# Deckblatt

# Inhaltsverzeichnis

# 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.[1]

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 unnötig langem Code und würde somit die Verständlichkeit und Übersichtlichkeit dessen stark verschlechtern.

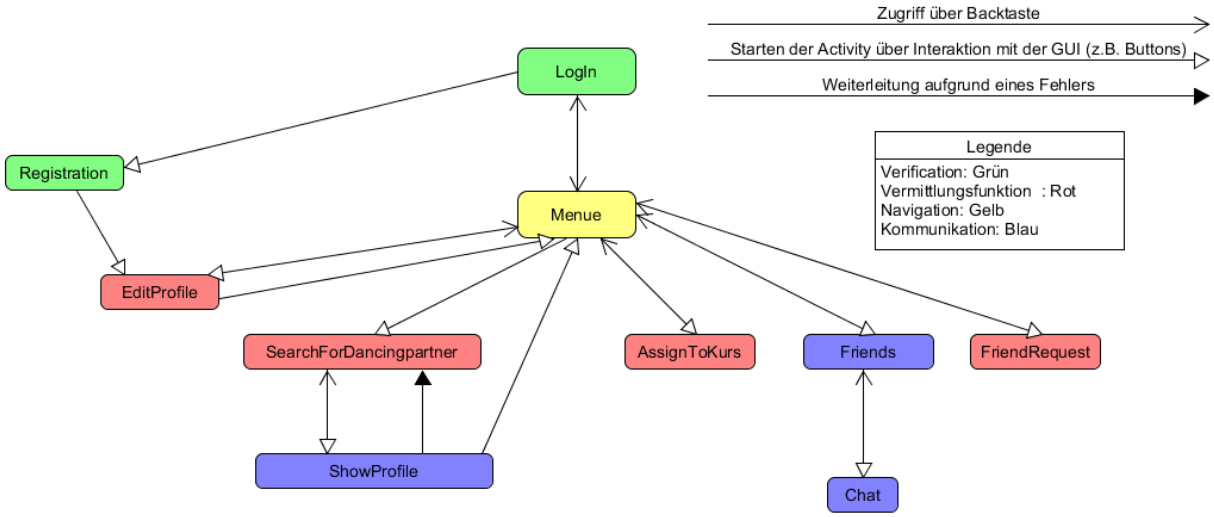
Zu guter Letzt müssten bei der neuzeichnenden Variante sehr viele Attribute in der Activity-Klasse und so im Arbeitsspeicher des Geräts abgespeichert werden die nur in bestimmten Zuständen der App überhaupt Verwendung finden.

Beim Verwenden mehrerer Activitys ist die Abgrenzung untereinander deutlich einfacher. Denn in diesem Fall kann der vordefinierte Lebenszyklus einer Activity verwendet werden. Dieser bring Methoden wie *onPause*, *onStop* oder *onDestroy* mit sich in denen nicht mehr verwendet Ressourcen freigegeben werden können. Diese werden aufgerufen, wenn die Activity, neugezeichnet, temporär in den

[1] Vgl. ADROID, *Activity*

Hintergrund verschoben oder beendet wird. Beim 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. SharedPreferrences gespeichert werden.




Diese Grafik zeigt die in der Tanzschul App vorhandenen Aktivities und so auch alle Zustände. Die Pfeile veranschaulichen, durch welche Art von User Interaktion der Zustandswechsel ausgelöst wurde und so auch eine neue Aktiviy in den Vordergrund rückt. Alle grün markierten Aktivitys sind 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 Acivitys sind sozusagen das Herz der App sie sind sie Implementieren die Funktionen die zur eigentlichen Vermittlung beitragen. Zur direkten oder indirekten Interaktion zwischen Usern dienen die blau gefärbten Activitys.

# 4. Die Funktionen der App und ihre Umsetzung

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

ist eine abstrakte Klasse die von allen Activitys erweitert wird, die auf die Internetressourcen des Endgeräts zugreifen oder deren Existenz 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 Activitys 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. In diesem Fall wird die Verbindung 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 Fehlern *nf (not found)*, *wl (wrong login)* und *ja (alles passt)*. Tritt im Server ein bestimmter Fehler auf wird der zum Fehler passende Fehlercode zum Client gesandt. Ist der vom Server erhaltener Fehlercode auf einen Eingabefehler zurück zu führen wird eine passende Fehlermeldung ausgelesen. Der Fehlercode *wl* wird vom Client erhalten falls die vom User im Request mitgesendete Identifikationsnummer, mit der sich jeder User ausweisen muss, 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 zu 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 eine 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 Überprüfung und 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 mir hier, den Login für den User bequem zu gestalten, deshalb wurde eine Funktion hinzugefügt mit der man E-Mail und Kennwort 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, beispielsweiße 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 die SharedPreferences an, die genau dafür ausgelegt sind wenige Datensätze primitiven Datentyps oder Strings in Datenpaaren zu speichern.[2] Möchte man komplexe Datentypen speichern ist eine SQLite Datenbank besser geeignet.[3] Das Speichern komplexerer Daten sollte 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.[4]

[2] Vgl. RACHITA NANDA
[3] Vgl. ANDROID, *Saving Key-Value Sets*
[4] Vgl. ANDROID, *Pause Your Activity*

Kommen wir zur Registration-Activity der schwierigste Teil des Codes liegt darin, die vom Nutzer eingegebenen Daten auf theoretische Richtigkeit zu überprüfen und passende Fehlermeldungen auszulesen. Letztendlich habe ich die Überprüfung mit einem *boolean array* umgesetzt, das für jede zu erfüllender Bedingung ein item aufweist, nur wenn alle Items des Arrays den Wert true haben wird dem Sever eine Anfrage zum Erstellen eines neuen Accounts zugesandt. Dabei ist anzumerken, dass sowohl Client als auch Server seitig eine passende Überprüfung der Gültigkeit einer E-Mail fehlt.

## 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. Ich habe für die Stufenbezeichnungen der Grundkurse von der Tanzschule Fischer Ingolstadt übernommen. Die höheren Stufen entsprechen dem einheitlichen Medaillensystem der ADTV geprüften Tanzschulen. Der Wochentag bezeichnet den Tag in dem die wöchentlichen 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 getan werden.

## 4.3 Das Freunde-System und die Chatfunktion

Für den erfolgreichen Abschluss der App war es mir war es mir wichtig, dass diese tatsächlich die Kontaktaufnahme zwischen den Nutzer erleichterte, ein Chat durfte dazu nicht fehlen. Die Implementierung eines solchen Chats war jedoch deutlich komplizierter als anfangs erwartet. Der Chat der Tanzschul App sollte nämlich kein einfacher Gruppenchat zwischen allen Usern sein. Sondern eine Kommunikationsmöglichkeit zwischen zwei Usern bieten. Ein Chat soll jedoch nur

bestehen, wenn beide User dessen Entstehung zugestimmt haben. Um das 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 Messanger gedacht. Deshalb ist es zu diesem Entwicklungsstand nicht möglich andere Nutzer über ihren Namen zu suchen und so zu 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 gelang.

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 neuen 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 List versandt.

Polling allgemein steht für ein zyklisches Abrufen einer bestimmten Information. Wenn sich diese ändert kann man relativ zeitnah die angerufene Änderung übermitteln und verarbeiten. Der maximale Delay ist vor allem von der Zeitspanne in dem der Zyklus ausgeführt wird abhängig. Es gibt mehrere Arten des Pollens darunter short-polling und long-polling.

Beim short-polling wird in einem bestimmten Zeitabstand ein Requests an den Server gesandt dieser antwortet gleich nach dem Erhalten des Requests mit einem 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. Nachteil an diesem Prinzip ist, dass wenn die Daten nicht zyklisch in der Datenbank erneuert werden viele Requests an den Server eventuell nutzlos sind, da es keine neuen Daten zu übermitteln gibt. Bei einem Chat zwischen mehreren Nutzern trifft dieser Nachteil offensichtlich zu.

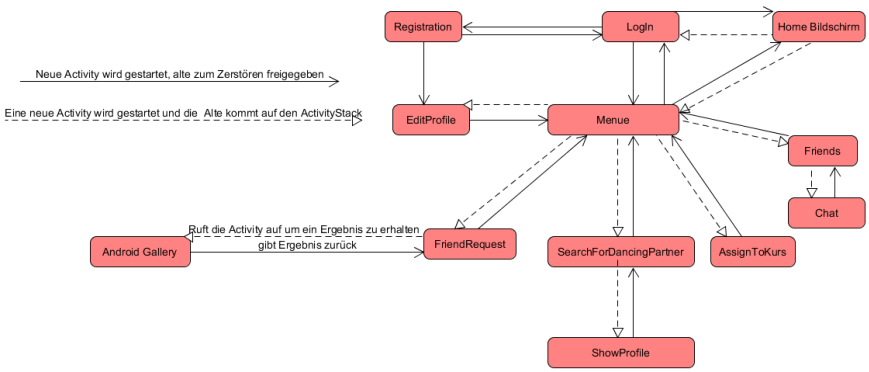
Eine andere Methode ist long-polling bei diesem Prinzip wird nach dem Erhalt des Requests serverseitig die gewünschte Information zyklisch abgerufen. Sobald die Abgerufene Information nicht mehr mit dem Stand des Clients übereinstimmt wird der Zyklus gestoppt und die neue Information an den Client weitergeleitet.[5] 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 meinen Chat sind beide Methoden durchaus denkbar, ich habe mich letztendlich jedoch für short-polling entscheiden, da es für mich leichter umzusetzen war.

Zudem ist short-polling in meinem Fall um einiges effektiver, da bei mir nur während die Chat-Activity im Vordergrund ist überhaut gepollt wird. Eine Alternative zum Pollen wäre die Umsetzung mit eine Websocket für diese Umsetzung wäre in meine Fall jedoch unnötig kompliziert gewesen insbesondere, da der Chat keine Haupt sondern eine Nebenfunktion der App ist.

[5] Vgl. KAZING

# 5. Activity-Stack



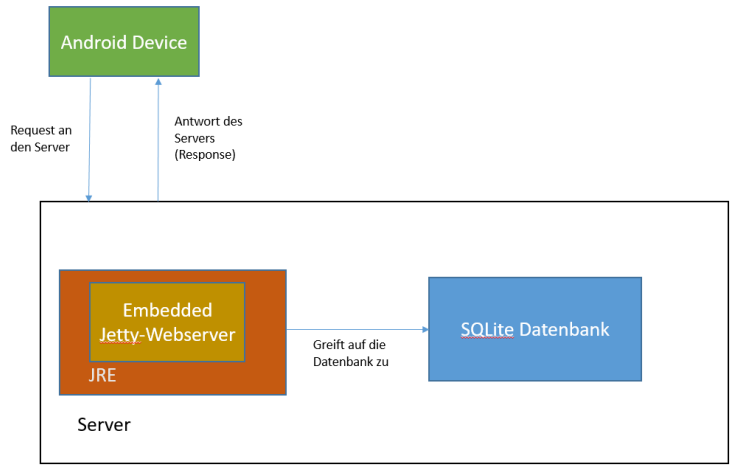

„A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a stack (the *back stack*), in the order in which each activity is opened."(Android Open Source Project, 2016)[6] Dieses Zitat beschreibt in kurzen Worten, was ein *back stack* ist und wie die abgelegten Activitys im LiFo (Last in First out) Prinzip angeordnet sind. Jedes Mal wenn eine Activity eine andere aufruft rückt die aufrufende Activity in den Hintergrund und wird auf den *back stack* gelegt. [7] Wenn der Nutzer die Zurücktaste an seinem Endgerät betätigt wir *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 Rechenvorgänge 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 *onBackPressed()* oder vor dem Aufrufen von *finish()* eine neue Activity gestartet wurde.[8] 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 Threads in denen die Activitys laufen verwaltet und der Stack jedes einzelnen Threads angepasst werden.[9] Die Tanzschul App besitzt kein kompliziertes System aus verschiedenen Threads für unterschiedliche Activitys. In der Tanzschul App laufen alle Activitys in einem Thread, nur Aufgaben wie das Versenden von

[6] Vgl. ANDROID, *Tasks and Back Stack*
[7] Vgl. ANDROID, *Tasks and Back Stack*
[8] Vgl. ANDROID, *Activity/ Shutting down an Activity, Managing the Activity Lifecycle*
[9] Vgl. ANDROID, *Tasks and Back Stack / Managing Tasks*

Requests werden in separaten Threads abgehandelt. Auch Flags finden hier kaum Anwendung, doch auch hier wurde dafür gesorgt, dass keine Zyklen entstehen können. Wie in der Abbildung zu sehen ist, hab ich das Entstehen von Zyklen dadurch verhindert, indem ich in Activitys die nicht auf den Back Stack gelangen sollen die *finish()* Methode aufgerufen habe. 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. Denn es wird immer wenn es möglich ist die Activty vom Back Stack genommen. Dabei ist anzumerken, dass nur die Menue Activity vom Login bis zum Logout bzw. dem löschen des Activity Stacks durch das System im Activity Stack bleibt. Alle anderen Activitys werden, wie in der Zeichnung auch zu erkennen, Pfad ähnlich durchschritte. 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.

Doch auch Flags finden bei mir direkte 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.

# 6 Aufbau des Servers



Der in Java geschriebene Server ist mindestens genauso wichtig wie der Client. Mithilfe von JDBC kann er auf auf eine SQLite Datenbank zugreifen, in dieser werden alle wichtigen Daten, von den Nutzern gesendeten Daten, gepeichert. 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. Zudem wurde das Jetty 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)[10]

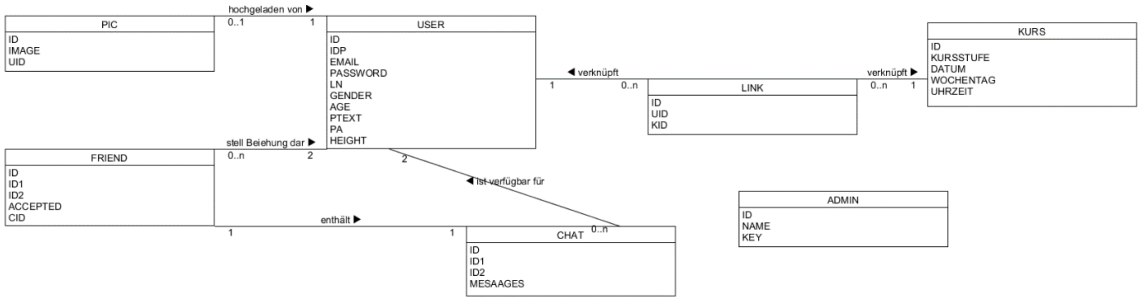
Der Server der Tanzschul App wurde auf Basis von im Unterricht besprochenen Inhalten aufgesetzt und weiterentwickelt. Das Diagramm ist ebenso dem im Unterricht gezeigten nachempfunden.

## 6.1 Datenbank

Zum Lesen und Schreiben in eine SQLite Datenbankdatei wird die JDBC Bibliothek verwendet.[11] Mithilfe der JDBC können SQL Statements formuliert werden über die, 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 Clientseitig Datenbank bestand jedoch schon in den ersten Konzepten. 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.

[10] Vgl. ECLIPSE, *Jetty/Tutorial/Embedding Jetty*
[11] Vgl. GITHUB, *SQLite JDBC Driver*

## 6.1.1 Struktur der Datenbank




Meine Datenbank hat insgesamt 7 verschiedene Tabellen. In diesem Teil wird genauer auf die Funktionen der einzelnen Tabellen und wie sie untereinander zusammenhängen 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, sind zum jetzigen Zeitpunkt aber nicht anpassbar. Eine neue Zeile der Tabelle User wird dann erstellt, wenn einen 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 auslesen sollte, 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 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 kann nur der Nutzer selbst seine wirkliche ID erfahren. Die KURS Tabelle speichert alle Kurse, die 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 da, 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 die 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[12] 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.[13] Da die Datenbank der App jedoch keine riesigen Datenmengen aufweist, finden Callable Statements hier keine Anwendung.

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 wir 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 da

[12] JAVA IST AUCH EINE INSEL, *Abfragen über das SQL Statement*
[13] Vgl. JENKOV

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.[14]

### 6.1.3 Umsetzung des Datenzugriffs

Die Klasse *Model* besitzt eine Instanz von sich selbst die sie im statischen Konstruktor instanziiert, insofern sie nicht bereits existiert. Die Methoden dieser Klasse dienen 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, die tatsächlich Methoden zur Interaktion mit der Datenbank ausführt. Es gibt 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 Methoden 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 werde, wenn sie sich alle in einer Klasse befinden, da Eclipse keine Projekt weite Suche unsterstützt.

[14] STACK OVERFLOW, *Difference between Statement and Prepared Statement*

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 die Veröffentlichung trotzdem umzusetzen, da es anderen Programmierern wahrscheinlich einfacher fallen wird mit eine 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. Diese ist dabei auch der wichtigste Teil des Projekts denn ohne die Informationen des Servers würde der Client nicht funktionieren. Das Grundprinzip der 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 werden die Vorgänge genauer unter die Lupe genommen und zur Kommunikation benötigte Datenstrukturen am Beispiel des RegisterRequest genauer beleuchtet.

## 7. 1 Client

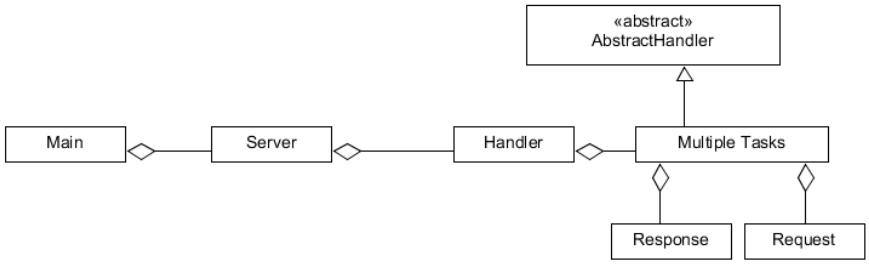
In dem Vorliegenden Konzept kann nur der Client einen Request versenden, welchen der Server mit einer Response beantwortet. Um einen LoginRequest zu versenden, muss der *RegisterTask* von der *Register-Activity* aus gestartet werden. Alle Task laufen in einem eigen Thread, denn sie erben alle von der *BaseHttpRequestTask* Klasse, welche ein Kind der Klasse *AsyncTask* ist. Im *RegisterTast* wird ein neues *RegisterRequest*-Objekt, das die vom Nutzer angegebenen Registrationsdaten beinhaltet. Aus dem Requestobjekt wird nun ein XML- String gebaut. In der *doInBackground(String… params)* Methode des Tasks wird aus der Server URL, die Trennung der URL vom den übermittelten Daten „?&do= „ dem Befehl „*register“* und dem XML-String ein Http-Request erstellt und an den Server gesandt. Alle Befehle, die an den Server gesandt werden können sind in dem Enum Command gespeichert. Ist der HttpRequest versandt, wartet der *RegisterTask* auf die Response des Servers. Insofern diese ankommt muss diese zu eine Request Objekt geparst 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 eine neue Menue-Activity wird 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 HttpServletRequests 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 geparst. 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. Beispielsweiße 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 Androidexterous: *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 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 PreparedStatemen*, 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