

## Deckblatt

# Inhaltsverzeichnis

- 1 Die Idee hinter der App
- 2 Vorteile der App gegenüber herkömmlicher Methoden
- 3 Die App und ihre Zustände
- 4 Die Funktionen der App und deren Umsetzung
  - 4.1 Login und Registrierung
  - 4.2 Das Steckbriefkonzept
  - 4.3 Das Freunde-System und die Chat-Funktion
- 5 Activity-Stack
- 6 Aufbau des Servers
  - 6.1 Datenbank
    - 6.1.1 Struktur der Datenbank
    - 6.1.2 PreparedStatements vs. Statements
    - 6.1.3 Umsetzung Datenzugriffs
  - 6.2 User Interface des Servers
- 7 Server-Client Kommunikation
  - 7.1 Client
  - 7.2 Server
- 8 Ausblick in die Zukunft
- 9 Schluss
- 10 Danksagung
- 11 Verwendete Programme
- 12 Literaturverzeichnis
- 13 Eidesstattliche Erklärung

## **1. Die Idee hinter der App**

Ein großer Teil der Jugendlichen hat ab dem 14ten Lebensjahr schon einmal, zumindest einen, Anfängerkurs für Standardtänze belegt. Bevor das Belegen eines solchen Kurses jedoch möglich ist, muss erst ein passender Tanzpartner gefunden werden. Die Vermittlung kann z.B. über den Ausbildungsplatz wie z.B. Gymnasien, Realschulen oder Unis organisiert sein. Die Tanzpartnersuche über den Freundeskreis ist auch sehr weit verbreitet. Für Jugendliche, bei denen genannte Methoden entweder nicht zur Verfügung standen oder erfolglos waren, insbesondere infolge einer Knappheit an tanzwilligen Jungen, gab es an der Tanzschule, an der ich einst das Tanzen lernte, eine weitere Vermittlungsmethode.

Über eine Pinnwand mit öffentlich aushängenden Steckbriefen konnten die Suchenden Kontaktdaten austauschen. Das Ziel meiner Tanzschul App ist es die Funktionen dieser Pinnwand in eine Server-Client basierende App zu integrieren, diese zu erweitern und neue Features hinzuzufügen. Wobei der Server die Pinnwand darstellt, auf der die Steckbriefe der User gespeichert werden. Die App ist kombiniert mit einem User-System, damit Nutzer ihre Steckbriefe leichter anpassen, löschen oder erstellen können. Der Client kann dann die gespeicherten Steckbriefe vom Server abrufen und danach auslesen. Oder auch Änderungen an den Steckbriefen vornehmen indem er dem Server neue Daten sendet.

## **2. Vorteile der App gegenüber herkömmlicher Methoden**

Die Tanzschul App übernimmt nicht nur das Prinzip einer wie vorhin beschrieben Pinnwand, sondern bringt auch viele Vorteile und Verbesserungen mit sich. Dank eines Servers werden alle wichtigen Daten, welche die App zum Funktionieren benötigt online abgespeichert. Das führt dazu, dass die Applikation jederzeit und von überall verwendet werden kann. Steckbriefe können schnell und bequem von unterwegs erstellt oder bei Bedarf angepasst werden. Zudem können Änderungen an der „Pinnwand“ des Servers jederzeit und von überall mithilfe der App abgerufen werden.

Alle wichtigen Daten werden serverseitig in einer SQL Datenbank abgespeichert. Dies ermöglicht praktisches Filtern der zum Client gesendeten Steckbriefe nach den Wünschen des Users. Zudem ist bei meiner App eine erleichterte

Kontaktaufnahme gegeben. Rufnummern anderer User können im Geräte eigenen Telefonbuch abgespeichert oder in die Zwischenablage kopiert werden. Für Nutzer, die lieber eine Nachricht hinterlassen wollen implementiert die App einen eigenen Chat.

### 3. Die App und ihre Zustände

Die Tanzschul App weist für jede ihre Zustände eine eigene Activity auf. Diese sind dafür zuständig die Benutzeroberfläche zu definieren und auf Benutzer Interaktionen wie z.B. durch Buttons zu hören und die vom User empfangen Informationen zu verarbeiten.<sup>1</sup>

Ein Konstrukt aus mehreren Activities hat viele Vorteile gegenüber einem Neuzeichnen der [GUI](#) (Graphical User Interface) innerhalb einer einzigen [Activity-Klasse](#), für jeden Wechsel des Zustands. Bei letzterem müssen nämlich mehrere Layouts innerhalb einer Klasse implementiert werden. Das führt dazu, dass der Code insgesamt komplizierter gestaltet ist, um die verschiedenen Zustände der App umzusetzen und voneinander abzugrenzen zu können. Beispielsweise muss bei jedem Zustandswechsel unnötiger Ressourcenspeicher freigegeben, wie z.B. der von nicht mehr verwendeten Attributen. Des Weiteren führte die Umsetzung von mehreren Benutzeroberflächen in einer Klasse zu einem unnötig langen Code und würde somit die Verständlichkeit und Übersichtlichkeit dessen stark verschlechtern.

Zu guter Letzt müssten bei der neuzeichnenden Variante sehr viele Attribute, die nur in bestimmten Zuständen der App überhaupt Verwendung finden, in der Activity-Klasse und so im Arbeitsspeicher des Geräts abgespeichert werden.

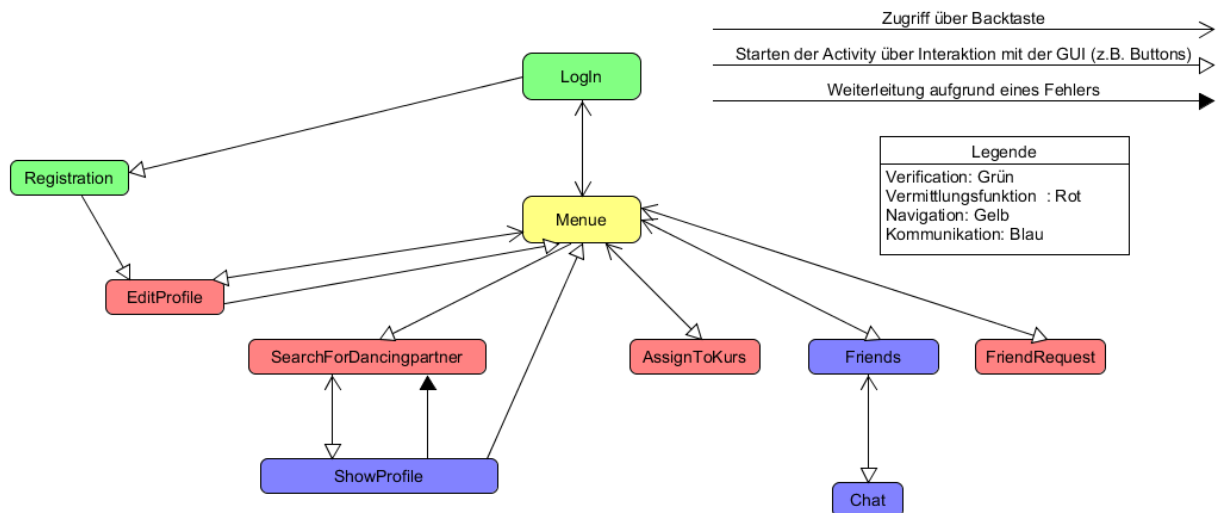
Beim Verwenden mehrerer Activitys ist die Abgrenzung untereinander deutlich einfacher. Hier kann nämlich der vordefinierte [Lebenszyklus](#) einer Activity verwendet werden. Dieser bringt Methoden, wie *onPause*, *onStop* oder *onDestroy* mit sich, in denen nicht mehr verwendete Ressourcen freigegeben werden können. Jeweils eine dieser Methode wird aufgerufen, je nachdem ob die Activity neu gezeichnet, temporär in den Hintergrund verschoben oder beendet wird. Beim

---

<sup>1</sup> Vgl. ADROID, *Activity*

Starten einer neuen Activity können wichtige Informationen über Intents weitergegeben werden.

Informationen die auch nach dem Beenden der App vorhanden sein sollen, können extern z.B. in sog. [SharedPreferences](#) gespeichert werden.



Diese Grafik zeigt die in der Tanzschul App vorhandenen Activities und so auch alle Zustände. Die Pfeile veranschaulichen, durch welche Art von User Interaktion der Zustandswechsel ausgelöst wurde und somit eine neue Activity in den Vordergrund rückt. Alle grün markierten Activities dienen zur Nutzer Verifikation. Jeder User hat eine individuelle ID, die nur er selbst kennt. Diese kann er bei den grün markierten Activities zugesandt bekommen. Die gelb markierte Activity dient zur Navigation zwischen den unterschiedlichen Funktionen der App. Die rot markierten Activities sind sozusagen das Herz der App, sie implementieren die Funktionen, die zur eigentlichen Vermittlung beitragen. Zur direkten oder indirekten Interaktion zwischen Usern dienen die blau gefärbten Activities.

## 4. Die Funktionen der App und ihre Umsetzung

Im Vorausgehenden Kapitel 3 bis 3.1 wurde die strukturelle Anordnung der Activities veranschaulicht. In diesem Teil wird auf die Aufgaben und Entstehungsgedanken, welche den einzelnen Activities zu Grunde liegen, eingegangen. Zu aller erst einmal zur so genannten *ConnectedActivity*. Sie ist eine selbstgeschriebene abstrakte

Klasse, die von allen *Activity*s erweitert wird, die auf Internetressourcen des Endgeräts zugreifen, beziehungsweise den Internetzugriff überprüfen.

Die Methoden der *ConnectedActivity* sollen die Implementierung der Server-Client Kommunikation erleichtern, dazu zählt ein einheitliches Fehlerbehandlungssystem.

Nun noch genaueres zur Fehler Behandlung bevor wir zu den eigentliche *Activity*s kommen Es gibt im Großen und Ganzen zwei Typen von Fehlern, die in meinem Konzept erfasst werden können. Der erste Typ sind Verbindungsfehler, welche durch eine nicht vorhandene oder zu langsame Internetverbindung des Geräts oder durch ein Nichtantworten des Servers ausgelöst werden können. Bei Verbindungsfehlern wird die Internetverbindung des Endgeräts geprüft und eine passende Fehlermeldung ausgelesen, je nachdem, ob die *Activity* generell auf eine Internetressource zugreifen kann oder nicht.

Der Zweite Typ sind Fehler Codes. Diese kennt sowohl der Server als auch der Client. Es gibt bis jetzt drei verschiedene Arten von Fehlermeldungen, die vom Server empfangen werden können. Diese sind: *nf* (*not found*), *ae* (*already exists*), *wl* (*wrong login*) und *ja* (*alles ist korrekt abgelaufen*). Tritt im Server ein bestimmter Fehler auf, wird der zum Fehler passende Fehlercode zum Client gesandt, ist alles richtig abgelaufen, lautet der Fehlercode „*ja*“. Ist der vom Server erhaltene Fehlercode auf einen Eingabefehler zurück zu führen, wird eine passende Fehlermeldung ausgelesen, das ist bei *nf* und *ae* meist der Fall.

Der Fehlercode *wl* wird vom Client erhalten, falls die vom User im Request mitgesendete Identifikationsnummer nicht in der Datenbank des Servers existiert. Erhält der Client den Fehlercode *wl* (*wronglogin*), wird zur Login-*Activity* zurückgekehrt, um mögliche Folgefehler oder unautorisierte Zugriffe auf Daten des Servers zu vermeiden.

Zum jetzigen Zeitpunkt steht der Fehlercode *nf* sowohl für eine Zugriffsanfrage auf ein nicht existierendes Objekt in der Datenbank, als auch für das Auftreten eines unbekannten Fehlers. In Zukunft könnte man diese beiden Arten von Fehlern, für die *nf* steht, trennen und einen weiteren Fehlercode, wie z.B. *fe* für fatal error hinzufügen.

Anfangs bestand die Idee, sowohl die Behandlung von Verbindungsfehlern als auch die von eingabespezifischen Fehlern in einer Methode umzusetzen. Jedoch wird die Methode `onConnectionError()`, die für das Behandeln von Verbindungsfehlern zuständig ist, meist von den Kinderklassen übernommen, wohingegen die `onError()` Methode in einigen Fällen zur individuellen Reaktion der Activity auf Fehlercodes überschrieben werden muss.

## 4.1 Login und Registrierung

Im ersten Konzept meiner App war keine Login-Funktion oder gar ein Account-System geplant. Schnell wurde jedoch klar, dass es ohne ein solches schwierig werden würde, Funktionen wie einen Chat zu implementieren oder die „Pinnwand“ vor Spam zu schützen.

Auf Basis dieser Überlegungen entstand die Login-Activity, mit dieser kann der Nutzer seine individuelle Identifikationsnummer (ID) im Austausch seiner passenden Kombination von E-Mail und Passwort vom Server abrufen. Die ID muss bei jedem Request zur Identifikation des Clients an den Server gesandt werden. Nutzer, die noch keinen Account besitzen, werden über den „Registrieren“-Button zur Registration-Activity weitergeleitet.

Besonders wichtig war es hier, den Login für den User bequem zu gestalten, deshalb wurde eine Funktion hinzugefügt, mit der man E-Mail und Kennwort bis zur Deinstallation der App speichern kann. Um das zu realisieren, müssen die beiden Parameter extern gespeichert werden, da nach dem Schließen der App alle nicht gespeicherten Informationen verloren gehen.

Dabei bieten sich mehrere Möglichkeiten, beispielsweise könnte man eine eigene SQLite Datenbank anlegen um die Werte zu speichern. Das würde jedoch einen unnötigen Programmier -und Rechenaufwand bedeuten. Für das Speichern einer kleinen Anzahl von sog. „key-values“ bieten sich SharedPreferences an, welche genau dafür ausgelegt sind wenige Datensätze primitiver Datentypen oder Strings in Datenpaaren zu speichern.<sup>2</sup> Möchte man komplexe Daten speichern, ist eine SQLite Datenbank besser geeignet.<sup>3</sup> Das Speichern komplexerer Daten, sollte

---

<sup>2</sup> Vgl. RACHITA NANDA

<sup>3</sup> Vgl. ANDROID, *Saving Key-Value Sets*

vorzugsweise in der *onStop()* und nicht in der *onPause()* Methode vorgenommen werden, da sonst durch rechenaufwendige Speicherprozesse das Starten einer neuen Activity verzögert werden kann.<sup>4</sup>

Kommen wir zur Registration-Activity. Der schwierigste Teil des Codes liegt darin, die vom Nutzer eingegebenen Daten auf syntaktische Richtigkeit zu überprüfen und passende Fehlermeldungen auszulesen. Letztendlich wird die Überprüfung mit einem *boolean array* umgesetzt, welches für jede zu erfüllende Bedingung ein Item aufweist. Nur wenn alle Items des Arrays den Wert *true* haben wird dem Server eine Anfrage zum Erstellen eines neuen Accounts zugesandt.

## 4.2 Das Steckbriefkonzept

Jeder Nutzer besitzt ein Profil, dieses enthält alle persönlichen Daten, die über den Nutzer gespeichert werden. Es kann jederzeit über die EditProfile-Activity angepasst werden. Über die SearchForDancinpartner-Activity kann der Nutzer die Steckbriefe nach Kursstufe und dem Wochentag, an dem der gewünschte Tanzkurs stattfinden soll, filtern. Die Stufenbezeichnungen der Grundkurse sind von der Tanzschule Fischer Ingolstadt übernommen worden. Die höheren Stufen entsprechen dem einheitlichen Medaillensystem der [ADTV](#) geprüften Tanzschulen. Der Wochentag bezeichnet den Tag, an dem die wöchentlich stattfindenden Kurse gehalten werden.

Die Steckbriefe (nur die des anderen Geschlechts) anderer User werden vom Server abgerufen und in einer ListView angezeigt. Klickt man auf eine der ListView-Items, kann man das Profil des Nutzers, der diesen Steckbrief erstellt hat, einsehen. Möchte man Steckbriefe von sich selbst für bestimmte Kurse hinzufügen oder löschen, kann man das in der AssignToKurs-Activity tun.

## 4.3 Das Freunde-System und die Chatfunktion

Für den erfolgreichen Abschluss der App, war es mir wichtig, dass die Kontaktaufnahme zwischen den Nutzer durch die App erleichtert wird. Ein App eigener Chat, durfte dafür nicht fehlen. Die Implementierung eines solchen Chats ist jedoch deutlich komplizierter als anfangs erwartet. Der Chat der Tanzschul App

---

<sup>4</sup> Vgl. ANDROID, *Pause Your Activity*



sollte nämlich kein einfacher Gruppenchat zwischen allen Usern sein, sondern eine private Kommunikationsmöglichkeit zwischen zwei Usern bieten. Ein solcher Chat soll jedoch nur bestehen, wenn beide User dessen Entstehung zugestimmt haben. Um dies umzusetzen, wurde ein Freunde System, nach diesen Anforderungen, geschaffen.

Im Server werden alle Beziehungen zwischen den Nutzern gespeichert, der Client speichert diese nur temporär. Nachdem die Friend-Activity den Fokus verloren hat, werden die Daten beim nächsten Fokus der Activity neu vom Server abgerufen. So bleiben die Beziehungen zwischen den Usern automatisch auch im Client aktuell, ohne Verfahren wie Polling umsetzen zu müssen.

Der Chat ist in erste Linie zum Austausch weiterer Kontaktdaten und nicht als Instant Messenger gedacht. Deshalb ist es zu diesem Entwicklungsstand nicht möglich andere Nutzer über ihren Namen zu suchen und so zu den Freunden hinzuzufügen. Die Freundschaftsanfrage ist stattdessen eine der Möglichkeiten zur Kontaktaufnahme in der ShowProfile-Activity, auf die man über das Klicken auf einen der Steckbriefe in der SearchForDancingpartner-Activity gelangen kann.

Freundschaftsanfragen können in der FriendRequest-Activity vom User angenommen werden. Tut er das, wird dem Server ein Request zugesandt, der ihn dazu auffordert, ein neues Freundschaftsobjekt und einen Chat zwischen dem Sender des Requests und dem Ersteller der Freundschaftsanfrage in der SQLite Datenbank zu erstellen. Freunde können bis zum jetzigen Stand nicht entfernt werden, solch eine Funktion wäre jedoch für die Zukunft durchaus denkbar.

Kommen wir nun aber zum eigentlichen Chat. In der Friends-Activity werden dem User alle bestehenden Chats bzw. Freunde in einer ListView angezeigt. Ausstehende Freundschaftsanfragen, die der User selbst versendet hat, sind in derselben einzusehen. Klickt man nun auf eine der bestehenden Freundschaften (Achtung Bug, nicht akzeptierte Freundschaften können auch angeklickt werden) wird man zu einem Chatfenster weitergeleitet. Das Updaten, der im Chat versendeten Nachrichten, ist mithilfe von Polling umgesetzt. Dabei hat jedes Nachrichten-Objekt ein individuelles Integer Attribut *id*, das ihm im Server zugewiesen wird. Um nun die fehlenden Nachrichten vom Server anzufordern wird die *id* des letzten im Client gespeicherten Nachrichten-Objekts an den Server

gesandt. Der Server antwortet mit einer Liste aller Nachrichten-Objekte, deren *id* Attributwert höher als der vom Client gesendete Wert ist. Ist die gesendete *id* bereits die höchste in dem vom Server angeforderten Chat, wird eine leere Liste versandt.

Polling allgemein steht für ein zyklisches Abrufen einer bestimmten Information. Wenn sich diese ändert, kann man relativ zeitnah die abgerufene Änderung übermitteln und verarbeiten. Der maximale Delay ist dabei insbesondere von der Zeitspanne, in der der Zyklus ausgeführt wird, abhängig.

Es gibt mehrere Arten des Pollens darunter short-polling und long-polling. Beim short-polling wird in einem konstanten Zeitabstand zyklisch ein Request an den Server gesandt. Dieser antwortet gleich nach dem Erhalten des Requests mit einer Response. Nachdem der Client die Response erhält gilt der Request als beendet. Dieses Verfahren ist besonders günstig um Daten abzurufen, die sich in einem festgelegten Zyklus erneuern wie z.B. eine Datenbank in der alle 2 Minuten vom Client empfangene Änderungen in die Datenbank aufgenommen werden. Diese Änderungen könnten dann von den Clients effizient zyklisch durch short-polling abgerufen werden. Ein Nachteil an diesem Prinzip ist, dass falls die Daten nicht zyklisch in der Datenbank erneuert werden, viele Requests an den Server eventuell ohne direkten Nutzen sind, da es keine neuen Daten zu übermitteln gibt. Bei einem Chat zwischen mehreren Nutzern trifft dieser Nachteil offensichtlich zu.

Eine andere Methode ist long-polling bei diesem Prinzip wird nach dem Erhalt des Requests serverseitig die gewünschte Information zyklisch abgerufen. Sobald die abgerufene Information nicht mehr mit dem Stand des Clients übereinstimmt, wird der Zyklus gestoppt und die neue Information an den Client weitergeleitet.<sup>5</sup> Wie auch beim short-polling, kann hier das zyklische abrufen der Information aus der Datenbank unnötig teuer sein. Im Vergleich zum short-polling trägt hier jedoch nur der Server mehr Rechenlast als der Client, da das Polling serverseitig stattfindet.

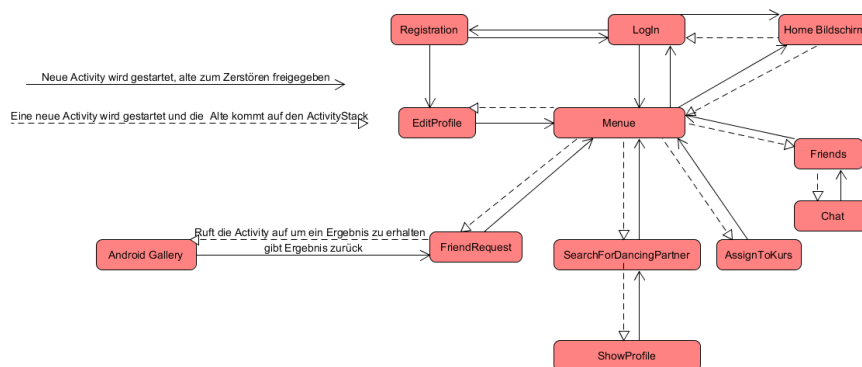
Für die Anwendung in einem Chat, sind beide Methoden durchaus denkbar, short-polling wurde letztendlich favorisiert, da dessen Umsetzung einfacher und für den Server, in Bezug auf meine Umsetzung des Chats, potenziell weniger rechenaufwendig ist.

---

<sup>5</sup> Vgl. KAZING

Short-polling ist hier um einiges effektiver, da dabei nur während die Chat-Activity im Vordergrund ist überhaupt gepollt wird. Würde im Server für eine feste Zeit gepollt werden, hat der Nutzer wohlmöglich die Chat Activity bereits verlassen, so dass ihn die abgerufenen Daten nichtmehr erreichen können. Eine Alternative zum Pollen ist die Umsetzung mit einem Websocket. Eben solche wäre hier jedoch unnötig kompliziert, da der Chat keine Haupt, sondern allein eine Nebenfunktion der App ist.

## 5. Activity-Stack



„A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a stack (the *back stack*), in the order in which each activity is opened.“ (Android Open Source Project, 2016)<sup>6</sup> Dieses Zitat beschreibt in kurzen Worten, was ein *back stack* ist, und die Activities im LiFo (Last in First out) abgelegt werden. Jedes Mal, wenn eine Activity eine andere aufruft, rückt die aufrufende Activity in den Hintergrund und wird auf den *back stack* gelegt.<sup>7</sup> Wenn der Nutzer die Zurücktaste an seinem Endgerät betätigt, wird *onBackPressed()* aufgerufen. Diese Methode gibt genauso wie *finish()* die aktuelle Activity zum Zerstören frei. Sie wird jedoch nur zerstört, wenn das System Ressourcen für andere Rechengvorgänge frei machen muss. Wird *onBackPressed()* oder *finish()* aufgerufen, erhält die letzte Activity auf dem ActivityStack den Fokus insofern keine andere Activity den Fokus besitzt. Das wäre z.B. der Fall, wenn in

<sup>6</sup> Vgl. ANDROID, *Tasks and Back Stack*

<sup>7</sup> Vgl. ANDROID, *Tasks and Back Stack*

*onBackPressed()* oder vor dem Aufrufen von *finish()* eine neue Activity gestartet wurde.<sup>8</sup>

Um Zyklische Aufrufe zu vermeiden, muss sichergestellt werden, dass keine der Activitys mehrfach aufgerufen werden kann oder ungewollt auf dem ActivityStack verbleibt. Das kann mit sog. Flags erreicht werden. Mit diesen können die Threads, in denen die Activitys laufen verwaltet und der Stack jedes einzelnen Threads angepasst werden.<sup>9</sup> Die Tanzschul App besitzt kein kompliziertes System aus verschiedenen Threads für die unterschiedlichen Activitys. Hier laufen alle Activitys in einem Thread, nur Aufgaben, wie das Versenden von Requests, werden in separaten Threads abgehandelt. Flags finden in der App zwar kaum Anwendung, trotzdem wird dafür gesorgt, dass keine Zyklen entstehen.

Wie in der Abbildung zu sehen ist, wird das Entstehen von Zyklen dadurch verhindert, indem in Activitys, die nicht auf den Back Stack gelangen sollen, die *finish()* Methode aufgerufen wird. In dem Diagramm ist jedoch nicht berücksichtigt, ob beim Starten einer neuen Activity tatsächlich eine neue Activity gestartet wird oder eine „alte“ vom Back Stack genommen wird.

Das ist jedoch irrelevant, da nur die Menue-Activity länger im *back stack* verbleibt. Die Menue-Activity wird nur beendet, wenn der User sich ausloggt. Oder wenn der *back stack* vom System, zur Ressourcengewinnung, gelöscht wird, da die App im Hintergrund läuft. Alle anderen Activitys werden, wie in der Zeichnung zu erkennen, Pfad ähnlich durchschritten. Der User kann zwar an jeder Stelle des Pfads umkehren ihn jedoch nicht verlassen. So dass nur die Activitys, die hinter der Position des Nutzers liegen im Activity Stack abgelegt sind. Geht der Nutzer zurück, wird die Activity die er grade verlässt zum Zerstören freigegeben und die letzte im Activity Stack aufgerufen, bis er wieder in der Menue-Activity angekommen ist. Geht der User nach „vorne“ wird die Activity in der er sich befindet auf den Activity Stack gelegt und die darauffolgende im Pfad gestartet.

Flags finden trotzdem an manche Stellung Anwendung z.B. beim Auftreten eines Verbindungsfehlers wird die Methode *setFlags(flag)* verwendet um die Menue-Activity in einem Neuen Thread zu starten und den alten Activity Stack zu räumen.

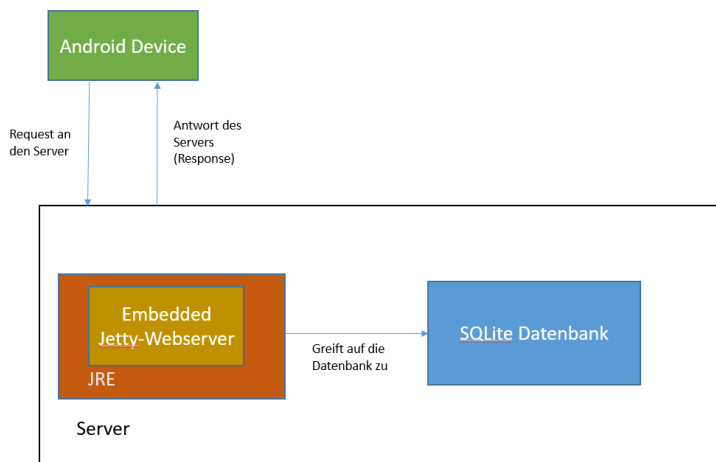
---

<sup>8</sup> Vgl. ANDROID, *Activity/ Shutting down an Activity, Managing the Activity Lifecycle*

<sup>9</sup> Vgl. ANDROID, *Tasks and Back Stack / Managing Tasks*

Hier müssen Flaggs verwendet werden, da es, mit der vorhin beschriebenen Methode, unnötig umständlich ist, mehr als einen Schritt zurück zu gehen.

## 6 Aufbau des Servers



Der in Java geschriebene Server ist mindestens genauso wichtig wie der Client. Mithilfe von JDBC kann er auf eine SQLite Datenbank zugreifen, in dieser werden alle wichtigen Daten, von den Nutzern gesendeten Daten, gespeichert. Um http-Requests empfangen und mit http-Responses antworten zu können verwendet der Server das Jetty-Framework. Dieses zeichnet sich dadurch aus, dass es mit Leichtigkeit in ein Framework wie eine JRE (Java Runtime Environment) integrieren lässt. Jetty wurde dazu entworfen, um in eine Anwendung eingebettet zu werden und nicht um alleinstehend zu laufen, wie untenstehendes Zitat zeigt.

*"Jetty has a slogan, "Don't deploy your application in Jetty, deploy Jetty in your application." (Eclipse, 2016)<sup>10</sup>*

Der Server der Tanzschul App wurde auf Basis von im Unterricht besprochenen Inhalten aufgesetzt und weiterentwickelt. Das Diagramm ist ebenso dem im Unterricht gezeigten Material nachempfunden.

### 6.1 Datenbank

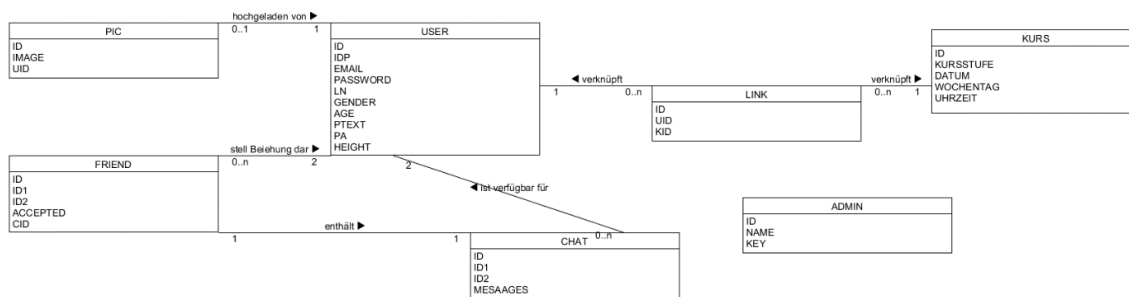
Zum Lesen und Schreiben in eine SQLite Datenbankdatei wird die JDBC Bibliothek verwendet.<sup>11</sup> Mithilfe der JDBC können SQL Statements formuliert werden über die,

<sup>10</sup> Vgl. ECLIPSE, *Jetty/Tutorial/Embedding Jetty*

<sup>11</sup> Vgl. GITHUB, *SQLite JDBC Driver*

der Programmierer die Datenbank manipulieren oder auch komplett neu erstellen kann. In Android ist die Verwendung von SQLite auch möglich und schon in das Betriebssystem eingebettet, so dass keine API wie JDBC zusätzlich von Nöten ist. Auch der Client besitzt eine SQLite Datenbank diese ist jedoch erst teilweise Umgesetzt. Die Idee für die clientseitige Datenbank stammt noch aus den ersten Konzepten für die App. Daher war es nur naheliegend, dass auch im Server eine SQLite Datenbank anstatt eines SQL Servers verwendet wurde, da Hoffnung auf eine leichtere Interaktion zwischen Client und Server bestand.

### 6.1.1 Struktur der Datenbank



Die serverseitige Datenbank hat insgesamt 7 verschiedene Tabellen. In diesem Teil wird genauer auf die Funktionen und die Zusammenhänge zwischen den einzelnen Tabellen eingegangen.

Die User Tabelle, war die erste in der Datenbank. Sie speichert persönliche Daten, die der User dem Server bei der Registration sendet. Der Nutzer kann die an den Server gesendeten Informationen im Nachhinein über Requests anpassen. Seine Login Daten, E-Mail und Passwort, sind hier ebenfalls abgespeichert, zum jetzigen Zeitpunkt aber nicht anpassbar. Eine neue Zeile der Tabelle User wird dann erstellt, wenn ein neues Benutzerkonto angelegt wird. Das erste Konzept der Datenbank, sah eine Trennung zwischen den persönlichen und den Login Daten vor, um bessere Sicherheitsvorkehrungen für das Abrufen letzterer festlegen zu können. Im Laufe der Umsetzung wurde jedoch schnell klar, dass simple Abfragen unnötig lang und kompliziert wurden, da man bei fast jeder Abfrage, die persönliche Daten ausliefert, automatisch die Login Tabelle joinen musste. Die Spalte ID ist der Primär Schlüssel der Tabelle User und kann nur von dem Client abgerufen werden, der im Request die passende Kombination von E-Mail und Kennwort für diese User Zeile

an den Server sendet. ID und IDP verweisen beide auf den gleichen User, sind für jeden User einzigartig und werden mithilfe eines *SecureRandom* erzeugt.

Die IDP unterscheidet sich von der ID, dass sie verwendet wird um den User für andere Nutzer auszuweisen und dass die IDP jeder Zeile von allen Clients abgerufen werden kann, insofern der Nutzer bereits eingeloggt ist. Durch diese Trennung, darf nur der Nutzer selbst seine wirkliche ID erfahren. Die KURS Tabelle speichert alle Kurse, welche die Tanzschule zurzeit unterrichtet. Sendet der User einen Request an den Server, dass für ihn ein neuer Steckbrief erstellt werden soll, wird ein neues Objekt in der LINK Tabelle erstellt. Die Tabelle Friend stellt eine Freundbeziehung zwischen zwei Usern dar, die Spalte ACCEPTED gibt an ob die Verbindung der zwischen den beiden Usern von beiden akzeptiert wurde. Ein User kann unbegrenzt vielen Usern die Freundschaft erklären. Wird der Wert der ACCEPTED Spalte in irgendeiner Zeile von 0 auf 1 geändert, wird für diese Zeile ein neues Objekt in der Tabelle CHAT erstellt und in der Friend Tabelle verweist der Passende Wert in der CID Spalte nun auf den neuen Chat.

Ein Objekt der in der Tabelle ADMIN kann nur vom Server aus erstellt werden. Diese Tabelle speichert die Admin Login Daten, über welche die Kurse mithilfe der Admin App angepasst werden können. Für die Tabelle PIC sind auf Grund niedriger Priorität noch nicht alle wichtigen Methoden implementiert. Sie soll in Zukunft dazu dienen Profilbilder der User abzuspeichern.

### 6.1.2 Prepared Statements und Statements

In Java kann mit Hilfe von JDBC über drei verschiedene Arten eine Datenbankmanipulation angefragt werden, über das normale Statement, das Prepared Statements und das Callable Statement<sup>12</sup> Dieses Statement ist in der Datenbank selbst gespeichert und kann mithilfe des JDBC aufgerufen werden. Das Callable Statement kann bei sehr großen Rechenaufgaben einen Vorteil bringen, da sie im Ressourcenpool des Datenbank Servers ausgeführt wird.<sup>13</sup> Da die Datenbank der App jedoch keine riesigen Datenmengen aufweist, finden Callable Statements hier keine Anwendung.

---

<sup>12</sup> JAVA IST AUCH EINE INSEL, *Abfragen über das SQL Statement*

<sup>13</sup> Vgl. JENKOV

Bei normalen Statements wird die Abfrage Syntax zusammen mit den Variablen kompiliert und an die Datenbank gesendet, es muss zudem bei jedem Aufruf neu kompiliert werden. Beim Prepared Statement hingegen wird zuerst die Syntax vorbereitet und temporär gespeichert, nun kann das Prepared Statement beliebig oft ausgeführt werden, indem bei jedem Ausführen neue Variablen in das Statement eingesetzt werden.

Das normale Statement ist dem Prepared Statement bei einmaliger Ausführung performancetechnisch überlegen, da letzteres zweimal kompiliert werden muss, weil es in zwei Teilen an die Datenbank gesendet wird. Bei wiederholtem Ausführen liegt jedoch das Prepared Statement vorne, da die Syntax des Statements nach dem ersten Ausführen bereits kompiliert ist und nur noch die fehlenden Variablen eingefügt werden müssen.

Das normale Statement ist anfällig für sogenannten SQL Injections, denn hier liegt nahezu keine Trennung zwischen den Abfrage Befehlen und den Variablen vor. Dies führt dazu, dass SQL Abfragen durch Nutzer umgeschrieben werden können, indem sie die Variable escapen. Da beim Prepared Statement die Befehle und die Variablen getrennt behandelt werden, sind Änderungen an dem Statement durch Nutzervariablen nicht möglich.<sup>14</sup>

### 6.1.3 Umsetzung des Datenzugriffs

Die Klasse *Model* besitzt eine Instanz von sich selbst als Attribut, welche sie, insofern sie nicht bereits existiert, im statischen Konstruktor instanziiert. Diese Klasse dient dazu, auf Methoden der DAO Klasse *TableManager* zuzugreifen und deren Ergebnisse zu verarbeiten. Das Modell kommt dann ins Spiel, wenn Abfragen an die Datenbank oder die Manipulation dieser gefragt sind. Das heißt z.B. wenn auf Eingaben über die Server GUI oder auf einen Request reagiert werden muss.

Eigentlich ist es üblich, für jede Tabelle eine eigene DAO Klasse zu haben. Der Server der Tanzschul App folgt diesem Ideal jedoch nicht. Er besitzt nur eine DAO Klasse, welche Methoden zur Interaktion mit der Datenbank implementiert. Es gibt

---

<sup>14</sup> STACK OVERFLOW, *Difference between Statement and Prepared Statement*



jedoch für jede Tabelle in der Datenbank eine Art DAO Klasse, in deren Konstruktor wird getestet ob eine Tabelle existiert, wenn nicht wird sie in der Datenbank erstellt.

Ein Grund für die Zentrierung der Methoden ist, dass sich viele davon nicht direkt einer Tabelle zuordnen lassen, da bei der Abfrage zwei oder drei verschiedene miteinander vereinigt werden oder weil mehrere verschiedene Aktionen vorgenommen werden. Dazu können die Methoden über Stichwortsuche leichter gefunden werden, wenn sie sich alle in einer Klasse befinden, da Eclipse keine projektweite Suche unterstützt.

Diese Gründe waren während dem Programmieren besonders ausschlaggebend für das Design der DAOs. Es würde jedoch durchaus Sinn machen das Prinzip der Tabellen bezogenen DAOs für eine Veröffentlichung, durch eine Tanzschule, trotzdem umzusetzen, da es anderen Wartungsprogrammierern wahrscheinlich einfacher fallen würde, mit einem bereits bekannten Konzept zu arbeiten.

## 7. Server-Client Kommunikation

Es wurde bereits sowohl auf die Struktur des Servers als auch auf die des Clients eingegangen, es fehlt nur noch der Datenaustausch zwischen den Beiden. Dieser ist auch der wichtigste Teil des Projekts, denn ohne die Informationen des Servers würde der Client nicht funktionieren. Das Grundprinzip der Server-Client Kommunikation ist relativ simpel. Der Client sendet einen Request an den Server, dieser verarbeitet ihn und sendet eine Antwort zurück an den Client. Im Folgenden wird dieser Vorgang genauer unter die Lupe genommen, und zur Kommunikation benötigte Datenstrukturen am Beispiel des RegisterRequest intensiver beleuchtet.

### 7. 1 Client

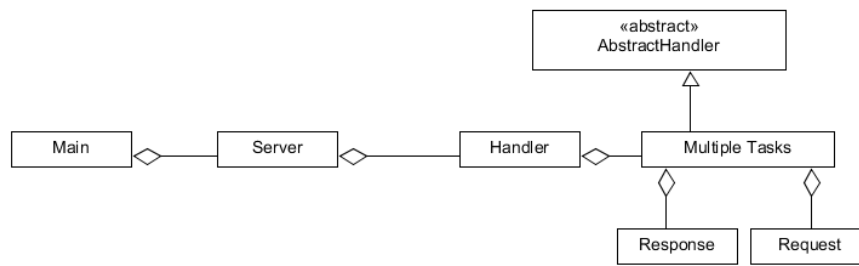
In dem Vorliegenden Konzept kann nur der Client einen Request versenden, welchen der Server mit einer Response beantwortet. Um einen LoginRequest zu versenden, muss der *RegisterTask* von der *Register-Activity* aus gestartet werden. Alle Task laufen in einem eigen Thread, denn sie erben alle von der *BaseHttpRequestTask* Klasse, welche ein Kind der Klasse *AsyncTask* ist. Im *RegisterTast* wird ein neues *RegisterRequest*-Objekt, das die vom Nutzer angegebenen Registrationsdaten beinhaltet erstellt. Aus dem Requestobjekt wird

nun ein XML-String gebaut. In der *doInBackground(String... params)* Methode des Tasks wird aus der Server URL, dem Einschub „?&do=“, dem Befehl „register“ und dem erwähnten XML-String ein Http-Request erstellt. Dieser wird an die im Request enthaltenen URL gesendet. Die statische URL ist in der Properties Klasse gespeichert. Alle Befehle, die an den Server gesandt werden können, sind in dem Enum Command gespeichert. Wurde der HttpRequest versandt, wartet der *RegisterTask* auf die Response des Servers. Insofern diese ankommt muss diese zu einem Responseobjekt geparkt werden, da sie im XML-Format vorliegt. Nun ist es wichtig, welcher Fehlercode in dem Responseobjekt enthalten ist. Er bestimmt, welche Aktion mit den mitgesendeten Daten vorgenommen wird. Im Fall des RegisterRequests, wird für den Fehlercode „ja“ die Registration-Activity verlassen und zum Zerstören freigegeben und die Menue-Activity mit der, in der Response enthaltenen ID als Intent-Extra gestartet.

## 7. 2 Server

Jeder Server läuft auf einer eigenen IP, diese kann dann vom Client über die Server URL kontaktiert werden. Die Main Methode, über die die JRE gestartet wird, besitzt ein JFrame und ein Objekt der Klasse *TServer*. Über die JFrame kann der Server auf Wunsch des Nutzers über Buttons gestoppt und gestartet werden. Die IP auf der der Server laufen soll ist auch über die JFrame anpassbar. Beim Aufrufen von *startServer()* wird in der *TServer* Klasse ein neuer Server der für die angegeben IP Adresse auf Port 8080 hört, gestartet. Dem Server wird ein Handler zugewiesen, der alle ankommenden *HttpServletRequest* verarbeitet. Nun kann der Server einen Request vom Client empfangen. Jeder Request hat einen Parameter „do“. Mit dessen Hilfe, weiß der Handler welchem Task er den XML Anhang des Request weiter geben soll. Auch der Server hat ein Enum *Command*, in welcher vordefiniert ist, welchen Wert „do“ einnehmen kann. Wobei der Server auch die Befehle der Admin App in dieser Klasse gespeichert hat. Im Falle des RegisterRequest wäre „do“ bei uns „register“. Der Request wird vom THandler an den RegisterTask weitergeleitet. In diesem wird der *httpBody*, der sich im XML Format befindet zu einem *RegisterRequest*-Objekt geparkt. War das Parsen erfolgreich, werden nun die vom Nutzer gesendeten persönlichen Daten aus dem *RegisterRequest*-Objekt ausgelesen. Diese werden nun an das Model weitergeleitet. Das Model weist den Table Manager dazu an eine neue Zeile in der Tabelle User zu erstellen, insofern

die mitgelieferte E-Mail nicht schon in der Datenbank vorliegt. Konnte erfolgreich eine neue Zeile erstellt werden, wird die ID der neuen Zeile an den Task zurückgegeben. War die Aktion nicht erfolgreich, wird für das werfen einer Exception -2 oder -1, falls die E-Mail bereits in der DB existiert, zurückgegeben. Der Task erstellt nun ein *RegisterResponse*-Objekt mit der ID und dem passenden *ErrorCode* als Antwort. Aus dem erstellten Objekt wird ein String im XML Format gebaut und an den THandler zurückgegeben. Von diesem wird die Response an den Client als geschickt. Nun wird vom THandler die Methode *setHandled(true)* aufgerufen, weil der Request sonst zyklisch wieder an den THandler gesandt werden würde. Die unten stehende Grafik bietet einen Überblick auf die zur Server-Client Kommunikation verwendeten Klassen und ihre Beziehungen untereinander.



## 8. Ausblick in die Zukunft

Sowohl im Server als auch im Client sind alle Grundfunktionen die zu Beginn des Projekts festgelegt wurden umgesetzt. Es gibt jedoch einige Dinge die durchaus noch verbesserungswürdig sind. Besonders wichtig wäre es die Kommunikation von Server und Client sicherer zu gestalten. Das kann z.B. durch eine SSL-Verschlüsselung erreicht werden. An zweiter Stelle liegt die Verbesserung des Chats, hier müssen noch zwei Bugs gefixt werden. Zudem wäre es sinnvoll die veraltete Polling-Technik gegen neuere Verfahren auszutauschen, wie z.B. mithilfe eines Web-Sockets. Die nächsten Funktionen die auf der List der zukünftigen Updates stehen ist Implementierung einer Profilbildfunktion und eine bessere Überprüfung der E-Mail Adressen, mit denen sich die Nutzer registrieren. Besonders interessant könnte zudem auch noch die Arbeit mit verschiedenen APIs werden. Beispielsweise könnte man durch eine Implementierung der Facebook API die Registration vereinfachen. Mithilfe der Spotify API könnten Spotify Premium Nutzer ihren eigenen Soundtrack während dem Nutzen der App erstellen.

## **9. Schluss**

Im Großen und Ganzen war das Projekt ein Erfolg, ich konnte nahezu alle Ziele, die ich mir gesetzt habe, in meiner App umsetzen. Es war zwar am Anfang schwer sich mit neuen, dem Unterricht vorausgehenden, Konzepten, wie Activitys, verschiedene Designelemente der GUI, der Umsetzung einer Datenbank sowohl in Java als auch in Android umzugehen.

Sobald man jedoch die Grundprinzipien verstanden hatte, hat das Programmieren selbst durchaus Spaß gemacht.

## 10. Danksagungen

Abschließend möchte ich allen Seminarmitgliedern für ihre Offenheit und Hilfsbereitschaft danken. Insbesondere Tim Möschel, Leander Dreier, Dominik Okwieka und Oskar Loeprecht die mir jeder Zeit mit Rat beiseite standen.

## 11. Verwendete Programme

Programme, die ich zur Erstellung der schriftlichen Arbeit verwendet habe:

- Umlet
- Microsoft Word 2011

## 12. Literaturverzeichnis

ANDROID Open Source Project: *Activity*,

<https://developer.android.com/reference/android/app/Activity.html>, abgerufen am 22.10.2016

ANDROID Open Source Project: *Activity/ Shutting down an Activity, Managing the Activity Lifecycle*

<https://developer.android.com/reference/android/app/Activity.html>, abgerufen am 2.11.2016

ANDROID Open Source Project: *Saving Key-Value Sets*,

<https://developer.android.com/training/basics/data-storage/shared-preferences.html>, abgerufen am 22.10.2016

ANDROID Open Source Project: *Pause Your Activity*,  
<https://developer.android.com/training/basics/activity-lifecycle/pausing.html>,  
abgerufen am 1.11.2016

ANDROID Open Source Project: *Tasks and Back Stack*,  
<https://developer.android.com/guide/components/tasks-and-back-stack.html>,  
abgerufen am 1.11.2016

RACHITA NANDA Androidexteros: *Difference between sqlite and shared preferences in android*, <https://blograchita.wordpress.com/2013/05/26/difference-between-sqlite-and-shared-preferences-in-android/> abgerufen am 23.10.2016

ECLIPSE Jesse McConnell (last modified): *Jetty/Tutorial/Embedding Jetty*,  
[https://wiki.eclipse.org/Jetty/Tutorial/Embedding\\_Jetty](https://wiki.eclipse.org/Jetty/Tutorial/Embedding_Jetty) abgerufen am 2.11.2016

KAAZING websocket.org: *quantum*, <http://www.websocket.org/quantum.html>  
abgerufen am 31.10.2016

JAVA IST AUCH EINE INSEL, Christian Ullenboom: *24.6.1 Abfragen über das SQL Statement*, 9.Auflage, Kapitel 24.6.1,Seitenzahlen nicht bekannt da das Werk im html Format vorliegt.

JENKOV, Jakob Jenkov: *JDBC: CallableStatement*,  
<http://tutorials.jenkov.com/jdbc/callablestatement.html> abgerufen am 4.11.2016

GITHUB, Taro L. Saito (Xerial): *SQLite JDBC Driver/README*,  
<https://github.com/xerial/sqlite-jdbc> abgerufen am 4.11.2016

STACK OVERFLOW, Glen Best: *Difference between Statement and PreparedStatement*, <http://stackoverflow.com/questions/3271249/difference-between-statement-and-preparedstatement> abgerufen am 4.11.2016

Beispiele aus dem Unterricht

### **13. Eidesstattliche Erklärung**

Ich erkläre, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis aufgeführten Quellen und Hilfsmittel verwendet habe.

Kösching, den 8. November 2016

---

Abgabe nach den Ferien 8.11.16

**Stuff:**

<http://www.websocket.org/quantum.html>

[17:08](#)