



OpenDolphin

Ein Showcase für verschiedene Client-Plattformen

Auftraggeber: Canoo Engineering AG
Prof. Dierk König

Betreuung: Dr. Dieter Holz

Zusammenfassung

Die Firma Canoo mit Sitz in Basel entwickelt die Open-source Plattform «OpenDolphin». Diese fokussiert sich auf die Nutzung des «Presentation Model Patterns», welches eine einfache Einbindung verschiedener Frontend-Lösungen an ein- und denselben Server ermöglicht.

Das Ziel ist es, für die Firma Canoo einen repräsentativen Showcase zu entwickeln, welcher die von OpenDolphin vertretenen Konzepte und Lösungen optimal mit einbezieht. Der Showcase kann als Werbung oder als Beispielprojekt dienen, um möglichen Kunden aufzuzeigen, welche Möglichkeiten sich ihnen mit dieser Bibliothek bieten.

Um den Showcase möglichst attraktiv zu gestalten, musste eine Idee her, welche das Projekt umspannt. Es ging dabei um eine Applikation, welche es einem ermöglicht, in der Gruppe einen geeigneten Essensstandort und -zeitpunkt zu finden. So kann man als Benutzer darin einen virtuellen Tisch mit Zeit- und Ortsangabe erstellen, an welchen man andere Benutzer einladen kann.

Da OpenDolphin schon einige Client-Showcases auf ihrer Webseite anbietet, welche hauptsächlich auf dem Framework JavaFX oder der Scriptsprache JavaScript basieren, wurde der Wunsch des Kunden im Verlauf des Projektes gross, sich auf einen noch nicht existierenden Showcase zu konzentrieren. Heraus kristallisiert hat sich dabei die Umsetzung eines Android-Clients. Das Projektteam hatte wegen einer Fehleinschätzung neben einem inzwischen fertigen Android-Clients vorgängig viel Zeit in einen JavaFX- sowie einen JavaScript-Clients investiert, sodass das Endresultat nun zwei unfertige Clients beinhaltet. Hier mangelte es an einer konkreten Analyse des neuen Auftrages, man hatte die Arbeit mit den verschiedenen Technologien unterschätzt und dem Kunden infolgedessen zuviel versprochen.

Inhaltsverzeichnis

1 Einleitung	5
2 Umsetzung	6
2.1 Vorbereitung und Planung	6
2.1.1 Anforderungen für das MVP	6
2.1.2 Optionale Anforderungen	7
2.2 Design	8
2.2.1 Ideengenerierung	8
2.2.2 Wireframes	9
2.2.3 Designentscheidungen während der Entwicklung	12
2.2.4 Abschliessendes zum Design	16
2.3 GitHub & Wiederherstellung der Projekte	17
2.3.1 Issues & Projects	17
2.3.2 Code auf GitHub	17
2.3.3 Wiederherstellung der Projekte	18
2.3.3.1 Server	18
2.3.3.2 Android	19
2.4 Source Code	20
2.4.1 OpenDolphin Server-Applikation	20
2.4.1.1 Client	20
2.4.1.2 Combined	21
2.4.1.3 Pmdescription	21
2.4.1.4 Server	21
2.4.1.5 Server-app	24
2.4.1.6 Shared	24
2.4.2 JavaFX-Client	25
2.4.3 JavaScript-Client	25
2.4.4 Android-Client	31
2.4.4.1 Die Projekt-Struktur	32
2.4.4.2 Ablauf	32
2.4.4.3 Activity vs. Fragments	33
2.4.4.4 Veneer	34
2.4.4.5 Adapter	34
2.4.4.6 Features	35
3 Schlusswort	37
4 Abkürzungsverzeichnis	39
5 Abbildungsverzeichnis	39
6 Quellenverzeichnis	40

1 Einleitung

Die Welt ist im digitalen Wandel. Es gibt unzählige Apps für alle möglichen Anwendungen. Die Anzahl der Benutzer sowie die Vielfalt der Hardware steigt stetig. Ein durchschnittlicher Benutzer ist heute nicht mehr auf einen stationären Computer fixiert, vielmehr wurde sein Smartphone zum täglichen Begleiter und Hauptgerät. Aber auch andere Hardwaretypen wie zum Beispiel Tablets, Smartwatches, Smart-TVs oder IoT-Devices sind stark am aufkommen. Da verwundert es nicht, dass die Erwartungen an die App-Entwickler gross sind, möglichst viele Hardware-Plattformen gleichzeitig zu bedienen. Nur schon wenn man die gängigsten Smartphone- und Notebook-Typen abdecken will, schlägt man sich als Entwickler mit diversen Technologien im Bereich von Android, iOS, Windows und dem Web herum. Da ist ein starkes Know-How auf allen geplanten Plattformen gefragt. Wenn dann sowohl die Logik wie auch die View der App für alle diese Plattformen separat entwickelt werden, steigt der Programmieraufwand schnell ins Unermessliche.

Hier kommt die Plattform OpenDolphin, welche von der Canoo Engineering AG entwickelt wurde, ins Spiel. Bei dieser wird die gesamte Logik einer Applikation auf dem Server installiert. Auf dem Client liegt nur noch die View. Was angezeigt werden soll, bestimmt OpenDolphin und stellt dafür sogenannte Presentation-Models bereit. Clients unterschiedlicher Herkunft können sich an dieses Presentation-Models binden und übernehmen nur noch die Darstellung der Daten.

Der Auftrag

Im Rahmen des Projektes 5 im Studiengang iCompetence an der Technischen Fachhochschule Nordwestschweiz in Windisch soll für die Firma Canoo einen repräsentativen Showcase entwickelt werden. Dieser soll die Stärken von OpenDolphin hervorheben. Die Wahl der Client-Technologie ist grundsätzlich nicht in Stein gemeisselt und kann sich daher im Verlauf des Projektes noch ändern..

Die Startphase

Was das Thema Technologiewahl anbelangt, so waren sich Betreuer, Kunde und das Projektteam zu Beginn uneinig. Der Betreuer war der Meinung, dass eine JavaFX-Applikation den einfachsten Einstieg in die OpenDolphin-Welt bietet. Von Seiten des Kunden wurde jedoch der Wunsch geäussert, sich auf eine andere Technologie zu konzentrieren, da für JavaFX bereits Beispielprojekte existieren. Das Projektteam hingegen wollte die Idee der Applikation mehr in den Vordergrund stellen und damit viele Leute erreichen. Worauf der Wunsch nach einem Webclient aufkam, da man mit einer Weblösungen unterschiedliche Plattformen gleichzeitig bedienen kann.

Nach einer Einführung zum Thema OpenDolphin von Betreuer und Kunde, hat man sich mit der Server-Applikation auseinandergesetzt. Gleichzeitig machte man sich daran, einen einfachen JavaFX-Client aufzubauen, welcher beim Verständnis der Server-Applikation unterstützend wirken soll. Bald fühlte man sich sicher, was die Basics von OpenDolphin anbelangte und man musste diskutieren, in welche Richtung sich das Projekt entwickeln soll. Gemeinsam an einem Tisch hat man sich unter der Anwesenheit von Kunde und Betreuer darauf geeinigt, so schnell wie möglich mit der Entwicklung eines Android-Clients und eines JavaScript-Clients zu starten. Wobei sich die eine Hälfte des Teams auf den Android-Client und die andere auf den JavaScript-Client konzentrieren sollte. Dies schien die ideale Lösung zu sein, da man damit sowohl die Interessen des Kunden decken konnte, wie auch die persönlichen Interessen der beiden Projekteure.

2 Umsetzung

Im Folgenden wird auf die Ausarbeitung der OpenDolphin-Server-Applikation sowie auf die Entwicklung der verschiedenen Clients eingegangen. Ebenfalls werden zu Beginn die Anforderungen erläutert, so wie einige Prototypen vorgestellt.

2.1 Vorbereitung und Planung

In einer ersten Phase musste festgelegt werden, was die Anforderungen an die geplante Applikation sind. Die Usability bzw. das Aussehen wurde vorerst in den Hintergrund gestellt, da zu diesem Zeitpunkt noch nicht klar war, ob eine Desktop- oder eine Mobile-Applikation entwickelt werden soll. Zusammen mit dem Kunden Dierk König sowie der Betreuungsperson Dieter Holz wurden folgende Punkte für ein Minimum Viable Product festgelegt:

2.1.1 Anforderungen für das MVP

Server-Applikation

Der Grundstein bildet eine Server-Applikation auf Basis von OpenDolphin. Da die Applikation im Endeffekt als Showcase dient, muss diese streng nach den Prinzipien von OpenDolphin aufgebaut sein. So soll zum Beispiel die Applikationslogik auf der Serverseite implementiert werden. Hingegen werden die Clients ausschliesslich zur Darstellung der Inhalte genutzt.

Tische

Als Abstraktion für eine Essensgruppe werden Tische verwendet. Ein Tisch wird von einem Benutzer erstellt und hat diverse Attribute wie einen Standort, ein Reservationsdatum und eine Zeit, eine Anzahl von Stühlen bzw. Plätzen und eine Liste

mit Benutzern, welche am Tisch Platz genommen haben. Ein Tisch existiert bis zwei Stunden nach Erreichen der Reservationszeit am entsprechenden Datum.

Stream

Alle erstellten Tische sollen bei jedem Benutzer in Form eines Streams dargestellt werden. Dies ist sogleich die Hauptansicht der Applikation. Hier hat der Benutzer die Möglichkeit, neue Tische zu entdecken und auch gleich an einem Tisch Platz zu nehmen, wenn er das denn möchte.

Erstellen und Ändern

Neue Tische werden von den Benutzern erstellt. Ein existierender Tisch kann nur vom Ersteller weiter bearbeitet bzw. verändert werden. Dies ist möglich über eine separate Ansicht oder in einem speziellen Editiermodus.

Platz nehmen

Wie oben bereits erwähnt, kann ein Benutzer an einem bestimmten Tisch Platz nehmen. Dafür muss der Tisch noch freie Plätze bieten, denn jeder Tisch hat eine begrenzte Anzahl Plätze. Ausserdem darf der Reservationszeitpunkt nicht in der Vergangenheit liegen. Sind diese zwei Bedingungen erfüllt, kann sich der Benutzer über einen Klick auf dem Button «Platz nehmen» am Tisch beteiligen. Über denselbigen Button, welcher nach erfolgreicher Anmeldung eine neue Bezeichnung erhält, kann sich der Benutzer auch wieder aus der Teilnehmerliste austragen.

2.1.2 Optionale Anforderungen

Neben den Anforderungen für das Minimum Viable Product wurden auch noch weitere Funktionen in Aussicht gestellt:

Profile

Ein Benutzer kann ein Profil erstellen, in welchem er Informationen über sich und seine Tische teilt. Es soll möglich sein, das Profil mit einem Passwort zu schützen. Mit einer Profilkfunktion wären auch Funktionalitäten wie zum Beispiel eine Freundschaftsliste oder eine Follower- bzw. Beobachtungsfunktion möglich.

Benachrichtigungen

Über Push-Notification-Services lösen die verschiedenen Clients je nach Plattform entsprechende Benachrichtigungen für den Benutzer aus. Mögliche Szenarien wären zum Beispiel eine Erinnerung für ein anstehendes Essen oder ein Hinweis, dass eine Person den Tisch verlassen hat.

Einladungen versenden

Ein Tisch generiert automatisch einen entsprechenden Link. Dieser kann man über die App seiner Wahl teilen. Damit erreicht man schnell die Personen, die man wirklich am Tisch haben möchte. Freunden bleibt das mühsame Suchen erspart. Mit einem Klick auf den Link landen sie direkt am richtigen Ort.

Spielerei

Auf spielerische Art und Weise kann man sich zu Tisch begeben. Mögliche Ideen wären zum Beispiel ein Tisch-Roulette, bei welchem man zufällig an einen Tisch gesetzt wird. Dies ist eine Kampfansage an alle gängigen Dating-Apps.

2.2 Design

Um die Programmierarbeit zu erleichtern und die Vorstellungskraft anzuregen, wurden die in [Unterkapitel 2.1](#) erwähnten Anforderungen als Prototypen umgesetzt.

2.2.1 Ideengenerierung

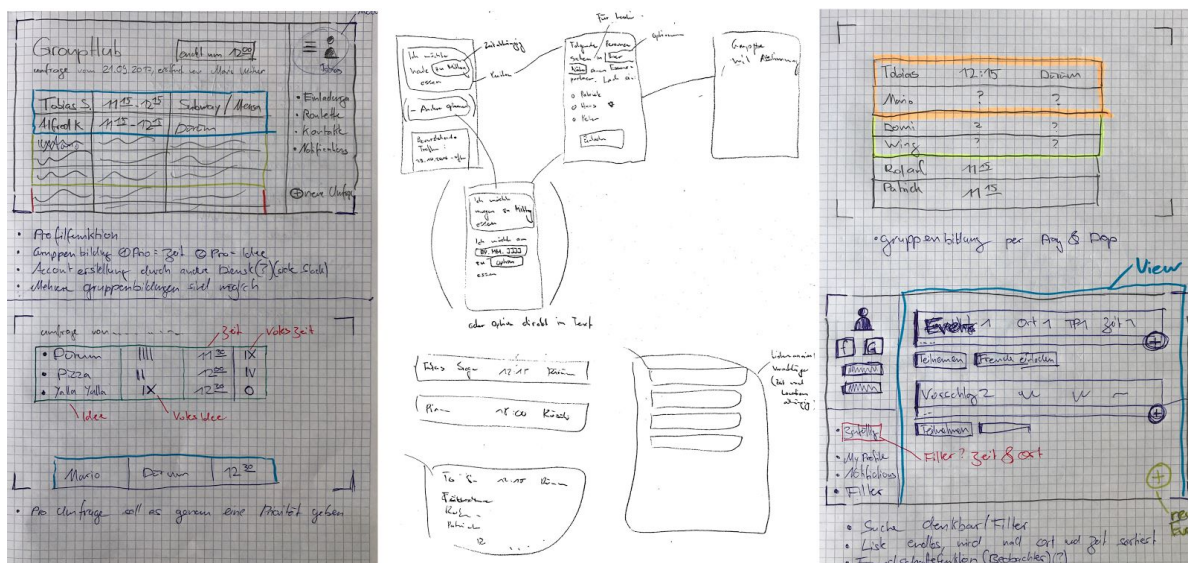


Abbildung 1: Die ersten Prototypen

In Abbildung 1 ist rechts oben die Übersicht der Tische als Tabellenform illustriert. Dies war die ursprüngliche Idee, sich stark an die Konkurrenz, namentlich Doodle, zu halten. Diese Ansicht wurde schnell verworfen und wie man später sehen wird, durch einen Stream von Cards ersetzt.

Die Profilansicht, so wie sie ebenfalls oben rechts oder unten links als Schieber zu erkennen ist, wurde so beibehalten. Ebenfalls ist der +-Button zum Erstellen eines neuen Tisches geblieben.

In der Mitte unten sehen wir eine erste unausgereifte Idee für einen Card-Stream. In der Mitte oben sieht man einen textbasierten Leitstrahl. Vorgängig wurde mal der Idee nachgegangen, den Benutzer auf der Basis eines Textes durch die Applikation zu führen. Allerdings kam man davon wieder weg, weil es einen erheblichen Mehraufwand nach sich gezogen hätte.

Kunde wie auch Betreuungsperson waren von der Idee mit den Cards im Zusammenhang mit den Tischen angetan und gaben uns das Einverständnis, diese Ideen weiterzuentwickeln.

2.2.2 Wireframes

In einem nächsten Schritt wurden vorangegangene Gedanken anhand von Wireframes verfeinert. Frei nach dem Motto «Mobile First» wurden erste Views kreiert. Es ist einfacher, einen Mobile basierten View später auf eine Desktop-Version auszubauen. Denn wenn etwas auf einem kleinen Bildschirm Platz findet, dann passt es gezwungenermassen auch auf einen Grossen. Der umgekehrte Fall ist nicht immer so fix realisierbar.

Stream



- Card-Stream mit allen verfügbaren bzw. erstellten Tischen
- Stream ist leer, wenn man an keinem Tisch angemeldet ist bzw. keine eigenen Tische erstellt hat.
- Header passt den Namen des aktuellen Benutzers an.
- Ein Klick auf einen Tisch öffnet die Detailansicht. Selbst erstellte Tische können in der Detailansicht editiert werden. Tische an denen man nur angemeldet ist, können bezüglich der Anmeldung editiert werden.

Abbildung 2: Der Stream als Wireframe

Profil



Abbildung 3: Das Profil als Wireframe

- Der Benutzername ist frei wählbar, muss aber einmalig sein.
- Der persönliche Stream bzw. die Anmeldungen an fremden Tischen und die eigens erstellten Tische sind an den Benutzernamen gebunden.

Tisch erstellen

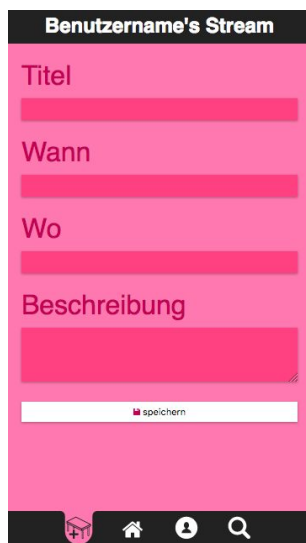


Abbildung 4: Tische erstellen als Wireframe

- Tische haben einen Titel (NOT_NULL), haben einen Zeitpunkt (Datum und Uhrzeit) (NOT_NULL), haben einen Standort (Restaurant so und so) und eine Beschreibung (für mehr Details).
- Mit einem Druck auf «speichern» schliesst sich der Dialog, man landet auf dem Stream wo der gerade erstellte Tisch sichtbar ist (gebunden an den aktuellen Benutzernamen)

Suche



- Öffnet einen Stream von Suchergebnissen
- Dieser Stream ist leer, wenn kein Suchbegriff eingegeben wurde.
- Suchbegriffe können sein: Benutzername oder Tisch-Titel

Abbildung 5: Die Suche als Wireframe

Des weiteren wurden die Cards nochmals separat ausgearbeitet.

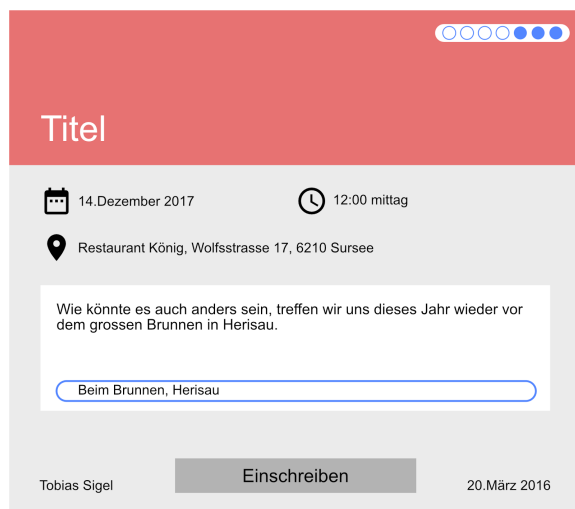


Abbildung 6: Card (nicht eingeschrieben)

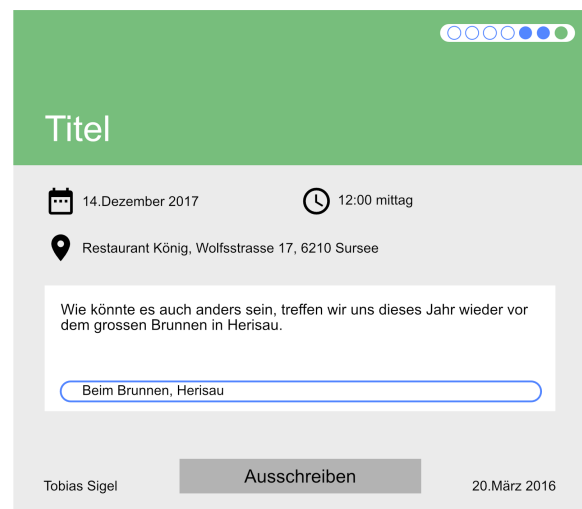


Abbildung 7: Card (eingeschrieben)

Es gibt drei Arten von Cards bzw. Tische: Solche die man selbst erstellt hat, Tische an denen man sich eingeschrieben hat (vgl. Abbildung 7) und Tische zu welchen man momentan keinen Bezug hat (vgl. Abbildung 6). Als Ersteller hat man nicht die Möglichkeit, sich bei einer seiner Tische ein- oder auszuschreiben. Der Button in der Mitte unten ist dabei nicht sichtbar. Ein Tisch der noch freie Plätze bietet und bei dem man weder eingeschrieben noch der Ersteller ist, kann man sich über den Einschreiben-Button registrieren. Über den Indikator oben rechts, gekennzeichnet mit einem grünen Punkt, hat man eine optische Bestätigung für die Registration. Der Einschreiben-Button wird zudem zum Ausschreiben-Button.

2.2.3 Designentscheidungen während der Entwicklung

Zu Beginn des Projekts wurde noch intensiv an einem JavaFX-Client gearbeitet. Bei diesem wurde in erster Linie versucht, einen Stream aus Tischen auf die Reihe zu bekommen. Ein Menü wurde im Header der Applikation verbaut (vgl. Abbildung 8). Die View vom JavaFX-Client wurde allerdings nicht weiterentwickelt und es wurde lediglich noch das Anzeigen der Teilnehmer implementiert. Später war der JavaFX-Client gar kein Bestand mehr vom Projekt.

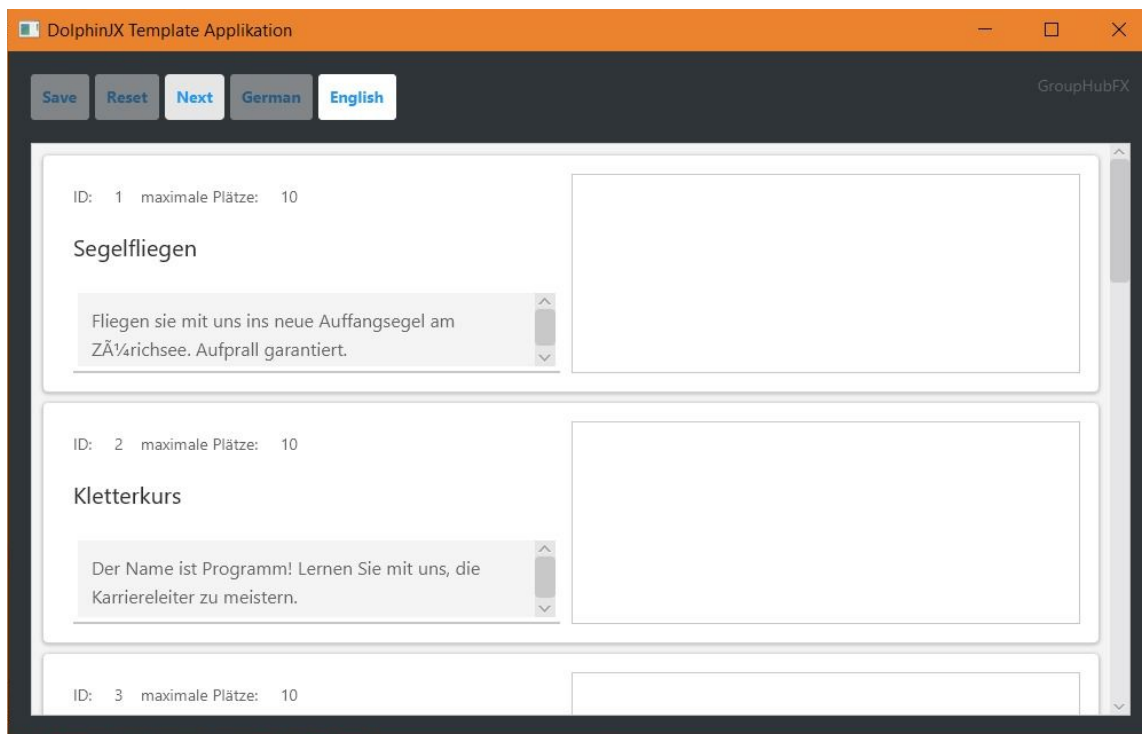


Abbildung 8: JavaFX-Client im Entwicklungsstadium

Auf Wunsch des Projekt-Teams wurde anfang November neben der Umsetzung eines JavaFX-Clients, ebenfalls noch ein JavaScript-Client in Angriff genommen. Zuerst als Mobile-Applikation gedacht, bestanden Kunde und Couch im weiteren Verlauf des Projekts darauf, sich im Webbereich ausschliesslich auf eine Desktop-Variante zu konzentrieren.

Der JavaScript-Client, wie er in Abbildung 9 dargestellt ist, enthält in der Desktop-Variante auf der linken Seite eine aufklappbare Menüleiste. Über diese navigiert man zwischen den drei verschiedenen Streams: Tische die es noch zu entdecken gibt, Tische bei denen man angemeldet ist und Tische welche man selbst erstellt hat. Ganz unten im Menü sieht man, wer aktuell eingeloggt ist (mittels Bild, wenn das Menü eingeklappt ist und mit zusätzlichem Namen bei aufgeklappter Menüleiste). Dieser Menüpunkt enthält ausserdem einen Link, welcher in den Editiermodus des Profils führt. Über den Link unten rechts, in der Abbildung ein runder grüner Kreis mit einem Plus in der Mitte, kann man einen neuen Tisch anlegen. Im Hintergrund wurden die Cards bereits nach Vorlage (vgl. Abbildungen 6 & 7) umgesetzt. Allerdings sind diese noch nicht korrekt skaliert und die Indikatoren für die Anzahl freier Plätze, hier schwarze Balken, funktionieren noch nicht richtig.

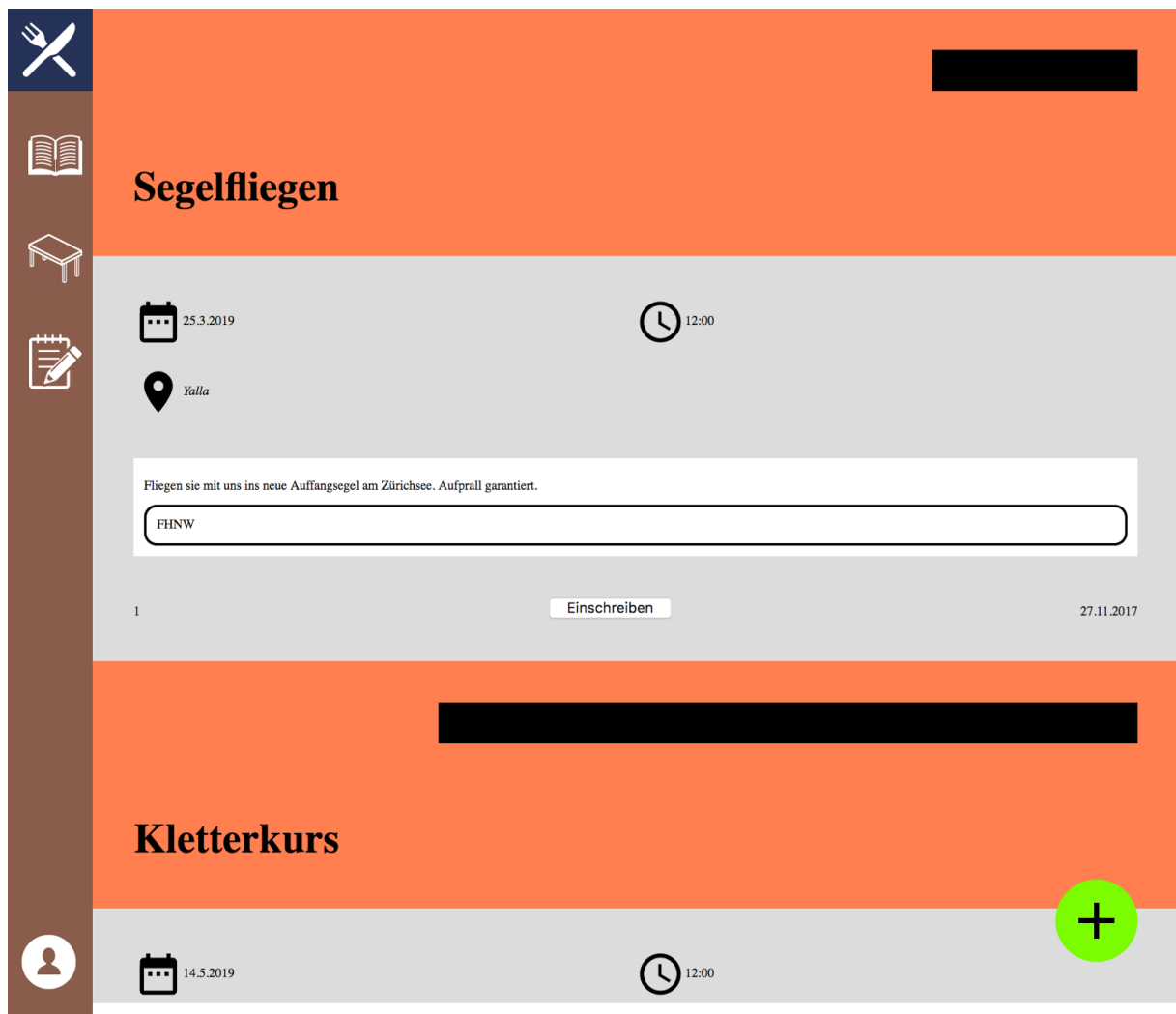


Abbildung 9: JavaScript-Client im Entwicklungsstadium

Die Idee einen Tisch als digitale Verkörperung für eine Verabredung zu verwenden, wurde sogar noch weiter gestrickt. Wenn man eine kleine Vektorgrafik in Verbindung mit JavaScript-Code bringen kann, in diesem Fall der Indikator oben rechts, dann funktioniert das auch mit einer grossen Grafik, zum Beispiel mit einem kompletten Restaurant. Anstelle der Cards könnte man verschiedene Restaurant-Ausschnitte darstellen, in welchen alle Informationen als Mini-Grafik ersichtlich sind. Alle kleinen Vektorgrafiken zusammen ergeben ein Gesamtbild. So wurden Tische, Stühle und ein kleines Restaurant designt.

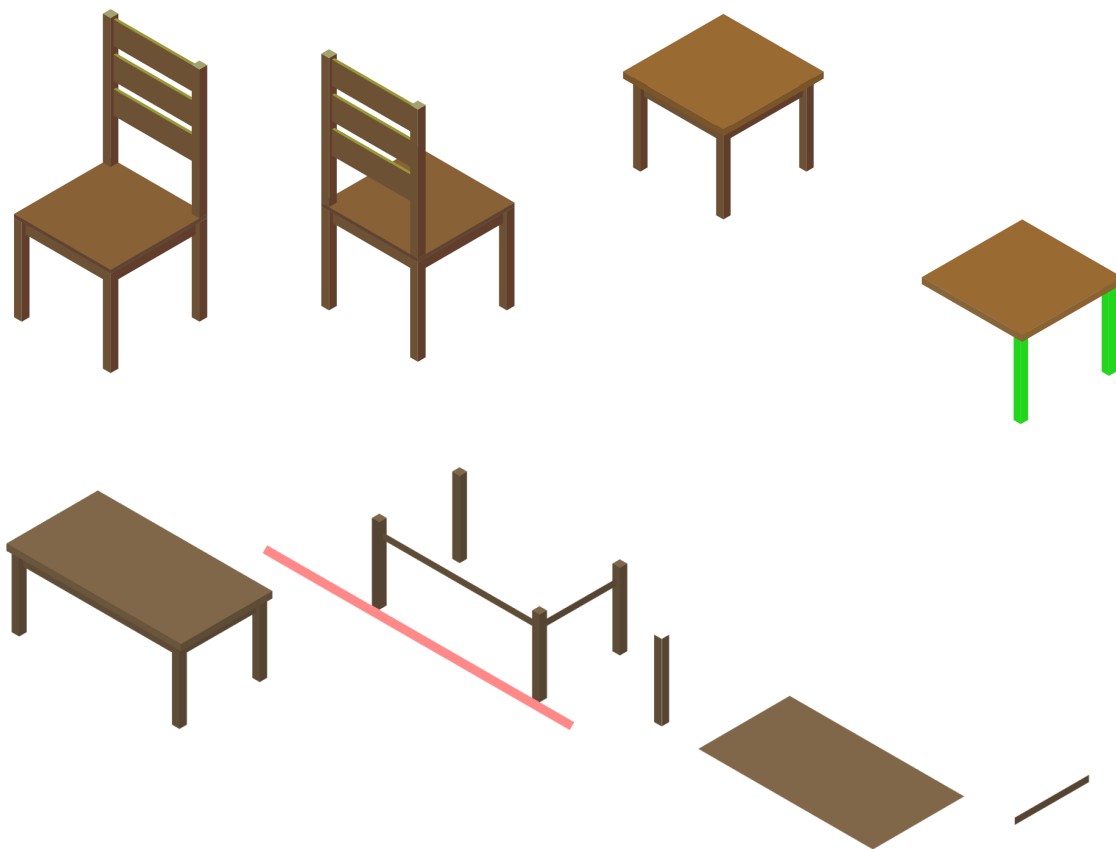


Abbildung 10: Vektorgrafikentwürfe von Stuhl & Tisch

Alle Grafiken wurden als SVG erstellt, sodass man diese einfach ins Web einbinden und mittels JavaScript-Code verändern hätte können. Der Tisch wurde dabei so konstruiert, dass sich die Tischplatte mit einer einfachen Längenangabe verändern liess. Die Tischbeine müssten um denselben Faktor verschoben werden. So wird aus einem Zweiertisch schnell ein beliebig langer Tisch. An diesem könnten eine gewisse Anzahl Stühle platziert werden, welche die maximale Anzahl verfügbarer Plätze darstellen. Ist der Tisch vor dem Stuhl gedeckt, würde das bedeuten, dass dieser Platz bereits besetzt und somit nicht mehr verfügbar ist. Eine Card könnte dann wie folgt aussehen:

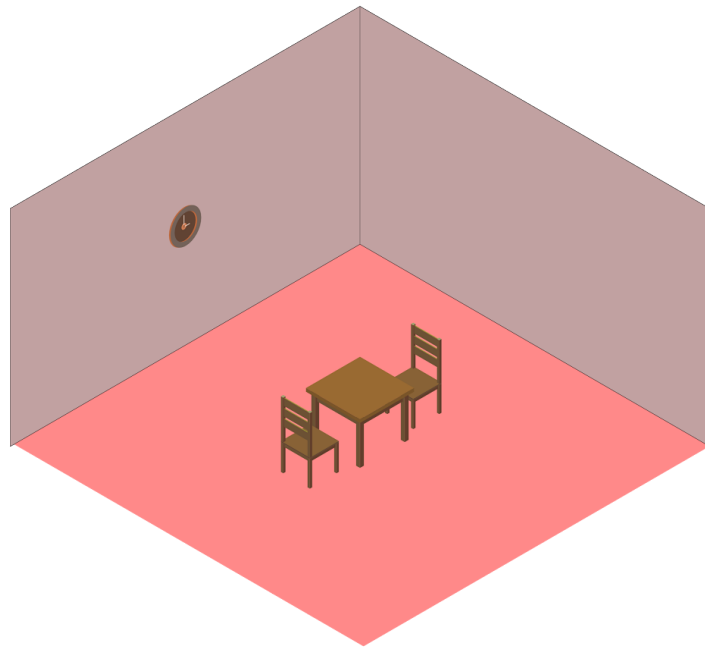


Abbildung 11: Ausschnitt aus einem Restaurant

Die analoge Uhr an der Wand zeigt die vereinbarte Zeit an, wann das Essen stattfindet. Die Tischgrösse und die Anzahl Stühle sind wie erwähnt variabel. Ein Wandteppich könnte den Ort des Treffens bzw. den Namen des Restaurants verraten. Eine aufgestellte Kreidetafel enthielte nähere Details zum Essen. Leider konnte diese Idee aus Zeitgründen nie zu Ende entwickelt werden.

Nach der Entscheidung, den JavaFX-Client einzustellen, wurde parallel zum JavaScript-Client mit der Entwicklung an einer nativen Android-Applikation begonnen. Aus diesem Grund, und weil man später den JavaScript-Client ebenfalls aufgegeben hat, wurden die ursprünglichen Designentwürfe zwangsläufig über Bord geworfen und man verwendete die Material-Designvorlage von Google.

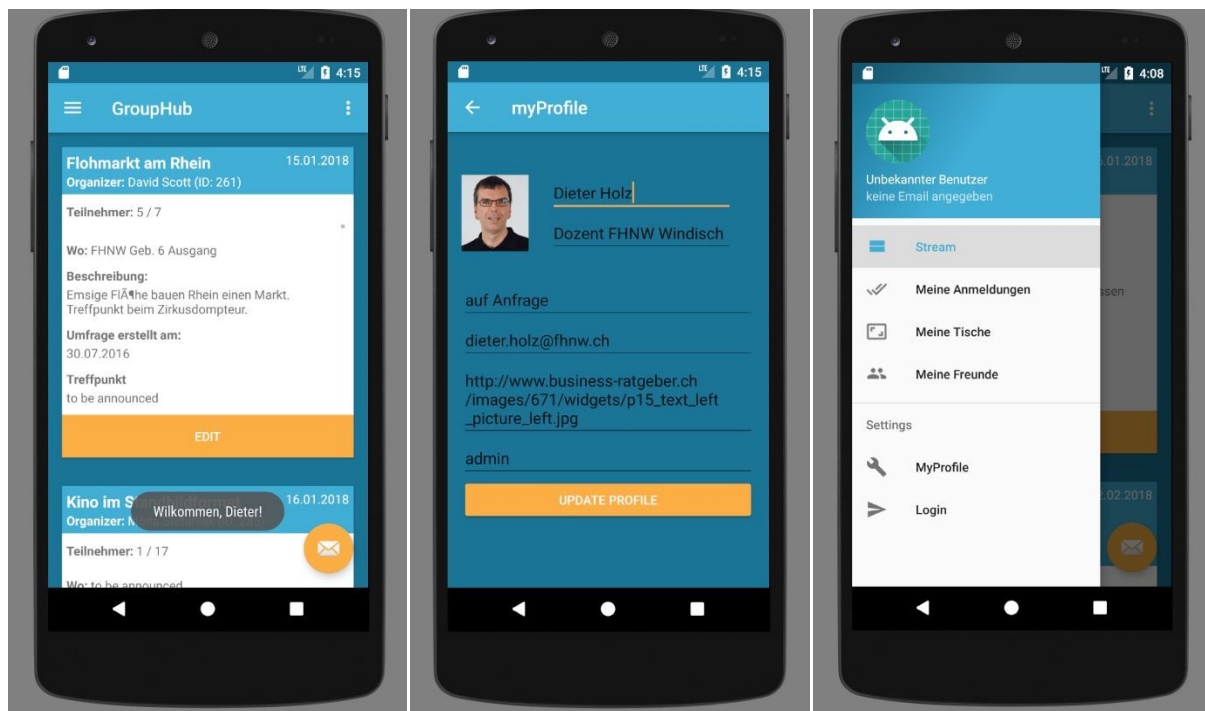


Abbildung 12: Endentwurf im Material-Design

Das Android typische Design strotzt geradezu vor Langweile, war jedoch in diesem Falle sehr nützlich, da es mit unzähligen Vorlagen einen schnellen Einstieg mit akzeptablem Design in die Android-Welt bot. Nichts desto trotz hat man die einzelnen Grundelemente so beibehalten, wie sie geplant waren. Auch beim Android-Client gibt es den bereits bekannten «Stream of Cards» in den drei verschiedenen Varianten «Stream» (alle Tische), «Meine Anmeldungen» (Tische an denen man Platz genommen hat) und «Meine Tische» (selbst erstellte Tische). Über die Seitennavigation wechselt man zwischen diesen Streams. Neben diesen findet man dort auch den Zugang in die Profilsicht, einen Login- bzw. Logout-Button und den aktuell eingeloggten Benutzer.

2.2.4 Abschliessendes zum Design

Ideen wären viele vorhanden gewesen, das beweisen die vorgängigen Unterkapitel. Allerdings muss man hier klar eingestehen, auf zu vielen Baustellen gewerkelt zu haben. Dies lag nicht zuletzt auch daran, dass mit unterschiedlichen Client-Technologien gearbeitet wurde. Und diese unterschiedlichen Clients wurden mehr oder weniger in Einzelarbeit anstelle im Team erstellt. So hatte der eine Ideen die jeweils nur auf seiner Plattform schnell und einfach umsetzbar waren. Der Andere hatte das Nachsehen und entschied sich wiederum für eine eigene Variante. Wie diese Fragmentierung hätte vermieden werden können, ist an dieser Stelle schwierig zu sagen. Ein von Anfang an ausgeprägteres Grunddesign, hätte vielleicht geholfen, die eine oder andere Designentscheidung zu erleichtern. Ein komplexeres einheitliches Design wäre jedoch in

dem zeitlich vorgegebenen Rahmen nahezu unmöglich gewesen, zumal sich im Verlauf des Projekts die Wahl der Technologie geändert hat.

2.3 GitHub & Wiederherstellung der Projekte

Über GitHub wurde nahezu die gesamte Projektorganisation verwaltet. Neben dem Source-Code-Backup und -Versionierung benutzte man den Onlinedienst für den Austausch von Fragen mit Coach und Kunde, sowie als Todo-Liste und Gesamtübersicht der Projektabläufe.

2.3.1 Issues & Projects

In den ersten beiden Wochen nach Projektstart wurde noch intensive mit Trello gearbeitet, einer webbasierten Projektmanagementsoftware. Für das Erfassen von Arbeitszeiten hat man toggl.com eingesetzt. Der Austausch von Files und Dokumentation fand über OneDrive, einem Cloud-Dienst von Microsoft, statt. Schnell wurde klar, mit so vielen verschiedenen Diensten ist es fast unmöglich die Übersicht zu behalten. Da man für die Versionierung vom Code Git verwendete und der Coach sogar ein privates GitHub-Repository zur Verfügung stellte, war schnell klar, dass alles nach GitHub verlegt werden soll. In einem ersten Schritt wurden alle Todos von Trello unter den Issues bei GitHub erfasst. Jedes Issue hat ein Label und einen Verantwortlichen. Issues können direkt per Git über eine Commit-Message referenziert oder geschlossen werden. Unter «Projects» können diese Issues auf einem kanban-ähnlichen Board organisiert werden. Die hilft enorm dabei, die Übersicht zu behalten.

2.3.2 Code auf GitHub

Die Verwaltung des Source-Codes via GitHub verlief bis zum Schluss holprig. Da die Studierenden beide eine andere Vorstellung davon hatten, wie man mit Git arbeitet, kam es hier und da zu Komplikationen. Teils Fehlentscheidungen ergaben sich auch durch das mangelnde Wissen im Zusammenhang mit Git. Kam noch die nicht gerade alltägliche Situation hinzu, in der man mehrere unterschiedliche Projekte in einem Git-Repository zu verwalten hatte. Dies resultierte in unzähligen unterschiedlichen Branches, bei welchen am Schluss niemand mehr genau wusste, inwiefern diese wichtig sind. Es wurde viel zu wenig committed und eben so wenig auf den Server gepusht. Die Commit-Messages sind teilweise unklar oder verwirrend. Es gab sogar den unglücklichen Fall, dass man einander Code überschrieben hatte. Solche Missverständnisse und Fehler müssen in Zukunft unbedingt vermieden werden. Aus diesem Grund wird vorgeschlagen, zu Beginn eines weiteren Projekts die Regeln für die Benutzung von Git schriftlich festzuhalten. Punkte wie: Eindeutig verständliche Namen für Branches, eine klare Hierarchie der Branches, die Notwendigkeit von möglichst

vielen Commits und Pushes, sowie eine gute Kommunikation im Team, was wie wo gemacht wurde, müssen unbedingt vorgängig diskutiert und dokumentiert werden. Nach einer kleinen Aufräumaktion am Schluss dieses Projekts, ist der Code wie folgt organisiert:

- Die definitiven Branches beginnen mit AAA (aus dem einfachen Grund, damit diese in der Liste auf GitHub an oberster Stelle erscheinen und man nicht lange suchen muss)
- Der Branch «AAA_develop_javascript» enthält einen einfachen OpenDolphin Server und eine unfertige, aber lauffähige Version eines JavaScript-Clients.
- Der Branch «AAA_final_android» enthält ein Android Studio Projekt mit dem aktuellen zur Abgabe bestimmten Android-Client.
- Im Branch «AAA_final_server_open_dolphin» ist die passende OpenDolphin Server Applikation zum Android Client abgelegt. Auch hier handelt es sich um die aktuellste Version welche für die Abgabe bestimmt ist.

2.3.3 Wiederherstellung der Projekte

Arbeitsumgebung einrichten:

- Java 8 JDK:
<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
- Git:
<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>
- Java IDE ihrer Wahl, empfohlen wird IntelliJ:
<https://www.jetbrains.com/idea/download/#section=windows>
- AndroidStudio 3.0+. Es wird später ein VirtualDevice mit API-Level 26+ benötigt.
<https://developer.android.com/studio/index.html>
- Gradle:
<https://gradle.org/install/>

2.3.3.1 Server

API-Level 1.8

1. Erstelle einen lokalen Ordner mit dem Namen «GroupHubRocks» in deinem Workspace. Ausrufezeichen sind leider nicht erlaubt, wir haben es probiert.
 - a. Erstelle zwei weitere Ordner innerhalb des «GroupHubRocks»-Ordners. Am besten passen die Namen «Server» und «Android».
2. Führe mit «Shift+Rechtsklick» im Ordner «Server» Git-Bash aus (oder starte cmd).
3. Erstelle ein neues lokales Git-Repo mit `git init`.

4. Füge dem lokalen Repo unser GitHub Remote-Repo hinzu:
`git remote add origin`
<https://github.com/FHNW-IP5-IP6/GroupHubFX.git>
5. Lade den Branch «AAA_final_server_open_dolphin» herunter:
`git pull origin AAA_final_server_open_dolphin`
6. Wechsle auf den Branch «AAA_final_server_open_dolphin»
`git checkout AAA_final_server_open_dolphin`

→ Das Projekt ist nun im Ordner «Server» abgelegt und kann von IntelliJ importiert werden. Es wurden folgende Einstellungen verwendet:

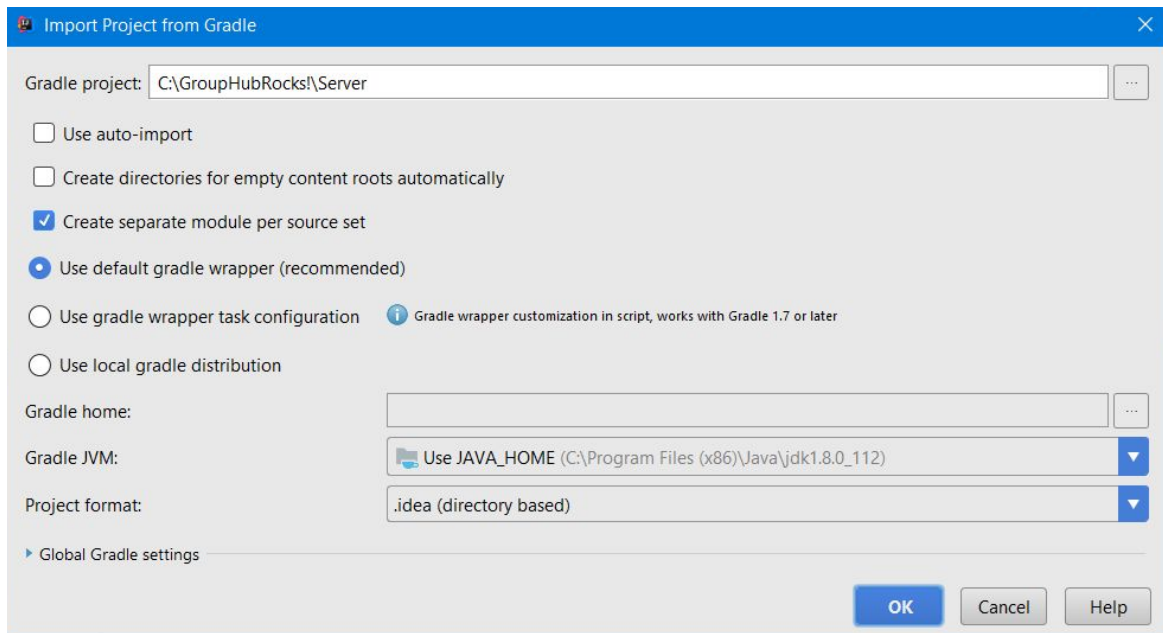


Abbildung 13: Einstellungen für Projektimport in IntelliJ IDEA

→ Der Server kann jetzt gestartet werden:

- Führe `gradlew install` in der Kommandozeile des Projektes aus (alt+F12) aus.
- Führe das Kommando `gradlew jettyRun` aus

2.3.3.2 Android

Es wird ein Emulator-Image mit API-Level 26+ benötigt. Es wird auch die Installation von «Haxm» auf dem Image (wird im Installer gefragt) empfohlen. Source- und Target Compatibility sind beide 1.8 (genau gleich wie bei dem Server).

- Wiederhole die Schritte 2 bis 4, aber im Ordner «Android».
- Lade den Branch «AAA_final_android» herunter:
`git pull origin AAA_final_android`
- Wechsle in den Branch «AAA_final_android»:
`git checkout AAA_final_android`

→ Das Android-Projekt ist nun im Ordner «Android» abgelegt und kann von Android Studio importiert werden.

→ Ist ein Emulator-Image mit API-Level 26+ installiert, so kann man jetzt die Applikation starten:

- Klicke in Android Studio auf das grüne Play-Symbol in der oberen Toolbar.
- Wähle das Image mit API-Level 26+ aus und bestätige dieses.

Hinweis: Es kann sein, dass die «[Intel Virtualization Technology](#)» im BIOS ihres Computers aktiviert werden muss. Die Einstellungen dazu können pro Hersteller unterschiedlich. Die Aufarbeitung des Android-Projektes dauert beim ersten Start ein wenig.

Die Login-Daten für die Android-Applikation sind wie folgt:

Dierk Koenig, Dieter Holz, Mario Winiker, Tobias Sigel → mit dem PW «admin»
alle anderen in der Applikation ersichtlichen Benutzer → mit dem PW
«GroupHubRocks!»

2.4 Source Code

Im Folgenden werden die Design- sowie Architekturentscheidungen im Hinblick auf die GroupHub-Applikation zusammengefasst und dargelegt. Im Laufe des Projektes ergaben sich einige Änderungen, die sich aufgrund von neuen Erkenntnissen durchgesetzt haben.

2.4.1 OpenDolphin Server-Applikation

Die Struktur des Projektes ist in Abbildung 14 ersichtlich. Das Projekt entstand anfangs mit Hinblick auf die Erstellung eines JavaFX-Clients, später wurde dies auf den Android-Client ausgeweitet. Aus diesem Grund existieren hier verschiedene Module, die nun im Näheren erläutert werden.

2.4.1.1 Client

Dies ist das Modul für den JavaFX-Clients. Hier ist eine ältere, zum jetzigen Zeitpunkt nicht mehr lauffähige (da in der Zwischenzeit einige Änderungen am Server vorgenommen wurden), Version abgelegt. Hier konnten die von Herrn Dieter Holz generierten Veneer-Klassen (engl: Furnier) für einen einfacheren Zugriff auf die PM's benutzt werden. Diese Strategie wurde im Android-Clients benutzt, denn solche Veneers vereinfachen die Arbeit enorm. Mehr Informationen zu diesem Modul befinden sich auch im [Unterkapitel 2.4.2 JavaFX-Client](#).

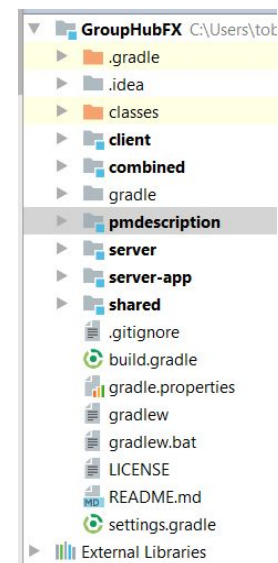


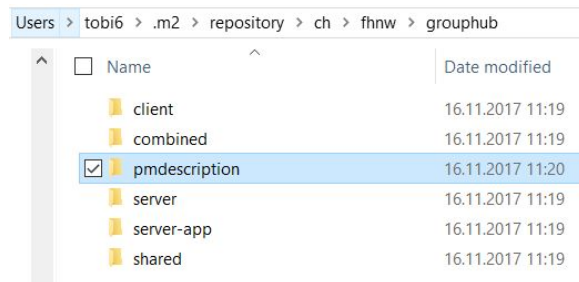
Abbildung 14:
Projektstruktur von
OpenDolphin

2.4.1.2 Combined

Das Modul beinhaltet die kombinierte Lösung von Server- und Applikationsstart und ist als solches eine Hilfe für den Programmierer. Dieses Modul haben wir im Laufe des Projektes nicht berücksichtigt.

2.4.1.3 Pmdescription

Pmdescription entstand erst im Rahmen der Android-Entwicklung. Hier wurden die jeweiligen Attribute pro Presentation Model festgelegt und die dazugehörigen Kommandos deklariert. Wie der Titel sagt, werden hier die PMs genauer beschrieben. Hier ist das Zusammenspiel mit Maven bzw. dem Gradle Wrapper interessant.



The screenshot shows a file explorer view of a local Maven repository. The breadcrumb path is 'Users > tobi6 > .m2 > repository > ch > fhnw > grouphub'. The table below lists the contents of the repository:

Name	Date modified
client	16.11.2017 11:19
combined	16.11.2017 11:19
<input checked="" type="checkbox"/> pmdescription	16.11.2017 11:20
server	16.11.2017 11:19
server-app	16.11.2017 11:19
shared	16.11.2017 11:19

Abbildung 15: Ausschnitt aus dem lokalen Maven-Repository

Um von anderen Applikation auf die API bzw. das Modul zugreifen zu können, kann man den Befehl `gradlew install` aufrufen. Dieser installiert sämtliche Module im lokalen Maven-Repository (vgl. Abbildung 15). Dies ermöglichte dem GroupHub-Projektteam von der Android-Entwicklungsumgebung auf die beschriebenen Attribute zuzugreifen, indem man das lokale Maven-Repository als Dependency für das Projekt angibt. Im Folgenden sehen wir in der Android-Applikation in der Klasse «TableVeneer», dass ein solcher Zugriff problemlos möglich ist:

```
public void setDescription(String DESCRIPTION {

    model.findAttributeByPropertyName(TableAtt.DESCRPTION.name()).setV
    alue(DESCRIPTION;
}
```

2.4.1.4 Server

Server stellt das für uns wichtigste Modul dar. Hier werden die meisten applikationsspezifischen Methoden implementiert. Das Modul ist nach folgender Struktur aufgebaut:

TableController
<pre>private final GroupHubService service;</pre>
<pre>private void loadSoonestTables(); private void loadTablesByOrganizer(); private void loadTablesByParticipant(); private void loadEmptyTable();</pre>

PersonController
<pre>private final GroupHubService service; static private PresentationModel user;</pre>
<pre>protected void initializeBasePMs(); private void loadParticipants(); private void loadOrganizers(); private void confirmLogin(); public static getUserID();</pre>

ParticipationController
<pre>private final GroupHubService service;</pre>
<pre>public void loadActiveParticipations();</pre>

Abbildung 16: Die Controller des Server-Moduls

Controller enthält einen Application-State-Controller, alle type-spezifischen Controller (Table, Person, Participation) sowie die «Reception»-Klasse, um diese zu registrieren. Die Controller führen Datenabfragen über (unten genauer spezifizierte) Service-Methoden aus, und generieren anhand der zurückgegebenen DTOs neue PMs. Kurz gesagt: Die Controller sind die einzigen Instanzen, welche PMs erstellen können. Folglich werden hier alle Methoden implementiert, welche aus Sicht des Clients benötigt werden, um alle notwendigen Daten in dem PM-Store zu laden. Dabei wurde die Reihenfolge dieser Instanziierung beim Start der Applikation wie folgt definiert:

Schritt	Beschreibung	Befehl	Effekt	INFO
1	<i>instanziiere alle Daten als DTO über Konstruktor von GroupHubRemoteService</i>			
2	lade die 20 zeitlich nächsten Tische	loadSoonestTables();	20 new PM's of Type "Table" in PMStore	PM Instanziierung basiert auf sortierter Liste --> Store ist bereits sortiert.
3	lade die dazugehörigen Organisierer	loadOrganizers();	X new PM's of Type "Person" in PMStore	geht für alle exist. Table-PM's durch alle DTO's und retourniert die passenden Personen. Personen können Organisatoren mehrerer Tische sein, weswegen nicht unbedingt 20 Personen zurückkommen.
4	lade die dazugehörigen Teilnehmer	loadParticipants();	X new PM's of Type "Person" in PMStore	Selbiges wie in Schritt 3, Personen können auch an mehreren Events teilnehmen.
5	lade die dazugehörigen Participation-PM's	loadParticipations();	X new PM's of Type "Participation" in PMStore	Damit der Client die Personen im PMStore den jeweiligen Tischen zuordnen kann, müssen die ParticipationPM's geladen werden.

Abbildung 17: Hierarchie der PM-Instanziierungen

Der Server dokumentiert dies in der Kommandozeile beispielsweise wie folgt:

```
[TableController]loadSoonestTables()-->loaded: 20 closest TablePM's
into PMStore.
[PersonController]loadOrganizers()--> loaded: 19 new PersonPM('s)
into the PMStore and skipped: 1 already existing PersonPM('s)
[PersonController]loadParticipants()--> loaded: 37 PersonPM's and
skipped:
23 already existing PersonPM's.
```

```
[ParticipationController]loadActiveParticipations-->loaded new
Participation
PM's into the PMStore.
```

Service enthält das Interface «GroupHubService» sowie dessen konkrete Implementation «GroupHubRemoteService», welche im Wesentlichen die Logik zur Instanziierung und Abfrage von Beispieldaten ermöglicht. Die Klasse «GroupHubRemoteService» arbeitet ausschliesslich mit DTOs.

GroupHubRemoteService (impl)
<pre>List<DTO> tables; List<DTO> persons; List<DTO> participations; (List<DTO> friends;)</pre>
<pre>//retrieve Data from existing Lists<DTO> public List<DTO> findSoonestTables(int amount); public List<DTO> findActiveParticipations(String tableID); public DTO findPersonByID(String ID); public DTO findPersonByName(String name); public DTO findTablesByOrganizerID(String organizerID); public DTO createEmptyTableDTO(String organizerID); public DTO createEmptyPersonDTO(); //instantiate Data and store in the Lists<DTO> private void initRandomParticipations(int amount); private void initRandomTables(int amount); private void initRandomUsers(int amount); private void initPowerUsers();</pre>

Abbildung 18: Ausschnitt aus dem GroupHubRemoteService

Diese Strategie wurde verfolgt, um eine möglichst realitätsnahe Simulation eines Servers zu bekommen. Wenn der Server mit `gradlew jettyRun` gestartet wird und ein Client eine Connection erfolgreich aufbaut, wird eine Instanz von «GroupHubRemoteService» generiert, welche durch den Konstruktoraufruf eine endliche Menge an festdefinierten (im Unterschied zu laufzeitgenerierten Random-Objekten) DTOs in den dazugehörigen Listen abspeichert. Dadurch befindet sich immer eine feste Anzahl an konkret definierten DTOs im Speicher. Die Kommandozeile informiert uns dabei über die jeweilige Instanziierung:

```
:server-app:jettyRun
[GroupHubRemoteService]initiated PowerUser Tobias Sigel
[GroupHubRemoteService]initiated PowerUser Mario Winiker
[GroupHubRemoteService]initiated PowerUser Dieter Holz
[GroupHubRemoteService]initiated PowerUser Dierk Koenig
[GroupHubRemoteService]initiated 100 random users.
[GroupHubRemoteService]initiated 100 random tables.
[GroupHubRemoteService]initiated 300 random participations.
[TableController]loadSoonestTables()-->loaded: 19 closest T:
```

Abbildung 19: Ausschnitt aus dem Logging bei der Erstellung von Personen

Zusätzlich dazu sind das Projektteam und der Kunde als Power-User definiert. Der Unterschied ist, dass diese bei jedem Applikationsstart immer genau gleich instanziiert werden, während die Zufalls-DTOs beim Start aus willkürlich ausgewählten Daten zusammengesetzt werden. Im Gegensatz zu zur Laufzeit generierten Zufalls-DTOs kann man mit dieser Strategie jetzt effektiv nach Personen, Tischen und Teilnahmen im Speicher suchen und diese dann modifizieren (und später wieder finden).

Servlet ist extrem klein und bietet lediglich eine Methode um mögliche Services zu registrieren. Da es nur einen Service gibt, ist dies eine relativ uninteressante Sache. Hier ist im Rahmen von GroupHub nichts Neues gemacht worden.

Util beinhaltet eine abstrakte Controller-Klasse, welche die konkreten Controller erweitert sowie das Interface DTOMixin', welches eine Sammlung an guten Methoden zum Gebrauch anbietet, wie etwa die im Projekt oft benutzte `getSlot(DTO, AttributeDescription)` Methode.

2.4.1.5 Server-app

Hier ist im Git-Branch «AAA_develop_javascript» der JavaScript-Client abgelegt.

2.4.1.6 Shared

Im [Unterkapitel 2.4.1.1](#) ist erstmals die Rede von Veneer-Klassen. Diese bilden eine Art Schnittstelle zwischen komplizierten Datensatzanfragen und Programmierer, denn sie vereinfachen dies, sodass sie für den Programmierer einfacher und verständlicher nutzbar sind. Der Code wird dadurch lesbarer, kleiner und weniger fehleranfällig. Würde der JavaFX-Clienten noch genutzt werden, so würden man die Veneer Klassen in diesem Verzeichnis ebenfalls noch nutzen. Da der Android-Client Vorrang hatte und Android keine JavaFX-Attribute kennt, wurden diese Veneers jedoch obsolet.

2.4.2 JavaFX-Client

Der JavaFX Client wurde ab Mitte des Projektes nicht weiter verfolgt. Der existierende Prototyp gab GroupHub die Gelegenheit, mit OpenDolphin und einer bekannten API (JavaFX) zu experimentieren. Die Projektstruktur ist sehr klein und schnell zu erfassen. «MyRemoteStarter» ist zuständig für die Verbindung von diesem Modul zum OD-Server. Ausserdem startet man von hier die Applikation. Die Klasse «MyAppView» sendet beim Start der Applikation das Kommando zur Instanziierung der Base-PMs und baut das UI bei der Beendigung des Kommandos auf.

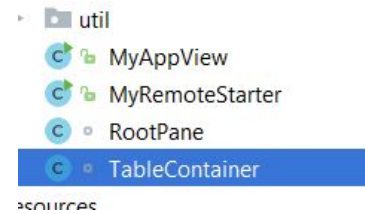


Abbildung 20: Ausschnitt aus der Projektstruktur von OpenDolphin

In «RootPane» passiert die gesamte Logik ausserhalb der Tische; hier werden die Buttons erstellt, das Logo gesetzt und das Layout definiert. Ausserdem wird hier ein Pattern verwendet, welches später im Android-Client wieder seinen Nutzen findet: das «ViewHolderPattern». Die Klasse «TableContainer» ist genau Selbiges, ein sogenannter ViewHolder, der die gegebenen Datensätze adequat präsentiert.

Übrigens: Der JavaFX-Client benutzt das Material-Design. Obwohl das schon ziemlich toll gefunden wurde, wäre in Zukunft die Benutzung des JavaFX-Scene-Builders noch besser.

2.4.3 JavaScript-Client

Wo was ist

Um an den Code des JavaScript-Client zu gelangen, wird die OpenDolphin Server-Applikation aus dem Git-Repositorys im Branch «AAA_develop_javascript» benötigt.

Im Verzeichnisbaum unter server-app/src/main/webapp sind alle relevanten Dateien des JavaScript-Clients abgelegt. Dazu gehören ein Ordner «js» mit allen JavaScript-Files, ein Ordner «css» mit den entsprechenden Stylesheets, ein Ordner «WEB-INF» mit dem Deployment Descriptor «web.xml» und eine HTML-Datei «index.html» sowie zwei temporäre HTML-Dateien, welche zur Unterstützung bei der JavaScript-Entwicklung dienen.

index.html

Nachdem man den Jetty-Server auf seinem Computer gestartet hat (gradle jettyRun) und in einem Browser auf demselben Gerät <http://localhost:8080/myApp> aufruft, wird als erstes das Dokument «index.html» zurückgeliefert.

Im <head>-Tag des Dokuments wird zum einen der Titel des HTML-Dokuments festgelegt. Zudem werden logische Links zu den Stylesheets gesetzt und die clientseitigen Skripts eingebunden. Der Body des HTML-Dokuments enthält das Tag

<nav> mit der kompletten Seitennavigation, ein <main>-Tag in welches der eigentliche Inhalt der Seite geschrieben wird, sowie ein <script>-Tag mit dem JavaScript-Code um die Verbindung mit OpenDolphin herzustellen und wenn diese erfolgreich aufgebaut wurde, die ersten Cards zu laden.

Die Navigationsleiste besteht zum grössten Teil aus einer unsortierten Liste von Listenelementen. Diese enthalten jeweils ein Textlabel und ein Icon als SVG. Auf dem Listenelement ist ein OnClick-Event-Handler registriert. Dieser ruft die jeweils entsprechende Funktion auf, um das <main>-Tag zu befüllen. Das <main>-Tag dient in diesem Fall lediglich als Container für den Inhalt der Hauptfläche von der Webseite.

(Wolf 2015)

Im Script-Teil wird zuerst ein neuer Dolphin aufgesetzt und über die Adresse des OpenDolphin Servers damit verbunden. Der serverseitige und der clientseitige Model Store sind nun synchron zueinander. Danach wird der clientseitige Store zurückgesetzt, um sicherzustellen, dass sich keine alten Daten bzw. Presentation Models darin befinden. Gelingt dies, wird ein Kommando `loadAllTables` abgesetzt. Serverseitig bewirkt dieses Kommando, den Aufruf der Methode `loadAllTables` im Table Controller, welcher sich unter `/server/src/main/java/myapp/controller/` befindet. Diese Methode wiederum löst die Erstellung einer Liste aus DTOs gefüllt mit Beispieldaten in der Klasse «SomeRemoteService»

(`/server/src/main/java/myapp/service/impl/`) aus und erstellt damit, in Verbindung einer Definition von Table

(`/pmdescription/src/main/java/myapp/presentationmodel/PMDescription.java`), pro Element ein Presentation Model. Diese werden in den Model Store gespeichert und stehen auf der Clientseite zur Verfügung.

```
dolphin.reset({
  onSuccess: function () {
    dolphin.send("myapp.presentationmodel.table
      .TableCommands.loadAllTables", {
      onFinish: (models) => {
        buildTables(models)
      }
    });
  }
});
```

Im JavaScript-Client wird, nach dem erfolgreichen Laden der Tische, ein Array von Presentation Models vom Typ «Table» als Parameter an eine Funktion `buildTables()` übergeben. Diese Funktion befindet sich in der JS-Datei «tables.js» und erstellt den initialen Stream aus Cards.

tables.js

In einem ersten Schritt wird über das Objekt `document` im DOM-Baum das Element mit der ID 'main-view' herausgepickt und einen Verweis darauf als Konstante abgespeichert. Das vereinfacht den mehrfachen Zugriff auf diesen Knoten, spart Schreibarbeit und führt zu einer schnelleren Ausführung des Programms. Mit Knoten ist hier ein HTML-Elementknoten gemeint, das heisst ein `node`-Objekt, welches ein HTML-Element im DOM repräsentiert. In diesem Fall das main-Element aus dem Body von «index.html».

```
const mainView = document.getElementById('main-view');
```

Des weiteren wird geprüft, ob der main-Knoten ein Kindknoten enthält bzw. ob darin weitere Elemente vorhanden sind oder ob dieser leer ist. Sollte mindestens ein Kindknoten vorhanden sein, wird dieser, bzw. derjenige welcher an oberster Stelle steht, gelöscht. Dieser Vorgang wiederholt sich gegebenenfalls, bis keine Kindknoten mehr vorhanden sind, das heisst das main-Element leer ist.

```
while (mainView.hasChildNodes()) {
  mainView.removeChild(mainView.firstChild);
}
```

Nun wird mit `document.createElement('div')` ein neues `div`-Element erstellt und mit Klassen- sowie ID-Attribut versehen. Über die Attribute kann das Element später per CSS angepasst oder per JavaScript direkt angesprochen werden.

```
const divTableStream= document.createElement('div');
divTableStream.setAttribute('class', 'table-stream');
divTableStream.setAttribute('id', 'table-stream');
```

Jetzt wird `models` gemapped. Bei `models` handelt es sich um den übergebenen Parameter und damit das Array, welches alle Tische enthält. Mit der Methode `map()` wird auf jedes Element des Arrays die übergebene Funktion angewendet. Es wird also pro enthaltenem `table`-Objekt ein `article`-Element erstellt und mit Klassen- sowie ID-Attribut versehen.

```
models.map((model) => {
  const articleTable = document.createElement('article');
  articleTable.setAttribute(
    'class',
    `table table-${model.attributes[0].value.toString()}`
  );
  articleTable.setAttribute(
    'id', `table-${model.attributes[0].value.toString()}`
  );
});
```

```

    articleTable.innerHTML = ' ... ';

    divTableStream.appendChild(articleTable);
  });

```

Um die Klassen- und ID-Attribute jedes einzelnen Artikels dynamisch zu erstellen, hat man Template-Strings eingesetzt, welche mit ECMAScript 6 eingeführt wurden. Die Template-Strings werden mit Back-Ticks (``) eingeschlossen und können Platzhalter beinhalten, die durch das Dollarsymbol gefolgt von geschweiften Klammern gekennzeichnet sind. (Mozilla und einzelne Mitwirkende 2017) Bei den Platzhaltern greift man auf die opendolphininternen IDs der Tische zurück.

Mit innerHTML wird der Inhalt jedes article-Elements befüllt. Der Inhalt sind die Elemente einer Card. Diese werden inklusive HTML-Markierungen übernommen. Hier wird auch ersichtlich, wozu die beiden Template-HTML-Dateien «temp_create-table.html» und «temp_table.html» dienen. Diese beinhalten den rohen HTML-Code, welcher für die article-Elemente gedacht sind, und wurden zur Vorschau für die CSS-Generierung verwendet.

Zum Schluss wird das fertige article-Element als Kindknoten dem div-Element angehängt. Das div-Element wiederum wird in das main-Element eingefügt und die Funktion buildButtonCreateTable() aus «create-table.js» wird aufgerufen.

create-table.js

Die Funktion buildButtonCreateTable() prüft als erstes, ob im HTML-Dokument bereits ein Button vorhanden ist, mit dem man neue Tische erstellen kann. Die Rede ist hier vom grünen Plus-Button (vgl. Abbildung 9 aus dem [Unterkapitel 2.2.3](#)). Existiert dieser Button noch nicht, wird er erstellt. Das Vorgehen unterscheidet sich dabei nicht vom Vorhergehenden, wo die Cards erstellt und dem main-Element angehängt wurden.

```

function buildButtonCreateTable() {
  let buttonCreateTable =
    document.getElementById('create-table__button');

  if (buttonCreateTable === null) {
    const mainView = document.getElementById('main-view');

    buttonCreateTable = document.createElement('button');
    buttonCreateTable.setAttribute(
      'class', 'create-table__button'
    );
    buttonCreateTable.setAttribute(

```

```

        'id', 'create-table__button'
    );

    buttonCreateTable.innerHTML =
        '<svg class="create-table__btn-plus-icon"
        viewBox="0 0 24 24"
        xmlns="http://www.w3.org/2000/svg">\n' +
        '<path d="M19 13h-6v6h-2v-6H5v-2h6V5h2v6h6v2z"/>\n' +
        '<path d="M0 0h24v24H0z" fill="none"/>\n' +
        '</svg>\n';

    buttonCreateTable.addEventListener(
        'click', buildFormCreateTable
    );

    mainView.appendChild(buttonCreateTable);
}
}

```

An zweitletzter Stelle des obigen Codes wird ein Event Listener auf dem Button registriert. Dieser hört auf einen Mausklick und führt umgehend die Funktion `buildFormCreateTable()` aus.

Diese Funktion befindet sich in der gleichen Datei direkt unter der oben erwähnten Funktion und erstellt, wieder nach genau dem selben Prinzip wie bei den anderen Funktionen, ein HTML-Element, welches in das main-Element eingebettet wird. Es handelt sich dabei um das Formular für die Erstellung von neuen Tischen. Das Formular wurde leider nicht mehr ausgebaut.

Andere JavaScript-Dateien

Neben den oben Erklärten enthält das Projekt noch vier weitere JS-Dateien, auf welche hier nicht speziell darauf eingegangen wird. Zum einen sind das die Dateien «my-chairs.js», «my-tables.js» und «table-stream.js», welche früher oder später dazu gedacht gewesen wären, gefilterte bzw. spezifische Streams auszugeben. Und zum anderen ist das die Datei «opendolphin.js», womit man OpenDolphin in ein Webprojekt einbinden kann und damit auch im Web die gleichen Vorteile und den gleichen Funktionsumfang wie bei einem Desktop-Client hat, das heisst die serverseitige Applikationslogik von OpenDolphin nutzen kann.

Stylesheets

Speziell erwähnenswert ist hierbei der Style der Navigationsleiste. Die entsprechenden Codezeilen von dem folgenden Ausschnitt befinden sich in der Datei «index.css» im Package «css».

```
.navigation {
  position: absolute;
  left: 0;
  z-index: 2000;
  width: 66px;
  height: 100%;
  display: flex;
  flex-direction: column;
  justify-content: space-between;
  background-color: #8A5C4C;
  padding-bottom: 1rem;
  transition: width 400ms
    cubic-bezier(0.165, 0.840, 0.440, 1.000) 0s;
}

.navigation:hover {
  background-color: #253358;
  width: 250px;
}
```

Über den Klassennamen wird das nav-Element aus dem Body von «index.html» angesprochen, wird das Element **absolute** positioniert und somit aus dem Dokumentenfluss herausgerissen. Über den Z-Index wird die Menüleiste vor alle anderen Elemente in den Vordergrund gestellt. Mittels Flexbox werden die Menüpunkte vertikal untereinander mit Abstand dazwischen ausgerichtet. Mit Hilfe einer Transition wird die Hover-Animation erzeugt.

Aktueller Stand des JavaScript-Client

Nach der Projektwoche Anfang Dezember 2017 musste man feststellen, dass die Zeit nicht reichen würde, um zwei Clients fristgerecht mit einem gewissen Funktionsumfang fertigzustellen. Bei einem Treffen mit Kunde und Coach am 06. Dezember 2017 wurde diese Thematik angesprochen, worauf der Kunde die Meinung vertrat, dass für ihn vor allem der Android-Client von grosser Bedeutung sei. Ein Showcase zu einer Android-Umsetzung existiert im Gegensatz zu JavaScript noch nicht. Daraufhin hat man sich im Projektteam darauf geeinigt, den JavaScript-Client einzustellen und sich voll und ganz auf den Android-Client zu konzentrieren. Der JavaScript-Client, so wie er

momentan existiert, ist nicht einsatzfähig. Diverse wichtige Funktionen fehlen und die Anknüpfung an den OpenDolphin-Server ist aktuell eher spärlich.

2.4.4 Android-Client

Viele moderne Lösungen benutzen heutzutage nicht mehr das Wort Software sondern Applikation. Und dies beschränkt sich nicht nur auf mobile Anwendungen. Die Applikationsentwicklung ist eigentlich lediglich eine Spezifizierung von Softwareentwicklung. Eine Applikation hat meistens einen sehr konkreten Nutzen, wie z.B. eine Zugverbindung herauszusuchen oder einen Partner für das nächste Mittagessen zu finden, während typische Nicht-Applikations-Software einen viel breiteren Nutzen findet, so wie zum Beispiel ein Grafikkartentreiber. Ähnlich zu dieser Thematik hat sich auch GroupHub dafür entschieden, die Idee zu spezifizieren.

Android Applikation spielen allerdings nochmals eine ganz besondere Rolle, denn es gibt einige Punkte, die im Design (Ästhetik sowie Codearchitektur) eine wichtige Rolle spielen:

- der Platz ist extrem begrenzt. Alle Informationen müssen in geeigneten Views und nachvollziehbare Navigationen gepackt werden.
- die Leistung der Geräte ist überraschend stark, aber nicht zu vergleichen mit Desktop-Geräten. Es müssen effiziente Design Patterns verwendet werden. Apps welche sich unverhältnismässig negativ auf die Akkulaufzeit auswirken, werden von beiden Global-Players (Google, Apple) aus dem AppStore genommen.

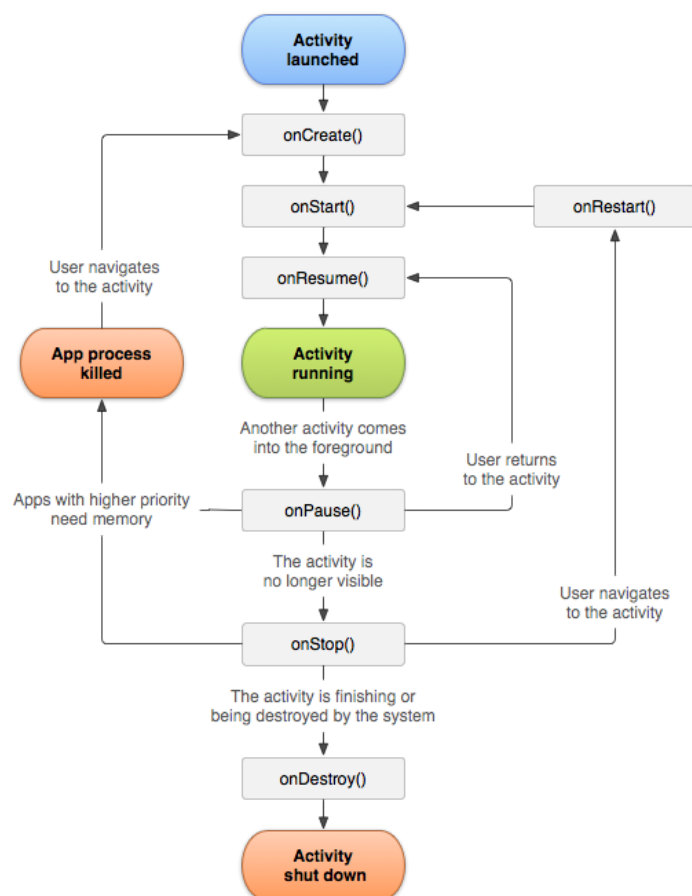


Abbildung 21: Die Lifecycle-Methoden von Android

- es gibt einen klar definierten [Lifecycle](#), den man berücksichtigen muss, damit Dinge wie Multitasking und Background Tasks möglich werden.
- Die Platzverhältnisse können sich durch Tablets extrem ändern. Es muss entschieden werden, wie damit umgegangen wird. Fragments bieten dafür die optimale Lösung.

2.4.4.1 Die Projekt-Struktur

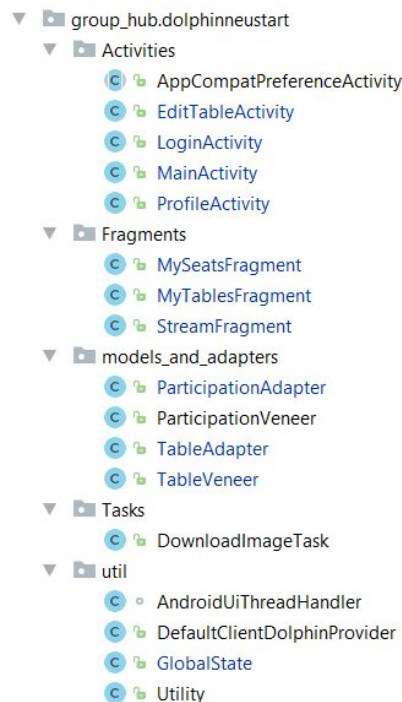


Abbildung 22: Ausschnitt aus der Projektstruktur des Android-Clients

Das Projekt ist auf der üblichen Android-Struktur aufgebaut: Beginnend mit dem Modul «manifest», wo das berühmt-berüchtigte Applikations-Manifest zuhause ist. Hier werden Berechtigungen der App festgelegt und die Beziehungen der verschiedenen Activities definiert. Die «MainActivity» ist in unserer App die «Mutter-Activity», denn sie ist zuständig für die meiste Logik. Im Manifest kann man ausserdem auch Themes bestimmen. Weiter geht es mit dem Modul «Java». Hier ist der gesamte Logik-Code abgelegt, den man als Entwickler schreibt. Eine Abbildung dieses Modules sieht man hier links. Mehr dazu in einem späteren Teil. Im Modul «res», abgekürzt für Ressourcen, befinden sich alle Icons, alle definierten Values (z.B. können Farben Variablen zugewiesen werden), sowie alle sogenannten View-Klassen im XML-Format. Wenn man eine Frage in der Form “Warum ist dieser Text hier blau?” beantworten möchte, dann liegt die Antwort meist in diesem Modul (die meisten Design-Entscheidungen können zwar auch anders verändert werden, doch dies sollte sich in Grenzen halten).

2.4.4.2 Ablauf

Da es im Manifest so definiert ist, startet die «MainActivity» als erstes, möchte man meinen. Dies ist nicht ganz der Fall. So hat man eine Klasse «GlobalState» (im Ordner Utility) implementiert, welche von «Application» erbt. Dadurch wird diese als allererstes instanziiert, noch vor jeder anderen Activity. Mehr dazu auf der offiziellen [Android-Website](#). «GlobalState» ist ein ziemlich treffender Name, denn diese Klasse speichert alle allgemein genutzten Objekte, welche nur einmal existieren sollen. Die Verbindung zum Server wurde dort beispielsweise geregelt, genauso wie auch nur dort

eine Instanz des aktuellen Benutzer existiert. Wird nun also die Applikation gestartet (`onCreate()` in «MainActivity»), so wird als erstes ein Ladebalken angezeigt. Währenddessen wird die oben beschriebene GlobalState-Instanz erstellt, danach die Verbindung zum DolphinServer hergestellt, einige Views referenziert und ein On-Click-Listener für den sogenannten «FloatingActionButton» (kurz: “fab”) definiert. Ebenso werden jetzt, so früh wie möglich, die Kommandos geschickt, um den PM-Store mit sinnvollen PMs zu befüllen. Die Reihenfolge dieser Kommandos ist im [Unterkapitel 2.4.1](#) beschrieben. Immer noch in der `onCreate()`-Methode der «MainActivity» werden alle Fragments instanziiert. Dies wäre mittlerweile gar nicht mehr nötig. Die Model-Store-Listener könnten nämlich im «GlobalState» implementiert werden, statt direkt in den Fragments. Sind die Kommandos fertig, so wird der Ladebalken weggeblendet, und angezeigt wird das wichtigste Fragment, das «StreamFragment».

2.4.4.3 Activity vs. Fragments

Die «MainActivity» bildet den Logik-Rahmen für die jeweiligen Fragments «StreamFragment», «MyTablesFragment» und «MySeatsFragment». Sie beinhaltet ebenso die Logik für den Navigation-Drawer sowie die Action-Buttons (hier nur refresh). Dies bedeutet, dass der Navigation-Drawer nur für «MainActivity» funktioniert. Hätten wir statt Fragments Activities implementiert, so würde bei einem Wechsel von «StreamFragment» zu «MyTablesFragment» die ganze Ansicht inklusive Drawer verschwinden und wieder auftauchen (je nach Animation). Dies bedeutet für den Benutzer eine schlechtere nachvollziehbare Navigation. Es lohnt sich allerdings, klar abgetrennte Features zu unterscheiden.

Aus diesem Grunde wurde entschieden, die «Mein Profil»- sowie «Login/Logout»-Features als Activities zu implementieren. Das letzte spannende Feature der «MainActivity» ist in der `onResume()`-Methode definiert. Jedesmal wenn die «MainActivity» benutzt wird, wird geschaut, ob der Benutzer angemeldet ist oder nicht. Wenn ja, dann werden seine Informationen direkt in den Drawer geschrieben. Der Runde Rahmen wurde mit Hilfe einer «CardView» erstellt.



Abbildung 23: Angemeldeter Benutzer in der Android-Applikation

2.4.4.4 Veneer

Auch interessant ist die Wiederverwendung der Veneer-Strategie aus dem initialen Beispielprojekt. Dies hat uns enorm geholfen, den Code lesbarer zu gestalten und als Folge davon, Fehler zu vermindern. Eine direkte Verbindung zum dahinterliegenden Modell haben wir erst in einer zweiten Runde aufgrund des Inputs unseres Betreuers und unserem Kunden implementiert. Was im Nachhinein auch viel logischer erscheint. Die einzelnen Adapter greifen nur noch über die Veneers auf die PMs zu. Dies ist ein wichtiger Lerngewinn, denn hätte man dies früher implementiert, wäre man einigen Syntax-Fehlern aus dem Wege gegangen. So hätten zum Beispiel die folgende Codezeile für grosse Verwirrung sorgen können:

```
state.getUser().getAt(PersonATT.NAME).getValue().toString()
```

2.4.4.5 Adapter

Im direkten Zusammenhang mit den Veneers stehen die zwei, im selben Ordner abgelegten, Adapter. Der «ParticipationAdapter» ist relativ langweilig, er iteriert durch seinen Datensatz und liefert die dazugehörigen Views zurück. Er basiert auf dem «ArrayAdapter». Im Gegensatz dazu hat uns der «TableAdapter» extrem viel Mühe bereitet, da die gesamte Thematik neu war. Um nämlich die verschiedenen Tisch-Instanzen anzuzeigen, verwendete man nicht den normalen «ArrayAdapter», sondern eine «RecyclerView». Diese hat den Vorteil, dass sie automatisch erkennt, wieviele Views sich im aktuellen «ViewPort» befinden und tatsächlich auch nur diese instanziiert. Verschwindet eine View durch Scrolling, so wird sie nicht gelöscht, sondern später wiederverwendet. Dies spart einiges an Ressourcen, weswegen diese Strategie verfolgt wurde.

In jedem Falle setzte dies voraus, dass das «ViewHolder»-Pattern umgesetzt wurde. Deshalb existiert in der Klasse «TableAdapter» auch die «TableViewHolder»-Klasse, welche zu allem Übel auch noch Click-Listener-Interfaces implementieren musste, damit die Information, auf welches Element genau geklickt wurde, weitergegeben werden kann. Ausserdem kann so auch unterschieden werden, ob ein Click auf einen «ViewHolder» direkt oder etwa auf ein Element innerhalb des ViewHolder geschehen ist. Dies führt zu einem weiteren Malheur, zum offensichtlichsten Fehler an der Applikation.

Es gab eine Liste innerhalb einer Liste. Es wurde nicht geschafft, die Liste der Tisch-Teilnehmer zur Laufzeit des Programmes bzw. während der Erstellung des «ViewHolders» anzupassen. Entweder war die Liste zu gross oder zu klein, aber nie angepasst auf den tatsächlichen Inhalt. Man kann allerdings durch die Liste iterieren, indem man eine Person anklickt und danach mit den Pfeiltasten navigiert. Die vermeintlich offensichtliche Lösung «NestedScrollView» zu benutzen, trug auch keine Früchte. Dies ist eine Sache für einen neuen Branch.



Abbildung 24: Ein Tisch in der Android-Applikation

2.4.4.6 Features

- **Ladebalken**

Als erstes springt einem der Ladebalken ins Auge. Dieser wurde auf Anfrage des Kunden implementiert, denn es bestand der Wunsch, dass man clientseitige Technologie einbindet. Dadurch kann man aufzeigen, dass die Technologie OpenDolphin keinerlei Restriktionen gegenüber den Möglichkeiten der nativen API bringt.

- **Stream**

Wenn der Ladebalken verschwindet, sieht man den Stream. Hier werden die 20 (dies ist serverseitig so definiert, es wäre eine Idee für eine Erweiterung, dies dem Benutzer zu überlassen) neuesten Tische angezeigt. Im rechten, oberen Rand eines Tisches sieht man das Ausführungsdatum. Hier ist es interessant zu wissen, dass jeweils nur Tische in der Zukunft geladen werden, und diese serverseitig sortiert werden. Meldet man sich mit einem gültigen Konto an, so werden einem alle benötigten Tische nachgeliefert (z.B. falls ein Benutzer bereits Organisator von zwei Tischen ist, diese allerdings nicht zu den 20 Neusten gehören, dann werden diese bei der Anmeldung nachgeliefert).

- **Login-Funktion**

Es existieren vier Power-User: Dierk Koenig, Dieter Holz, Mario Winiker, Tobias Sigel. Diese vier haben das PW «admin» und sind mit Daten befüllt. Alle anderen Benutzer haben das PW «GroupHubRocks!».

Ohne Login ist die Applikation unbrauchbar. Jede Interaktion, die ein Login

benötigt, wird von einem Pop-Up-Alert-Dialog begleitet, wobei man sich dort gleich für das Login entscheiden kann. Interessanterweise sind deshalb für eine Anmeldung weniger Klicks nötig, wenn man auf eine falsche Schaltfläche klickt. Das mit dem Login funktioniert so, dass in jedem Falle beim Start der Applikation eine Verbindung zu einem leeren «PersonPM» erstellt wird. In der «LoginActivity» gibt der Benutzer seine Daten ein, der Server sucht und kontrolliert diese und wenn alles in Ordnung ist, werden die Daten des gefundenen Benutzers auf das leere «PersonPM» übertragen. Dies bedeutet, dass bei einem Logout (Logout wird nur ausgelöst, wenn man angemeldet ist und nochmals auf Login im Drawer klickt) die Daten wieder von dem einst leeren PM auf das Original überschrieben werden müssen. Diese Strategie wurde gewählt, da der Befehl «syncPM» anscheinend nicht dafür genutzt werden sollte. Eine Lösung mittels Qualifier wäre schön gewesen, aber es wurde zeitlich leider zu knapp.

- **Teilnehmen**

Hier kann man sich bei einem Tisch einschreiben und zugleich einen Kommentar abgeben. Die Teilnehmeranzahl aktualisiert sich sogleich und man kann sich selber in der Liste der teilnehmenden Personen sehen. Über den Drawer kann man nun zu «Meine Anmeldungen» navigieren, wo der Tisch aufgelistet wird. Power-User haben bei der Initiierung standardmässig weder Tische noch Anmeldungen. Zufalls-Benutzer können beides zufällig haben. Die Logik funktioniert hier so, dass beim Klick auf den Button das Kommando zur Erstellung eines leeren Participation-PMs ausgelöst wird. Im On-Finished-Handler des Kommandos kann man das neue PM abfangen und mit den richtigen Daten befüllen, was anhand des AlertDialogs gemacht wird.

- **Neuen Tisch erstellen / Tische bearbeiten**

Über den «FloatingActionButton» unten rechts kann man einen Tisch erstellen. Die Logik funktioniert hier genau gleich wie bei der «Teilnehmen»-Funktion. Wir fangen das neu erstellte PM ab und senden es direkt in die «EditTable»-Activity, welches daraus ein «TableVeneer» erstellt. Die «EditTable»-Ansicht ist sicherlich noch nicht die Schönste, dafür werden die Daten bereits sinnvoll befüllt. Der aktuelle Benutzer ist bereits Organisator, das Ausführungsdatum ist stets eine Woche nach dem heutigen Datum. Bestätigt man diese Ansicht, so ist der Tisch offiziell bearbeitet worden. Man findet ihn nun, wenn man zu «Meine Tische» navigiert oder im Stream ganz nach unten scrollt. Aus diesem Grund wäre eine clientseitige Sortierung der Tische ein zukünftiges Ziel. Der orange Edit-Button fällt sofort auf, wenn man den soeben erstellten Tisch in der Liste findet. Diesen sieht man nämlich nur, wenn man als Organisator des Tisches angemeldet ist.

- **Mein Profil**

Hier kann man als Benutzer seine Daten angeben. Alle Angaben sind freie

String-Felder, d.h. man kann hier alles eingeben, was man möchte. Es gibt keine Input-Validation. Das Interessante hier ist das runde Profilbild, welches wieder mit Hilfe einer «CardView» sowie dem dazugehörenden «DownloadImageTask» entstand. Im unteren Bereich ist ein Feld namens «URL-Profilbild» und hier kann ein Benutzer jeden URL angeben, den er möchte. Das Bild wird im Hintergrund heruntergeladen und angezeigt. Alle Zufalls-Benutzer haben «Delfine» als Profilbild.

3 Schlusswort

Die Entscheidung, das Projektteam quasi zu trennen und gleichzeitig an zwei unterschiedlichen Clients zu arbeiten, war, im Nachhinein betrachtet, keine Gute. Der Teamgeist musste, vor allem über die Zeit der Projektwoche, enorm darunter leiden. Jeder war mit seinem eigenen Süppchen beschäftigt und hatte keine Zeit für die Probleme des Anderen. Beide hatten das Gefühl, sie kämen mit ihren Arbeiten nicht voran und würden sich gegenseitig sogar im Weg stehen, da man auf der gleichen Basis, dem OpenDolphin-Sever, aufbaute. Gegen Ende dieser intensiven Woche, wurde dieses Gefühl sogar noch verstärkt, weil beide ein angefangenes Produkt hatten, welches nicht im entferntesten an den im MVP definierten Funktionsumfang herankam. Nach einem holprigen Projektstart nun auch noch das. Aus heutiger Sicht hätte man diese Trennung unbedingt vermeiden und sich stattdessen die gemeinsamen Stärken zunutze machen sollen. Könnte man das Projekt von neuem beginnen, würde man sich wohl auf eine einzige Client-Technologie fixieren.

Anfangs war die grösste Hürde das Konzept von OpenDolphin zu verstehen. Vieles wurde erklärt, vieles gelesen, doch ein ausreichend komplexes Konzept muss benutzt werden, damit es verstanden werden kann. So war die Fokussierung auf JavaFX am Anfang des Projekts definitiv ein guter Schritt. Ein gutes Instrument um die ersten Hürden zu meistern.

Im Nachhinein, beim Abschliessen dieser Arbeit, wurde uns immer mehr bewusst, was die Stärken von OpenDolphin sind und wie mächtig dieses Framework ist. Es hätte noch einiges zum Ausprobieren gegeben. Relativ dazu gesehen, hatten wir in der Mitte des Projektes gar keine Ahnung, was wir taten. Die Entscheidung zwei neue Clients zu implementieren war dann paradoxerweise eine sehr willkommene Herausforderung. Wir durften mit Technologien arbeiten, die uns beide sehr interessierten, in denen wir aber beide auch nur wenig Erfahrung hatten. Hier wird einem bewusst, dass man als Informatiker unbekanntes Territorium schnell mal unterschätzt. Wir fühlten uns überhaupt nicht sicher im Umgang mit der Haupttechnologie OpenDolphin und trotzdem wollten wir zwei zusätzliche neue Technologien dazunehmen. Ein fataler Fehler.

Alles in allem sind wir aber trotzdem sehr stolz auf die entstandene Lösung. Wir haben doch noch eine funktionierende Applikation hinbekommen, an die wir zeitweise nicht mehr geglaubt haben. Diese Lösung ist nicht perfekt. Einige Features fehlen noch, wie zum Beispiel eine vollintegrierte Freundes-Funktion, oder die Möglichkeit, die Personen-Liste innerhalb der Tisch-Liste mit dem Finger zu durchscrollen. Wir haben jedoch viel gelernt, vor allem auch im Hinblick auf die bevorstehende Bachelorarbeit. Mit der Erstellung der Dokumentation sollte zum Beispiel nicht erst gegen den Schluss des Projektes begonnen werden. Der Kampf um die richtigen Worte würde viel einfacher ausfallen, hätte man das Erledigte vorweg dokumentiert. Es darf zudem nicht unterschätzt werden, wie wichtig es ist, die Arbeitsaufwände für die einzelnen Anforderungen vorgängig abzuschätzen. Damit kann verhindert werden, dass man Stunden mit einzelnen Funktionen verbringt und dabei vergisst oder verdrängt, was daneben noch alles erledigt werden muss. Aber auch was die Technologien betrifft, konnten wir vieles aus diesem Projekt mitnehmen. Neben OpenDolphin und JavaFX, durften wir Android und JavaScript näher kennenlernen. Endlich hatten wir Zeit, etwas neues auszuprobieren. Auch das ViewHolder-Pattern und die Veneers waren neu für uns. Wir haben das erste Mal Lambdas und Streams zwecks einfacherem, kürzerem und lesbarerem Code verwendet und nicht wie sonst im Kontext einer Übung. Wir haben Neues zu den Themen «MemoryLeak», «GarbageCollection» und verbotenen Static-Fields in der Android-Entwicklung gelernt. Das Konzept von On-Finished-Handlern haben wir uns zum ersten Mal zunutze gemacht. Und wir haben zum ersten Mal eine Profilfunktion umgesetzt und sind mit dessen Fragestellungen in Kontakt gekommen.

Die Arbeit ist fertig, die Applikation und der Server auf GitHub, bald ist der Abgabetermin passé. Was bleibt, ist die Reflexion. Die Frage ob man zufrieden ist mit dem Geleisteten. Und wir müssen sagen: Die Applikation, die wir zu Beginn in unseren Köpfen hatten, die Applikation, die wir eigentlich machen wollten, die hätten wir nur machen können, wenn das nötige Wissen dazu bereits vorhanden gewesen wäre. Aber genau deshalb sind wir ja hier. Um zu lernen.

4 Abkürzungsverzeichnis

BIOS	Basic input/output system
CSS	Cascading style sheets
DOM	Document object model
DTO	Data transfer object
HTML	Hypertext markup language
ID	Identifikator
IoT	Internet of things
JS	JavaScript
MVP	Minimum viable product
OD	OpenDolphin
PM	Presentation model
PW	Passwort
SVG	Scalable vector graphics

5 Abbildungsverzeichnis

Abbildung 1: Die ersten Prototypen	Seite 07
Abbildung 2: Der Stream als Wireframe	Seite 08
Abbildung 3: Das Profil als Wireframe	Seite 09
Abbildung 4: Tische erstellen als Wireframe	Seite 09
Abbildung 5: Die Suche als Wireframe	Seite 10
Abbildung 6: Card (nicht eingeschrieben)	Seite 10
Abbildung 7: Card (eingeschrieben)	Seite 10
Abbildung 8: JavaFX-Client im Entwicklungsstadium	Seite 11
Abbildung 9: JavaScript-Client im Entwicklungsstadium	Seite 12
Abbildung 10: Vektorgrafikentwürfe von Stuhl & Tisch	Seite 13
Abbildung 11: Ausschnitt aus einem Restaurant	Seite 14
Abbildung 12: Endentwurf im Material-Design	Seite 15
Abbildung 13: Einstellungen für Projektimport in IntelliJ IDEA	Seite 18
Abbildung 14: Projektstruktur von OpenDolphin	Seite 19
Abbildung 15: Ausschnitt aus dem lokalen Maven-Repository	Seite 20
Abbildung 16: Die Controller des Server-Moduls	Seite 21
Abbildung 17: Hierarchie der PM-Instanziierungen	Seite 21
Abbildung 18: Ausschnitt aus dem GroupHubRemoteService	Seite 22
Abbildung 19: Ausschnitt aus dem Logging bei der Erstellung von Personen	Seite 22
Abbildung 20: Ausschnitt aus der Projektstruktur von OpenDolphin	Seite 24
Abbildung 21: Die Lifecycle-Methoden von Android	Seite 30
Abbildung 22: Ausschnitt aus der Projektstruktur des Android-Clients	Seite 31
Abbildung 23: Angemeldeter Benutzer in der Android-Applikation	Seite 32
Abbildung 24: Ein Tisch in der Android-Applikation	Seite 34

6 Quellenverzeichnis

Mozilla und einzelne Mitwirkende (2017): Array.prototype.map().

https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Array/map (abgerufen am 20.01.2018).

Mozilla und einzelne Mitwirkende (2017): Template-Strings.

https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/template_strings (abgerufen am 20.01.2018).

The Dolphin team (2014): OpenDolphin on the web - Reference Documentation.

http://open-dolphin.org/download/guide/guide/web.html#web_intro (abgerufen am 21.01.2018)

Theis, Thomas (2016): Einstieg in JavaScript. 2., aktualisierte Auflage. Bonn: Rheinwerk Verlag GmbH.

Wolf, Jürgen (2015): HTML5 und CSS3 - Das umfassende Handbuch. Bonn: Rheinwerk Verlag GmbH.