die sich auf die Identifizierung des vorhandenen Objekts beziehen, werden diese als Teile der URL (im Beispiel „123456“) angegeben, etwa so:

http://rest.server.com/users/123456

Die zu ändernden Daten werden, wie bei POST-Requests, als Parameter übergeben. Um Spark mitzuteilen, dass es sich beim zweiten Teil obiger URL um variable Angaben handelt, wird dies durch einen Doppelpunkt (:) signalisiert. Ein PUT-Handler zum Ändern bestehender User könnte etwa so aussehen:

```
Spark.put("/users/:id", new Route()
{
      public Object handle(Request req, Response res)
      {
          String id = req.params(":id");
          String vorname = req.queryParams("vorname");
          String nachname = req.queryParams("nachname");

          boolean success = updateUser(id, vorname, nachname);

          ...
```

Der Parameter „:id“ dient dabei als eine Art Platzhalter für das URL-Suffix und würde, auf das Beispiel bezogen, den Wert „123456“ annehmen.

Erstellen Sie einen PUT-Request-Handler zum Aktualisieren von bestehenden Usern und geben Sie auch hier bei Erfolg und Misserfolg erstmal nur „success“ oder „error“ zurück.

3.7 „Bessere“ Server-Antworten

Bisher haben Sie nur einfache Fehler oder Erfolgsnachrichten an den Client zurückgesandt. Dies wollen wir nun durch eine kleine Hilfs-Klasse ändern.

Legen sie eine neue Klasse mit Namen *ResponseMessage* an. Dieser geben Sie ein Feld vom Typ String mit Namen „message“. Durch die Verwendung des vorher erstellten JSONTransformers wird diese Klasse in ein JSON-Objekt umgewandelt, welches nach der Umwandlung etwa so aussieht:

```
{ "message" : "meine nachricht" }
```

Dies ist hier von Vorteil, weil Sie so bei konsequenter Nutzung davon ausgehen können, dass immer ein JSON-Objekt mit einer Nachricht zurückkommt. Durch Vererbung können leicht weitere Felder ergänzt werden, wie etwa Fehler-Codes oder ähnliches.

Verändern Sie ihren bisherigen Request-Handler so, dass Sie die soeben erstellte *ResponseMessage* als Rückgabewert verwenden.

4 Rest-Client in Android

Im folgenden Abschnitt soll der REST-Server mit der Android-Anwendung kombiniert werden.

4.1 Android-Anwendung als Client des Servers

Eine Verbindung zu einem Server ist immer mit Unsicherheit verbunden und kann bei größeren Aufrufen auch das Übertragen vieler Daten beinhalten. Wenn eine Applikation über die Zeit komplexer wird, kann der Verbindungsaufbau, das Senden, Empfangen oder auch das Verarbeiten von Daten viel Zeit in Anspruch nehmen. Wenn diese Vorgänge in einer *Activity* direkt durchgeführt werden, kommt es für den Benutzer zu unangenehmen Pausen bei der Arbeit mit der Applikation, da die Anwendung während dieser Zeit keine anderen Aktionen, wie das Anzeigen des Verbindungsfortschrittes, durchführen kann.

Deshalb sollten zeitintensive Aktionen nebenläufig und unabhängig vom Ablauf einer Activity durchgeführt werden. Dieses kann über einen vom sogenannten *UI-Thread*, der durch die Activity genutzt wird, verschiedenen, sogenannten *Worker-Thread* geregelt werden. Um die Arbeit mit Threads zu erleichtern, gibt es in Android die Klasse `AsyncTask`. In dieser Klasse können Aktionen von der Activity ausgelagert werden, wobei die Klasse dann *asynchron* durch die Activity verwendet wird.

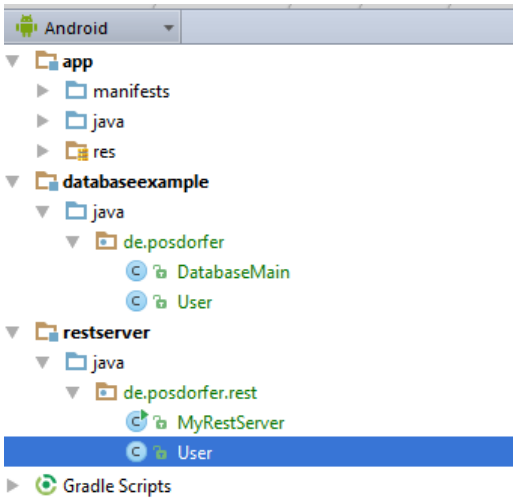
Informieren Sie sich deshalb zunächst über die AsyncTask Klasse. Beispielsweise auf der API Seite zu Android:

http://developer.android.com/reference/android/os/AsyncTask.html

Verstehen Sie dabei zumindest den Zusammenhang der Methoden *doInBackround*, *onPreExecute* und *onPostExecute,* sowie die Möglichkeit zur Parametrisierung der generischen Java Klasse.

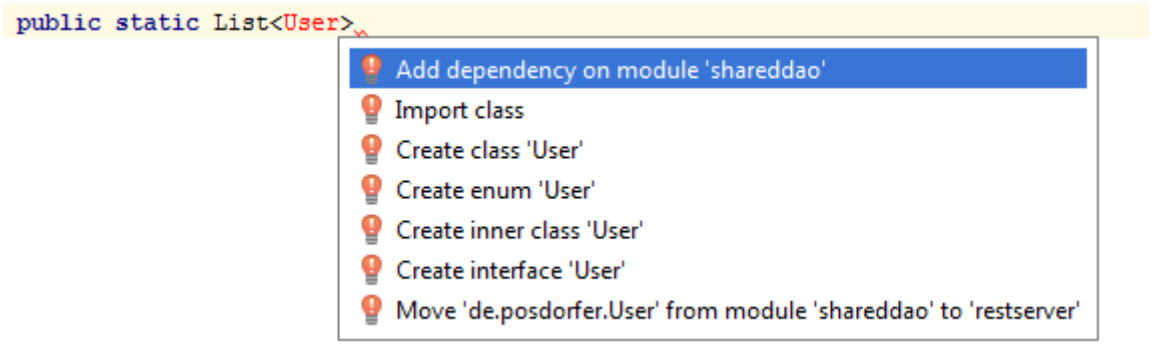
4.2 Refactoring

Wenn Sie dem bisherigen Tutorial aufmerksam gefolgt sind, sollte Ihr Projektlayout in etwa so aussehen, wie im nebenstehenden Bild.



Auffallen sollte Ihnen hier, dass die `User`-Klasse mehrfach vorkommt. Um Code-duplizierung zu vermeiden, legen Sie ein weiteres Modul vom Typ „*Java-Library*“ an. Wählen Sie einen geeigneten Namen wie *SharedDAO* (**D**ata **A**ccess **O**bject). In diesem Modul werden alle Klassen angelegt, die zum Datenaustausch zwischen Datenbank und REST-Server sowie REST-Server und Android-App dienen.

Über den *„Add dependency on module X"*-Dialog des Quick-Fixes können Sie das gemeinsame SharedDAO-Modul zum Buildpath von anderen Modulen hinzufügen.



Alternativ können Sie auch folgende Zeile in die *build.gradle* eines Moduls einfügen:

```
compile project (':shareddao')
```

Um Kompilierungsprobleme bei der Android App zu vermeiden, sollten Sie die Java-Version des SharedDAO-Modules auf Java 1.7 stellen. Dazu ändern Sie die *build.gradle* des Modules ab, so dass sie etwa wie folgt aussieht:

```
apply plugin: 'java'
sourceCompatibility = 1.7
targetCompatibility = 1.7

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

4.3 REST-Aufrufe

In Android gibt es viele verschiedene Möglichkeiten HTTP-Aufrufe zu tätigen. Grundsätzlich basieren alle auf der *HttpURLConnection*-Klasse. Machen Sie sich mit den Grundlagen dieser vertraut, um ein generelles Verständnis davon zu entwickeln.

http://developer.android.com/reference/java/net/HttpURLConnection.html

In diesem Tutorial wird jedoch ein Framework benutzt, um automatisch aus Java-Interfaces mit möglichst wenig Code ein funktionierendes REST-Interface zu generieren. *HttpUrlConnection* und *AsyncTask* werden im Framework verwendet, müssen aber nicht mehr explizit vom Entwickler benutzt werden.

Dazu importieren Sie, wie bereits im REST-Server-Abschnitt, automatisch einige Bibliotheken. Bearbeiten sie die *build.gradle* Datei des app-Modules damit sie etwa wie folgt aussieht:

```
apply plugin: 'com.android.application'

android {
        ... // Platzsparende Punkte ;-)
}

dependencies {
 compile fileTree(include: ['*.jar'], dir: 'libs')
 testCompile 'junit:junit:4.12'
 compile 'com.android.support:appcompat-v7:23.1.1'
 compile 'com.squareup.retrofit:retrofit:2.0.0-beta2' //hinzufügen
 compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2' //hinzufügen
 compile project(':shareddao')
}
```

Bestätigen Sie beim Speichern die Warnmeldung erneut mit „*Sync Now*" oder wählen Sie manuell „*Tools → Android → Sync Project with Gradle Files*", um das Importieren abzuschließen.

4.3.1 Retrofit-Framework

Das Retrofit Framework[4] generiert automatisch aus einem Java-Interface aufrufbare REST-kompatible Methoden über Annotationen. REST-Aufrufe werden einfacher und kompakter, da sich Entwickler nichtmehr selbst mit *HttpURLConnection*s oder *AsyncTasks* auseinandersetzen müssen.

Der erste Schritt dazu ist das Erstellen eines Java-Interfaces. Erstellen Sie ein neues Paket in der Android App um REST-Spezifische Klassen dort abzulegen.

Erstellen Sie ein Interface mit beliebigen Namen (z.B.: `APIService.java`). Fügen Sie eine Parameterlose-Methode ein, welche ein Objekt vom Typ `Call<List<User>>` zurückgibt.

In Retrofit werden alle REST-Aufrufe über Annotationen an den Interface-Methoden konfiguriert. Dazu gibt es Annotationen für alle gängigen HTTP-Requests (`@GET`, `@PUT`, …).

Um also die vorher implementierte REST-Schnittstelle zum Abrufen aller User aufzurufen, muss die Interface-Methode mit `@GET("/users")` annotiert werden.

```
public interface APIService
{
    @GET("/users")
    public Call<List<User>> users();
}
```

Als nächstes erstellen Sie eine Klasse (z.B. *APIFactory.java*), welche zum „Instanziieren" des Interfaces und Konfigurieren von Retrofit benutzt wird.

[4] Retrofit Website: http://square.github.io/retrofit/

Weil das Erzeugen des Interfaces durch Retrofit relativ „teuer" ist (auf alten Geräten einige Sekunden), bietet sich die Benutzung des „Singleton-Patterns"[5] an. Dadurch wird erreicht, dass es nur eine Instanz geben kann, welche auch nicht mehrfach erzeugt werden muss/kann. Zusätzlich ist die Speicherbelastung durch eine Klasse geringer, als wenn für jeden REST-Aufruf eine neue Klasse erzeugt wird.

Die APIFactory kann mit Singleton etwa so aussehen:

```
public final class APIFactory // final verhindert Vererbung
{
    private static final String API_URL = "http://10.0.2.2:4567";
    private static final APIFactory singleton = new APIFactory();

    private final APIService apiservice;

    private APIFactory() //privater Konstruktor kann nur innerhalb benutzt werden
    {
        Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(API_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build();
        apiservice = retrofit.create(APIService.class);
    }


    public static APIService getInstance()
    {
        return singleton.apiservice;
    }
}
```

Zu beachten: Die IP-Adresse des Host-Gerätes ist **10.0.2.2** und nicht *localhost* oder *127.0.0.1*.

Hiermit sind alle Voreinstellungen zur REST-Schnittstelle getroffen und sie kann verwendet werden.

4.3.2 Permissions einräumen

Da Sie mittels Retrofit auf eine externe Ressource (die Internet-Verbindung) des Gerätes zugreifen, muss die „AndroidManifest.xml" im Hauptverzeichnis ihres Projektes näher betrachtet werden, da unserer Anwendung die nötigen Rechte eingeräumt werden müssen. Fügen Sie deshalb

```
<manifest ...>
      <uses-permission android:name="android.permission.INTERNET"/>
      </application ...>...</application>
</manifest>
```

in die XML-Datei ein. Damit hat die Anwendung die Erlaubnis eine Verbindung zum Server aufzubauen.

Eine Liste von weiteren Permissions finden Sie unter:

http://developer.android.com/reference/android/Manifest.permission.html

[5] Singleton-Pattern: https://de.wikipedia.org/wiki/Singleton_%28Entwurfsmuster%29

4.3.3 Anwendung testen

Retrofit bietet für REST-Aufrufe zwei Varianten der Ausführung an, *synchron* und *asynchron*. Um den UI-Thread nicht zu blockieren und eine „stotternde" App zu entwickeln, sollten nur asynchrone Aufrufe getätigt werden.

Für synchrone Aufrufe wird `.execute()` und für asynchrone wird `.enqueue(…)` verwendet. Dies sind Methoden die durch das `Call`-Objekt (siehe APIService-Interface) von Retrofit bereitgestellt werden.

Um die Anwendung zu testen, kann in der *onResponse*-Methode des *Callbacks* eine Nachricht ausgegeben werden. Über *LoginActivity.this* kann dabei auf das Objekt der äußeren Klasse zugegriffen werden, welches den *Context* zur Anzeige auf dem Bildschirm darstellt.

```
Toast.makeText(LoginActivity.this, response.body().toString(),
    Toast.LENGTH_SHORT).show();
```

Abschließend betrachten Sie den Fehlerfall `onFailure(Throwable t)` und überlegen Sie, wie der Nutzer über Fehler informiert werden könnte.

4.4 Android Anwendung zur Darstellung des Datenbankabrufs erweitern

Abschließend sollen die aus der Datenbank abgerufenen Namen der Gruppenmitglieder statt in einer Toast-Nachricht noch in einer weiteren Aktivität dargestellt werden. Neue Aktivitäten können in Android über so genannte *Intents* gestartet werden. Das Konzept der *Intents* ist im Rahmen der Android-Entwicklung von großer Bedeutung, da mit ihnen durchzuführende Aktionen beschrieben werden. Sie sollten sich also entsprechend ausführlich mit dem Konzept vertraut machen.

Durch *Intents* können unterschiedliche Anwendungen (oder Teile von Anwendungen, z.B. Activities) zur Laufzeit kommunizieren. Das *`Intent`*-Objekt ist dabei eine passive, abstrakte Beschreibung von Aktionen, um bestimmte Anwendungen oder Aktivitäten zu starten oder zu beeinflussen. Sie bilden also eine Art Verbindungsglied zwischen Activities und Anwendungen. Nachfolgend soll eine erste Verwendung von *Intents* vorgenommen werden.

4.4.1

Nach dem Abrufen der Daten vom Server soll nun eine neue Activity aufgerufen werden. Erstellen Sie deshalb nach dem Vorbild der *`LoginActivity`* eine *`WelcomeActivity`*. Dazu sind folgende Schritte vorzunehmen:

- Erstellen Sie eine neue Activity über File → New → Activity → Empty Activity
- Passen Sie das Layout der Acitivity an. Als Anhaltspunkt kann das Layout rechts dienen. Dabei sollten zumindest ein *`TextView`* zur Begrüßung, sowie ein *`ListView`*, in dem die verschiedenen Namen angezeigt werden können, vorhanden sein (die Liste ist vorerst leer).
- Definieren Sie für den *`ListView`* ein eigenes XML Layout *list_item.xml*. In diesem ist das Aussehen der einzelnen Listeneinträge definiert (beispielsweise durch einen einzelnen TextView).
- Fügen Sie ihre `WelcomeActivity` nach dem Vorbild der `LoginActivity` in die *AndroidManifest.xml* ein, falls dies nicht bereits automatisch passiert ist. Einen *Intent-Filter* benötigen Sie in diesem Fall nicht.

4.4.2
Wenn der Login-Vorgang abgeschlossen ist, muss nun statt einer Nachricht an den Benutzer ein *Intent* erzeugt werden, mit dem die Aktivität gestartet wird. Versuchen Sie ein *Intent* an geeigneter Stelle zu erzeugen. Die *Intent* Klasse besitzt dazu folgenden Konstruktor:

```
public Intent(Context packageContext, Class<?> clazz)
```

Als `Context` kann, wie bei der Toast-Nachricht aus Abschnitt 4.3.3, die Activity selbst und als `Class<?>` die `WelcomeActivity`-Klasse verwendet werden. Anschließend kann über das Objekt der Activity-Klasse, die Methode `startActivity(Intent intent)` aufgerufen werden.

4.4.3
Nun sollte die Aktivität bereits nach dem Datentransfer starten. Allerdings müssen der Aktivität noch die Ergebnisdaten ihrer Datenbank-Anfrage sowie der aktuelle Login-Name mitgegeben werden. Sie können diese dem `Intent` mittels der Methode `putExtra` beifügen.

4.4.4
Schließlich müssen nun in der `WelcomeActivity` die Daten verwendet werden. Das `Intent` kann über die Methode `getIntent()` referenziert und die Extras über die entsprechenden `get…Extra`-Methoden des *Intents* ausgelesen werden.

Die GUI-Elemente können, wie in der `LoginActivity`, über die `R`-Klasse referenziert werden. Die `ListView` benötigt dabei noch einen sogenannten „Adapter", welcher das Verhalten der Liste in der Aktivität steuert. Als Adapter kann dabei `ArrayAdapter<String>` dienen, welchem neben dem `Context` und der Liste mit Namen auch das in Abschnitt 4.4.1 erzeugte *list_item*-Layout mitgegeben wird.

4.4.5
Testen Sie nun Ihre Anwendung und überprüfen Sie ob der Login-Name und die Namen in der Activity angezeigt werden.

5 HTTP Authentifizierung

HTTP-Authentifizierung ist ein Verfahren, mit dem sich der Nutzer eines Webclients gegenüber dem Webserver bzw. einer Webanwendung als Benutzer authentifizieren kann, um danach für weitere Zugriffe autorisiert zu sein.

Der Nutzer ist nach Ablauf des Anmelde-Protokolls gegenüber dem Webserver authentifiziert, allerdings gilt die Umkehrung nicht: Der Nutzer kann nicht sicher sein, dass der Webserver wirklich der ist, der er vorgibt zu sein.

Meldet der Server bei dem Zugriff auf eine Ressource den Statuscode *401 Unauthorized,* weiß der Nutzer/Client dass er sich authentifizieren muss.

5.1 BasicAuth

Die Basic Authentication nach RFC 2617 ist die häufigste Art der HTTP-Authentifizierung.

Wenn der Server in seinem Header

```
WWW-Authenticate: Basic realm="geschuetzterBereich"
```

fordert, muss der Client sich gegenüber dem Server über BasicAuth authentifizieren. Dazu sendet der Client seinen Nutzernamen und Password kodiert in Base64[6] im HTTP-Header an den Server.

```
Authorization: Basic dnNpczpwYXNzd29yZA==
```

Der „kryptische Teil" besteht hier aus dem Funktionsaufruf von

```
Base64.encode(„vsis:password")

// unter Android
Base64.encodeToString("vsis:password".getBytes(), Base64.NO_WRAP)
```

Wichtig: Die Codierung von Nutzername:Passwort in Base64 ist **keine** Verschlüsselung. Nutzername und Passwort könnten genauso gut im Klartext übertragen werden. Hier dient Base64 nur als Transport-Codierung. Bei Aufrufen von Webseiten wird hier über HTTPS (SSL/TLS) schon vor der Übermittlung des Passwortes für eine komplette Transportverschlüsselung gesorgt.

[6] Base64: https://de.wikipedia.org/wiki/Base64

5.2 BasicAuth in Spark

Um Ressourcen in Spark vor nicht authentifizierten Nutzern zu schützen, kann mithilfe der `Spark.before(„/*“, ...)`-Funktion jeder Request auf Authentifikation überprüft werden.

Implementieren Sie dazu eine `AuthHandler.java`-Klasse, die über folgende Methode verfügt:

```
boolean allowAuthentication(String base64UserCredentials)
```

Der Authorization Header kann über das Request-Objekt entnommen werden:

```
String authHeader = request.headers("Authorization");
// authHeader = "Basic dnNpczpwYXNzd29yZA=="
```

Diesen String übergeben Sie an die AuthHandler-Klasse um ihn zu **zerlegen**, zu **dekodieren** und zu **verifizieren**.

Geeignete Methoden sind u.a.:

```
String.split(String regex)
Base64.getDecoder().decode(String base64encodedString)
```

Sollten der Username und das Passwort mit einem in der Datenbank hinterlegten übereinstimmen, muss nichts weiter getan werden. Sollte dies nicht der Fall sein beenden Sie die `Spark.before(...)` Methode mit:

```
if(!authenticated)
{
    Spark.halt(401, "Unauthorized");
}
```

Genauso sollte der `halt`-Befehl auch ausgeführt werden, wenn gar keine Username/Password Kombination verwendet wurden.

Wenn Sie jetzt die Android-App testen, werden Sie vermutlich im Log eine NullPointerException erhalten, denn `response.body()` liefert jetzt kein Objekt mehr zurück. Um zu überprüfen ob es sich dabei um einen Authentifizierungsfehler handelt können Sie am `response` Objekt den HTTP-Code[7] mittels `code()` abfragen. Der normale „OK“-Code ist 200, während „Unauthorized“ der Code 401 (s.o.) ist. Theoretisch können Sie eigene Status-Codes verwenden, davon ist allerdings abzuraten.

[7] Liste der HTTP-Codes: https://de.wikipedia.org/wiki/HTTP-Statuscode#Liste_der_HTTP-Statuscodes

5.2.1 Retrofit nachrüsten

Um dem Retrofit-Interface die Möglichkeit zu geben, sich mit dem Server zu authentifizieren, müssen Sie einige Anpassungen an der `APIFactory` vornehmen.

- Legen Sie ein neues Feld (Exemplarvariable) in der `APIFactory` vom Typ `OkHttpClient` an. Initialisieren Sie es im privaten Konstruktor der Klasse und fügen es in die „Build-Reihe" des `Retrofit`-Objektes ein:

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl(API_URL)
        .addConverterFactory(GsonConverterFactory.create())
        .client(okclient) // initialisierter OkHttpClient
        .build();
```

- Fügen Sie eine neue Klassenmethode zur `APIFactory` hinzu, die Username und Passwort als String entgegennehmen, z.B.:

```
setAuthCredentials(String usr, String pwd)
```

- Generieren Sie in dieser Methode den BasicAuth Header aus dem Username und Passwort. Dazu wiederholen Sie die Schritte, die Sie auf der Serverseite tätigten, in umgekehrter Reihenfolge.
- Fügen Sie dann dem OkHttpClient einen neuen Interceptor hinzu, der den Authorization Header ergänzt:

```
singleton.okclient.interceptors().clear();
singleton.okclient.interceptors().add(new Interceptor()
{
    public com.squareup.okhttp.Response intercept(Chain chain) throws
                                                              IOException
    {
        Request original = chain.request();
        Request.Builder requestBuilder = original.newBuilder()
                .header("Authorization", basicAuthHeaderString)
                .header("Accept", "applicaton/json")
                .method(original.method(), original.body());
        Request request = requestBuilder.build();
        return chain.proceed(request);
    }
});
```

- Wenn der RestAPI-Aufruf aufgrund von `response.code() == 401` fehlschlägt, setzen Sie Username/Passwort und führen den Aufruf erneut durch.
- Sollten Username/Passwort bereits bekannt sein, setzten Sie sie bereits vor dem ersten Aufruf.