**Deckblatt**

**Inhaltsverzeichnis** Seite

1. Die Idee hinter der App

2. Vorteile der App gegenüber herkömmlicher Methoden

3. Die App und ihre Zustände

3.1 Übersicht

4. Die Funktionen der App und deren Umsetzung

4.1 Login und Registration

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

6.1.1 Umsetzung Datenzugriffs

6.1.2 PreparedStatements und Statements

6.2 Jetty

6.2.1 Starten des Webservers

6.2.2 XML

7 Server-Client Kommunikation

7.1 Client

7.1.2 Request und Response

7.2 Server

7.1.2 Request und Response

8. Verschlüsselungen

8. Danksagung

9. Literaturverzeichnis

10. Eidesstattliche Erklärung

Verschlüsselung /Er Diagramm/ Zyklen/ neuer aufruf

**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 ist 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]](#footnote-1)

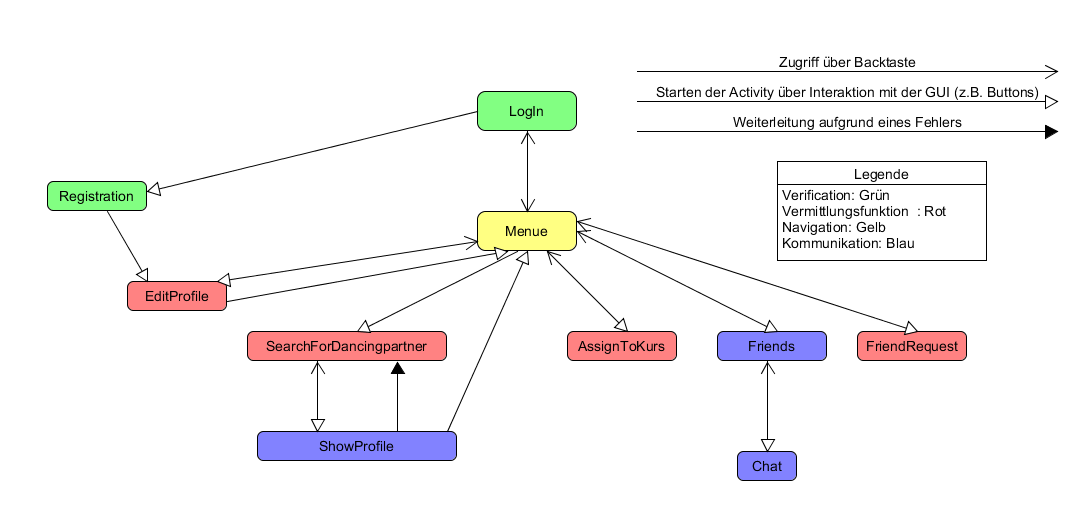
Ein Konstrukt aus mehreren Activities hat viele Vorteile gegenüber einem Neuzeichnen der [GUI](https://developer.android.com/guide/topics/ui/index.html) (Graphical User Interface) innerhalb einer einzigen [Activity-Klasse](https://developer.android.com/reference/android/app/Activity.html), für jeden 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](https://developer.android.com/training/basics/activity-lifecycle/index.html) 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 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](https://developer.android.com/reference/android/content/SharedPreferences.html) gespeichert werden.

**3.1 Überblick über die Zustände der App**

****

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* sollendie 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 Funktion**

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]](#footnote-2) Möchte man komplexe Datentypen speichern ist eine SQLite Datenbank besser geeignet.[[3]](#footnote-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]](#footnote-4)

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, an der ich einst selbst Schüler war übernommen. Die höheren Stufen entsprechen dem einheitlichen Medaillensystem der [ADTV](http://www.adtv.de/) 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 AssingToKurs-Activity getan werden.

**4.3 Das Freunde-System und die Chatfunktion**

Schon in den ersten Konzepten der App Bestand die Idee einen Chat zu implementieren. Dabei hört sich das ganze sehr einfach an. Bei genauerer Betrachtung ist dieser jedoch mit einiger Komplexität verbunden. Der Chat der TanzschulApp ist nämlich kein einfacher Gruppenchat zwischen allen Usern sondern zwischen zwei Usern die diesen Chat auch beide wünschen. Um das zu ermöglichen wurde ein Freunde System konzipiert.

Der Chat dient ausschließlich zum Austausch weiterer Kontaktdaten und nicht zum Führen längerer virtueller Unterhaltungen. Deshalb ist es nur möglich einem User über einen seiner Steckbriefe, die in der SearchForDancingpartner-Activity einzusehen sind, eine Freundschaftsanfrage zu senden. Das resultiert automatisch darin, dass nur Chats zwischen Usern unterschiedlichen Geschlechts bestehen können. 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.

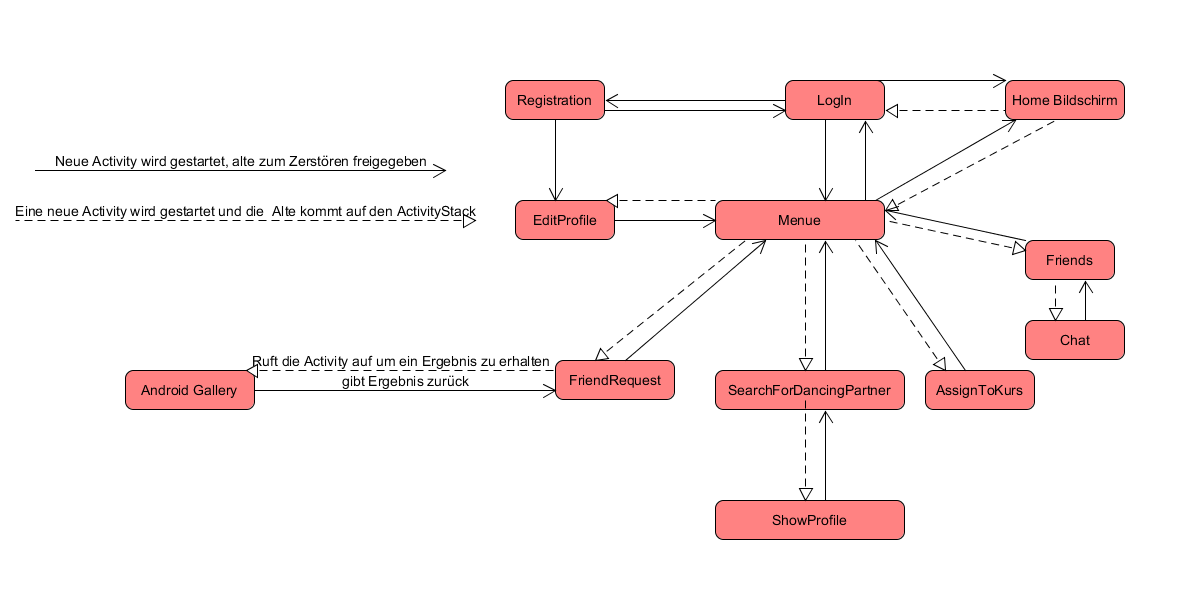
Beim short-polling wird in einem bestimmten Zeitabstand ein Requests an den Server gesandt dieser antwortet gleich nach dem Erhalten des Requests mit einem Response. Nachdem der Client die Response erhält gilt der Request als beendet. Dieses Verfahren ist besonders günstig um Daten abzurufen, die sich 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]](#footnote-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. Activity-Stack**

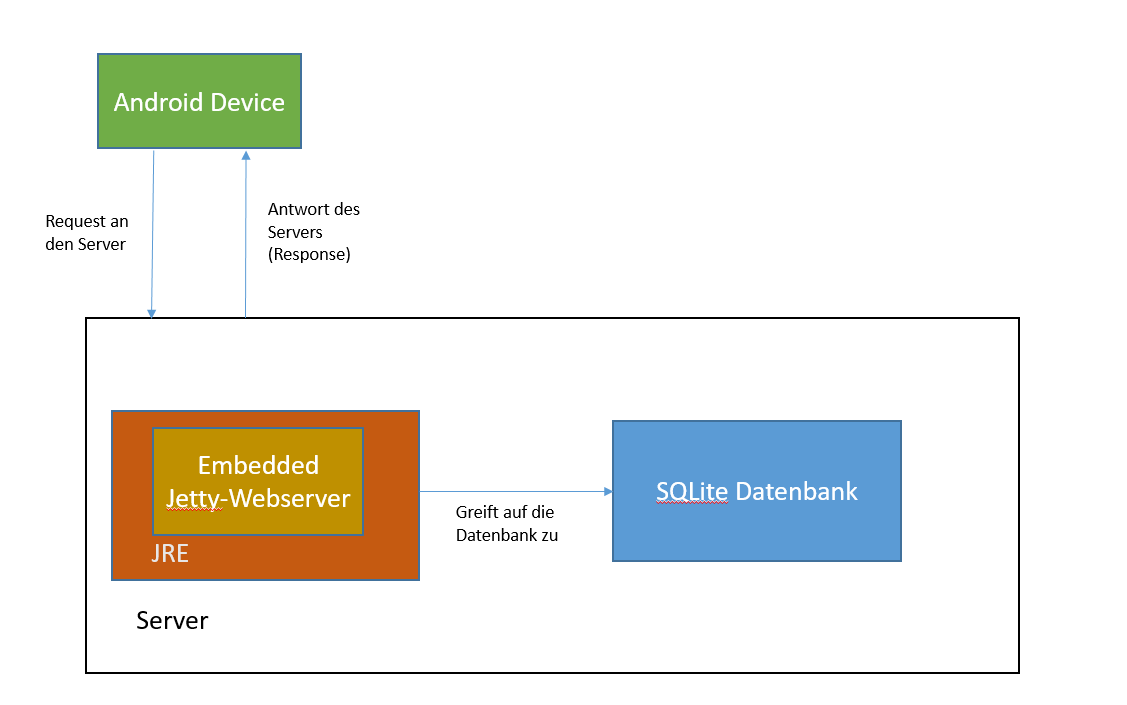


„A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a stack (the back stack), in the order in which each activity is opened.”(Android Open Source Project, 2016)[[6]](#footnote-6) Dieses Zitat beschreibt in kurzen Worten, was ein *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]](#footnote-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]](#footnote-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]](#footnote-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 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 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.

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) einbetten lässt. Zudem wurde das Jetty-Framework 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]](#footnote-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**

Die Datenbank stellt einen großen Teil des Servers da, das Programmieren dieser hat einen großen Teil der investierten Zeit in Anspruch genommen. Ich habe mich dazu entschieden die Datenbank mithilfe der Java Database Connectivity, kurz JDB, zu betreiben. Zum Beschreiben wird SQLite verwendet. Ich habe mich insbesondere für SQLite entscheiden, da es sehr weit verbreitet ist und es viel Dokumentation zum Umsetzen einer SQLite Datenbank in Java gibt. Zumdem reichen die in SQLite enthaltenen Funktionen vollkommen für meinen Verwendungszweck aus.

**6.1.1 Struktur der Datenbank**

Meine Datenbank besteht aus X Tabellen und X Abfragen.

**6. Danksagungen**

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

**7. Literaturverzeichnis**

ANDROID Open Source Project: *Activity*, https://developer.android.com/reference/android/app/Activity.html, aufgerufen 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, aufgerufen am 2.11.2016

ANDROID Open Source Project: *Saving Key-Value Sets*, https://developer.android.com/training/basics/data-storage/shared-preferences.html, aufgerufen am 22.10.2016

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

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

STACK OVERFLOW

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/ aufgerufen am 23.10.2016

ECLIPSE Jesse McConnell (last modified): *Jetty/Tutorial/Embedding Jetty*, <https://wiki.eclipse.org/Jetty/Tutorial/Embedding_Jetty> aufgerufen am 2.11.2016

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

**8. Eidesstattliche Erklärung**

Kösching, den 10. November 2016

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Abgabe nach den Ferien 8.11.16

**Stuff:**

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

https://acloud.guru/course/aws-certified-developer-associate/discuss/-KMR-yxQ5zh83ngSIXmB/why-would-you-use-short-polling-over-long-polling

1. Vgl. ADROID, *Activity* [↑](#footnote-ref-1)
2. Vgl. RACHITA NANDA [↑](#footnote-ref-2)
3. Vgl. ANDROID, *Saving Key-Value Sets* [↑](#footnote-ref-3)
4. Vgl. ANDROID, *Pause Your Activity* [↑](#footnote-ref-4)
5. Vgl. KAZING [↑](#footnote-ref-5)
6. Vgl. ANDROID, *Tasks and Back Stack* [↑](#footnote-ref-6)
7. Vgl. ANDROID, *Tasks and Back Stack* [↑](#footnote-ref-7)
8. Vgl. ANDROID, *Activity*/ *Shutting down an Activity, Managing the Activity Lifecycle* [↑](#footnote-ref-8)
9. Vgl. ANDROID, *Tasks and Back Stack / Managing Tasks* [↑](#footnote-ref-9)
10. ECLIPSE, *Jetty/Tutorial/Embedding Jetty* [↑](#footnote-ref-10)