Projektbericht Skatenight-App

Projektseminar „Android-Programmierung“ im WS 14/15

Bernd Eissing, Pascal Otto, Daniel Papoutzis,
Tristan Rust, Richard Schulze, Martin Wrodarczyk

Inhaltsverzeichnis

1 Einleitung

von allen

2 Vorgehensweise

von Tristan und Richard

Am Anfang des Projektes haben wir uns darauf geeinigt alle die gleiche Integrierte Entwicklungsumgebung (IDE) „Android Studio“ von Google zu nutzen, welches die offizielle Entwicklungsumgebung für Android ist, damit es nicht zu Problemen innerhalb des Projektes kommt. Weiterhin haben wir uns für die freie Software „Git“ zur Versionsverwaltung des Projekts entschieden. Hierfür gab es einen Master Branch auf dem sich immer eine lauffähige Version der Apps befunden hat. Für die Entwicklung einzelner Funktionen, wie zum Beispiel die „lokale Auswertung“wurden extra Branches angelegt. Für das Vorgehen im Projekt wurde das Rahmenwerk Scrum verwendet. Die Rolle des Product Owners wurde von den Betreuern übernommen. Außerdem hat die Rolle des Scrum Masters nach jedem einzelnen Sprint, der in der Regel 4 Wochen betrug, ein anderes Mitglied des Entwicklungsteams übernommen. Zudem wurden die einzelnen Aufgaben für das Sprint Backlog vor jedem Sprint bzw. nach dem Sprint Review in Tasks aufgeteilt. Damit ein Task wirklich als bearbeitet („done“) gilt wurde vor dem Projekt definiert, ab wann ein Task als dies gelten darf:

- Programmierung abgeschlossen
- Funktion getestet
- Ein Code Review durchgeführt wurde
- Die Feature-Branch auf den master-Branch gemerged wurden
- Die Dokumentation (Java Doc) geschrieben wurde

Außerdem wurde regelmäßig das Daily Scrum durchgeführt, wodurch wir uns gegenseitig Informiert haben, wer gerade an was arbeitet, ob Probleme aufgetreten sind oder Hilfe benötigt wird.
Zum kontinuierlichen Bauen der Apps und des Backends wurde ein Jenkinsserver eingesetzt. Da es einige Schwierigkeiten bei der Durchführung von Tests auf dem Jenkinsserver gab, haben wir bei unserer Projektarbeit nur sehr selten den Jenkinsserver in Anspruch genommen. Dennoch besteht eine Möglichkeit das Projekt auf dem Jenkinsserver zu bauen. Der Ablauf ist dabei wie folgt:

1. Säubern des Projekts mit dem Befehl „clean“
2. Anpassen der Versionsnummer der Build-Tools von 1.9.17 auf 1.9.8 in der Gradle-Builddatei für das Backend, da das Bauen des Backends sonst fehlschlägt.
3. Bauen und Hochladen des Backend auf den Release-Server
4. Zurücksetzen der Änderungen in der Build-Datei.
5. Bauen und Signieren der User- und Veranstalter-App in der Release-Konfiguration.

3 Features der Apps

von Daniel

Die Anforderungen und Features der Apps wurden von Sprint 1 bis Sprint 4 stetig erweitert. Angefangen mit eher simplen Anforderungen, gewannen diese im Laufe des Projektseminars immer mehr an Komplexität. Wie durch Scrumm vorgegeben werden diese Anforderungen vom Product Owner gestellt und in Zusammenarbeit mit dem Team ins Sprint Backlog übertragen. Dabei wird über die einzelnen Features des Sprint diskutiert und evtl. an ihrer Priorisierung gearbeitet. Ein Sprint dauerte 4 Wochen, in welchen das Team Zeit hat alle Punkte des Sprint Backlogs abzuarbeiten. Im folgenden werden die Anforderungen an die Apps pro Sprint aufgelistet und erläutert.

3.1 Sprint 1

Im ersten Sprint ging es darum ein Grundgerüst zu erstellen und rudimentäre Features zu implementieren. Diese wurden in den folgenden Sprints erweitert und ausgebaut.
Aktuelle Informationen anzeigen:
Es soll eine Möglichkeit geben, sich das aktuelle Event in der Benutzer App anzeigen zu lassen. Dieses besteht aus vorgegebenen Informationen, wie z.B. Name, Ort, Datum, Kosten, Beschreibung, usw. Die Informationen sollten dabei jedes mal vom Server heruntergeladen werden.
Aktuelle Strecke anzeigen:
Um eine Strecke anzuzeigen muss diese auf dem Server gespeichert und anschließend in der App durch eine Google Map visualisiert werden.
Veröffentlichen neuer Informationen:
In der Veranstalter App soll es möglich sein die Daten für die „Aktuellen Informationen“ einzutragen und zu verändern.
Erstellen neuer Strecken:
Für das Erstellen neuer Strecken muss die Veranstalter App ein Interface bereitstellen, indem auf einer Google Map interaktiv Strecken erstellt werden können, sodass diese zum Anzeigen in der Benutzer App verwendet werden.
Authentifikation des Veranstalters:
Sicherheitshalber muss eine Authentifizierung des Veranstalters stattfinden, damit nur befugte Benutzer Zugriff auf die Veranstalter App erhalten. Um dies zu gewährleisten soll eine Liste mit berechtigten E-Mail-Adressen auf dem Server hinterlegt werden und anschließend beim Ausführen der App die Authentifizierung des Benutzers überprüft werden.
Anzeigen der Position:
Die „Aktuelle Strecke anzeigen“ soll erweitert werden, sodass die, vom GPS ermittelte Position des Benutzers, angezeigt wird.
Übertragung der GPS-Daten an den Server:
Anschließen sollen die GPS Daten der Benutzer an den Server gesendet und gespeichert werden.

3.2 Sprint 2

Der Fokus des zweiten Sprints lag auf der logischen Weiterentwicklung der im ersten Sprint implementierten Features. Die Apps sollten mehr Möglichkeiten sowohl für Veranstalter als auch Benutzer bieten.
Anzeigen mehrerer Veranstaltungen:
Die Möglichkeit Aktuelle Informationen anzeigen zu lassen, sollte nun erweitert werden, sodass zwischen mehreren Veranstaltungen ausgewählt werden kann und zu jeder die hinterlegten Informationen abgerufen werden können.

Veröffentlichen mehrerer Veranstaltungen:
Sowie in der Benutzer App mehrere Veranstaltungen angezeigt werden sollen, muss auch für die Veranstalter eine Option bestehen mehrere Veranstaltungen anzulegen und zu verwalten.
Globale Auswertung der Positionen der Teilnehmer:
Die im ersten Sprint übertragenen Positionen der Teilnehmer sollen nun ausgewertet werden können um diese als Feld auf der Karte zu visualisieren. Dabei sollen regelmäßige Aktualisierungen dafür sorgen, dass der ständige Fortschritt der Skater während des Events verfolgt werden kann.
Lokale Auswertung der Position:
Anders als die Globale Auswertung bietet die Lokale Auswertung dem Nutzer viele nützliche Informationen ohne die Notwendigkeit seine Position an den Server zu übermitteln. Er kann seinen Fortschritt auf der Strecke nachvollziehen und weiter Informationen über seine Geschwindigkeit, Leistung usw. erhalten.
Veröffentlichen neuer Informationen 2.0:
Das Informationen Veröffentlichen 2.0 hat einige Features gefordert, die sich vom bisherigen Veröffentlichen unterscheiden. Zum einen soll es nun möglich sein bereits erstellte Veranstaltungen im Nachhinein zu bearbeiten. Zum anderen sollen frei definierbar neue Informationsfelder zu den Veranstaltungen hinzugefügt werden können, sowie z.B. Bilder, Links zu Webseiten usw.

3.3 Sprint 3

Da aus Zeitmangel im zweiten Sprint leider nicht alle Anforderungen erfüllt werden konnten, wurden einige in den dritten Sprint übertragen. So gab es insbesondere Probleme die „Veröffentlichen neuer Informationen 2.0“ fehlerfrei mit den anderen Programmteilen zu verbinden. Zusätzlich war es nicht gelungen alle Aspekte der Lokalen Auswertung vollständig in die Benutzer App zu integrieren.
Lokale Auswertung der Position 2.0:
Die Anforderungen aus dem zweiten Sprint wurden übernommen und erweitert, so soll es nun dem Benutzer ermöglicht werden seine Fahrt aufzuzeichnen und sein Geschwindigkeitsprofil anzeigen zu lassen.
Push-Notifications / Broadcast-Receiver:
Die Benutzer sollen automatisch per Notification darüber informiert werden ob ein neues Event erstellt wurde.
Handhabung der Position:
Es gibt Einstellungsmöglichkeite, die es den Benutzern erlauben das Senden ihrer Position an den Server zu unterbinde.
Rechteverwaltung für Veranstalter:
Veranstalter können Nutzer zu der Liste der authentifizierten Veranstalter hinzufügen, was diesen Zugriff auf die Veranstalter App ermöglicht.

3.4 Sprint 4

Der vierte Sprint war der Abschluss des Projektseminars und fiel aufgrund der Klausurenphase kürzer aus als die übrigen Sprints. So wurde die Fertigstellung, Fehlerbehebung und reibungslose Integration aller Programmteile in diesem Sprint priorisiert.
Lokale Auswertung der Position 3.0:
Die Daten der Lokalen Auswertung sollen auch nach dem Ende eines Events erhalten bleiben, sodass Benutzer auch in der Zukunft die Aufzeichnung vergangener Events anschauen können.
Aktuelle Informationen + Notifications:
Die Benutzer sollen per Notification darüber informiert werden, wenn ein Event, an dem sie teilnehmen geändert wurde oder startet.
Nutzergruppen:

Die Anwender der User App können Nutzergruppen erstellen, beitreten um sich anschließend die Benutzer ihrer Gruppen auf der Karten anzeigen zu lassen. So ist es möglich ein Überblick darüber zu erhalten auf welchem Teil der Strecke sich die verschiedenen Gruppenmitglieder aufhalten.
Erstellen neuer Strecken 2.0:
Es soll für Veranstalter möglich sein Wegpunkte zu Strecken hinzuzufügen. Diese können anschließend in der Benutzer App angezeigt werden, sodass sich die Skater eine Übersicht über Start, Pausen, Checkpoints usw. machen können.

4 Vorstellung von Android

von Pascal

4.1 Beschreibung von Adaptern im Zusammenhang mit ListViews

von Martin und Bernd

Eine ListView ist eine UI-Komponente die eine scrollende Liste mit einzelnen Einträgen darstellt. Ein Eintrag in der ListView wird meist als Item betrachtet. Ein Item ist ein View-Objekt, das häufig durch eine eigene xml-Datei beschrieben wird. Eine ListView-Instanz benötigt einen Adapter um die Einträge der ListView mit Inhalt zu füllen. Ein Adapter ist eine Java-Klasse, die eine Liste von Datenmodell-Objekten hält und diese über die *getView(...)* Methode zur Anzeige aufbereitet. Diese Methode wird für jedes Item immer dann aufgerufen, wenn es sichtbar wird.

5 Server-Backend

5.1 Anforderungen

von Richard

Zur Kommunikation der Apps, aber auch zur zentralen Speicherung der Veranstaltungen, Routen, Nutzern und Nutzergruppen ist ein Backend, im folgenden auch Skatenight API genannt, notwendig, das eine öffentlich erreichbare Schnittstelle zur Verfügung stellt. Neben der Speicherung von Daten ist aber auch notwendig, dass der Server eigene Berechnungen durchführen kann. So wird zum Beispiel das Feld, das den Benutzern in der User-App angezeigt wird, zentral auf dem Server berechnet und von den User-Apps abgerufen (siehe Abschnitt 5.7).
Eine weitere wichtige Anforderung war für uns, dass der Server kostenlos nutzbar ist und nach Möglichkeit bereits ein Grundgerüst bietet, über das aus Android-Apps möglichst einfach auf die Server-API zugegriffen werden kann. Wir wollten Hauptaugenmerk auf die Entwicklung der Apps legen und die Zeit zur Entwicklung des Backends möglichst gering halten. Das Google App Engine Projekt bietet diese Möglichkeit, da es von den realen Servern vollkommen abstrahiert. Zusätzlich dazu werden einige nützliche APIs angeboten, die unter anderem Funktionalität zur Speicherung von Daten bereitstellen und die Definition von Kommunikations-Endpunkten, sowie die Kommunikation vom Server zum Handy ermöglichen.

5.2 Server, Keys und Build-Varianten

von Richard

Während der Entwicklung der Apps ist uns schnell aufgefallen, dass aufgrund der verschiedenen Features, die jeweils auf eigenen Branches entwickelt wurden, auch der Server von mehr als

Web-Client ID: Dient der Authentifizierung über OAuth2, das die gesicherten API-Funktionen, wie beispielsweise das Anlegen eines Events schützt.		
Key	Release	37947570052-dk3rjhgran1s38gscv6va2rmmv2bei8r.apps.googleusercontent.com
	Debug	644721617929-unb9em0kl73b9evdv2h52ufn26fao20p.apps.googleusercontent.com
	Jenkins	1032268444653-7agre4q3eosqhlh92sq62hf5fan9jbv5.apps.googleusercontent.com
Client-ID der User-App: Dient der Identifikation der User-App am Backend.		
Key	Release	37947570052-g006o3ovfotnjqreltom6c7cbktm7dap.apps.googleusercontent.com
	Debug	644721617929-mk0do3jm5lec1dasijdj3220ot87gbn7.apps.googleusercontent.com
	Jenkins	1032268444653-kj1mpisvlrpl67e7db2fu2005aube7mu.apps.googleusercontent.com
Client-ID der Veranstalter-App: Dient der Identifikation der Veranstalter-App am Backend.		
Key	Release	37947570052-4ru7asfhnrjmmqvj3qdpo2rrp31fudf3.apps.googleusercontent.com
	Debug	644721617929-kbcha00vb3j30sh9at05sagm73iltqqo.apps.googleusercontent.com
	Jenkins	1032268444653-83ih5mp5mguh66c012u0sobqe4oqoupr.apps.googleusercontent.com
Google-Maps API-Key für die User-App: Wird zur Nutzung der Google-Maps API in der User-App benötigt.		
Key	Release	AIzaSyBKGVQHij0JcjgHyCnT0ZrLihbCn8Av2jg
	Debug	AIzaSyDQgEKF42jjm57x7kgnY7EI62CcXW_zDS0
	Jenkins	AIzaSyAQ47zRjgPRk1sxJYifd2URQdEnk5HvnRw
Google-Maps API-Key für die Veranstalter-App: Wird zur Nutzung der Google-Maps API in der Veranstalter-App benötigt.		
Key	Release	AIzaSyBedRi4A–p3LNQLy5lgOsBEdIdKuLbi2U
	Debug	AIzaSyAO4zGlaKexByZmEiFKLbo18Y_I-t9KbMc
	Jenkins	AIzaSyBes8RSzVdqcQY-vnj9GLWxCLBKAFCzh98
Google Cloud Messaging API-Key: Dient der Registrierung beim GCM-Dienst von Google in der User-App.		
Key	Release	AIzaSyDO8mosWwYXjjGZ9besu9CZw1LDEEXrFXE
	Debug	AIzaSyCpwhxda1Lb5E61_fybZq2iSJgViZu3QNM
	Jenkins	AIzaSyBjeKzx_vzkUgQZdT83BB0BMD3gFebpet0

Tabelle 1: Eine Liste aller verwendeter IDs und Keys, die zur Authentifizierung der Apps und Server notwendig sind.

einer Person angepasst wurde. Damit einher ging, dass zwischenzeitlich verschiedene Versionen des Servers notwendig waren, damit jede Teilgruppe des Teams das eigene Feature implementieren konnte, ohne die Entwicklung anderer Teammitglieder zu stören. Aus diesem Grund haben wir mehrere Projekte bei der Google App Engine angelegt, zwischen denen relativ einfach gewechselt werden kann. In unserem Projekt in den Modulen „app“ und „veranstalterapp“ gibt es entsprechende Build-Varianten, mit denen der gewünschte Server ausgewählt werden kann. Da es im Modul „SkatenightBackend“ nicht die Möglichkeit gibt Build-Varianten zu definieren, gibt es dort die Befehle „applyProductionserverConfig“, „applyTestserverConfig“ und „applyJenkinsserverConfig“. Diese kopieren jeweils aus den entsprechenden Unterordnern des src-Ordners die Konfigurationsdateien, die für einen Server-Wechsel notwendig sind, in den main-Ordner. Bei einem Serverwechsel wird die *Constants.java* und die Projekt-ID in der Datei *appengine-web.xml* ausgetauscht. Wenn anschließend der Befehl „appengineUpdate“ zur Aktualisierung des Servers aufgerufen wird, wird der Server aktualisiert, dessen Projekt-ID in der *appengine-web.xml* steht. Tabelle 1 listet nocheinmal die Details zu den einzelnen Google App Eingine Projekten und auch die notwendigen Keys auf, mit denen sich die Apps gegenüber dem Backend und anderen Diensten identifizieren.

5.3 Klassenstruktur der Skatenight API

von Richard

Das Backend ist eine in Java geschriebene Anwendung, die sich im Projektordner in dem Modul „SkatenightBackend“ befindet. Das für die Apps zugängliche Interface wird in der Klasse *SkatenightServerEndpoint* beschrieben. Alle darin implementierten öffentlichen Methoden sind über die generierten Client-Libraries aufrufbar.
Die Klasse *EventStartServlet* wird über einen Cron-Job, der in der Datei *cron.xml* im WEB-INF-Ordner des Moduls definiert ist, alle zwei Minuten aufgerufen. Ihre Aufgabe ist es nach Events zu suchen, die kürzlich begonnen haben und für die noch keine GCM-Nachricht an die teilnehmenden Benutzer versandt wurde. Problematisch war, dass Cron-Jobs bei einem Google App Engine Projekt nicht dynamisch zur Laufzeit erstellt werden können. Die zunächst angedachte Lösung für jedes Event einen eigenen Cron-Job anzulegen, der genau zum Startzeitpunkt des Events aufgerufen wird, ist dadurch nicht möglich. Wir haben uns deshalb dafür entschieden, die Events mit einem Boolean zu markieren, wenn sie gestartet sind, sodass in der *doGet()* -Methode nach Events gesucht werden kann, für die diese Markierung noch nicht gesetzt wurde. Für jedes startende Event kann dann eine Liste der GCM-IDs der Benutzer erstellt werden, die über das Event benachrichtigt werden müssen. Gleichzeitig wird für die Benutzer auch die aktuelle Event-ID(*currentEventId*) gesetzt, die für die Berechnung des Feldes relevant ist.

5.4 Datenmodell

von Richard

Abbildung 1 zeigt das auf dem Server repräsentierte Datenmodell. Die Attribute, die mit einer dickeren Linie gestrichelt umrandet sind, sind Listen von Objekten, die serialisiert werden und nicht als eigener Typ in der Datenbank existieren. Die Veranstalter werden als Hosts in der Datenbank abgespeichert, haben jedoch keine weiteren Beziehungen zu anderen Entitätsklassen. Wie auf der Abbildung gezeigt, werden zu einem Event auch die teilnehmenden *Member* gespeichert. In dem Event existieren dazu jedoch nicht direkt die Member-Objekte, sondern nur Strings, die auf den Primary Key der Klasse *Member* (*email*) zeigen. Dies ist notwendig, da aufgrund der Struktur des Google Datastore die Member nicht existentiell von den Events abhängen dürfen. Im Datastore kann für jeden Datensatz ein übergeordneter Datensatz definiert werden. Würde dies das Event sein, so würden beim Löschen des Events auch alle teilnehmenden Member-Datensätze gelöscht werden. Analog sind auch die Benutzergruppen implementiert. Damit beim Löschen einer Benutzergruppe die Benutzer nicht ebenfalls gelöscht werden, enthalten die Benutzergruppen ebenfalls nur eine Liste von Strings, die auf die E-Mails der enthaltenen Benutzer zeigen.
Die Umsetzung des Klassenmodells des Backends weicht etwas von der in der Datenbank gewählten Struktur ab. Hauptunterschied ist, dass mehr Klassen zur Repräsentation der Daten definiert wurden. So gibt es beispielsweise zusätzlich zu den Routen auch die Klassen *RoutePoint* zur Repräsentation eines feinen Wegpunktes auf der Strecke, sowie die Klasse *ServerWaypoint* , die die beim Erstellen der Strecke gesetzten Marker speichert. Einen genauen Überblick über das Klassenmodell bietet Abbildung 2.

5.5 Verfügbare API-Aufrufe

von Richard

Zur Kommunikation der User- und Veranstalter-App mit dem Backend existieren einige API-Aufrufe auf dem Backend. Ein wichtiges Kriterium bei der Implementierung war, dass nicht jeder Benutzer die gleichen Rechte hat. So kann ein Veranstalter z.B. Events erstellen und löschen,

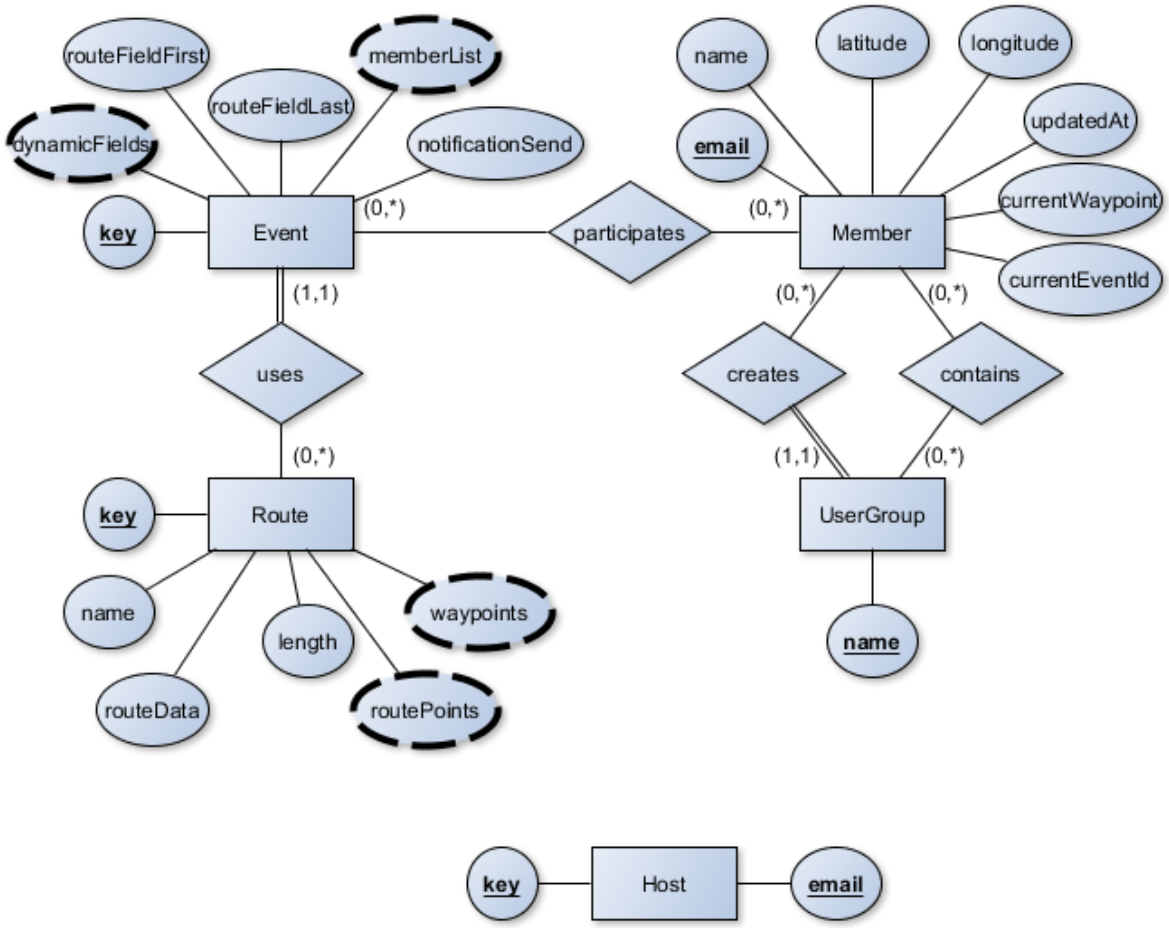


Abbildung 1: ER-Modell der Daten, die auf dem Server gespeichert werden

ein normaler Skatenight-Teilnehmer sollte dies jedoch nicht können. Aufgrund dieser Anforderungen haben wir uns für die OAuth2-API von Google entschieden. Mit ihr ist es einfach, eine Authentifizierung der Benutzer zu implementieren, da sie in Google Cloud Endpoints bereits integriert ist. Auf dem Server werden alle Methoden, die eine Authentifizierung benötigen, um einen Parameter vom Typ *com.google.appengine.api.users.User* erweitert. Beim Verbindungsaufbau in der App wird der am Handy angemeldete Benutzer ausgelesen und bei jedem Aufruf an das Backend nun automatisch übergeben. Hinzu kommt, dass die Authentifikation über den Google-Account geschieht. Da die App für Android-Geräte entwickelt wurde, ist sichergestellt, dass jeder Benutzer der App auch ein entsprechendes Konto besitzt. Die folgende Liste beschreibt die verfügbaren API-Aufrufe und stellt auch kurz dar, in welchem Zusammenhang diese in den Apps verwendet werden.

addHost

Die Methode *addHost* dient dem Hinzufügen von Veranstaltern. Durch den User-Parameter in der Signatur der Methode ist der Aufruf so geschützt, dass nur bereits eingetragene Veranstalter neue Veranstalter hinzufügen können. In der Veranstalter-App ist diese Funktion in der Rechteverwaltung zugänglich.

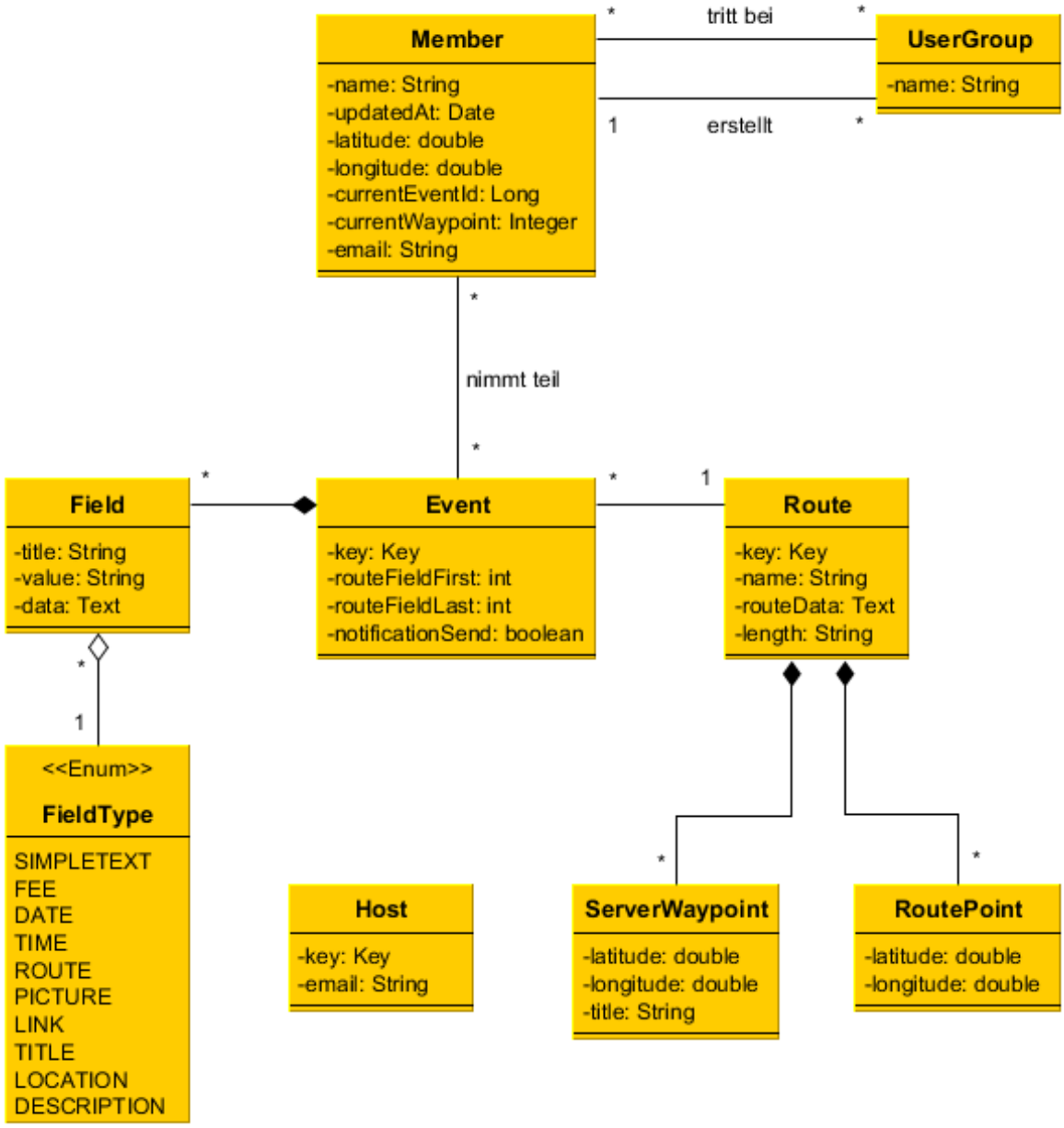


Abbildung 2: UML-Modell der Klassenstruktur des Backends

removeHost

Ähnlich wie die *addHost* -Methode ist die *removeHost* -Methode wieder über ein User-Parameter geschützt. Die Methode löscht den angegebenen Veranstalter, sofern der aufrufende Benutzer ein bereits eingetragener Veranstalter ist. Auch diese Funktion ist in der Rechteverwaltung der Veranstalter-App zugänglich.

isHost

Die *isHost* -Methode prüft für den angegebenen Benutzer, ob dieser ein eingetragener Veranstalter ist. Dies ist z.B. bei der Anmeldung in der Veranstalter-App notwendig.

getHosts

Diese Methode liefert eine Liste aller Veranstalter zurück, die zurzeit auf dem Server hinterlegt sind. Die Liste der Veranstalter kann nur von bereits eingetragenen Veranstaltern abgerufen werden.

createEvent

Zum Erstellen eines Events dient die Methode *createEvent* , die von eingetragenen Veranstaltern aufgerufen werden kann. Zusätzlich zu dem Event kann angegeben werden, ob es sich um einen Editiervorgang handelt und, ob das Datum des Events geändert wurde. Diese Daten sind notwendig, da Benutzer der User-App so über die Änderungen per GCM benachrichtigt werden können.

createMember

Die *createMember* -Methode wird aus der User-App heraus aufgerufen. Über diese Funktion wird sichergestellt, dass auf dem Server für den Benutzer ein *Member* -Objekt existiert, das unter anderem die Liste der Events speichert, an dem der Benutzer teilnimmt, und auch die aktuell gemeldete Position des Benutzers.

updateMemberLocation

Über die Methode *updateMemberLocation* können die Positionen der Benutzer der User-App an den Server übermittelt werden. Neben der neuen Position des Benutzers wird auch die ID des Events, an dem der Benutzer gerade teilnimmt, übertragen. Dies ist für die Feldberechnung auf dem Server notwendig, die ebenfalls über diese Methode angestoßen wird, falls die letzte Feldberechnung länger als einen gewissen Wert zurückliegt.

simulateMemberLocations

Für Präsentationen der App und der Feldauswertung haben wir neben den beiden Apps einen Simulator für Benutzer der Skatenight-App entwickelt. Zunächst war geplant, dass dieser Simulator genau wie die User-App die *updateMemberLocation* -Methode aufruft. Da bei bis zu 100 simulierten Teilnehmern die Anzahl der Serverzugriffe jedoch sehr hoch und damit nicht mehr im kostenlosen Rahmen der Google App Engine lag, haben wir uns dafür entschieden eine gesonderte Methode zur Simulation der Positionen zu erstellen. Diese kann mit nur einem Aufruf eine Liste von Teilnehmern und deren Positionen entgegen nehmen, sodass nun die Anzahl der Serveraufrufe unabhängig von der simulierten Teilnehmerzahl ist. Intern wird jedoch nach wie vor die *updateMemberLocation* -Methode aufgerufen.

getCurrentEventsForMember

Da im Datenmodell in den *Member* -Objekten die Liste der Events, an denen der Benutzer teilnimmt, nicht gespeichert werden, kann über diese Methode eine Liste der entsprechenden Events für einen Teilnehmer abgerufen werden.

getMember

Die *getMember* -Methode dient dem Abrufen der Member-Objekte zu entsprechenden E-Mail-Adressen.

addMemberToEvent

Die *addMemberToEvent* -Methode fügt einen Teilnehmer anhand seiner Mail zu dem Event mit der angegebenen ID hinzu. Dabei wird außerdem geprüft, ob das Event bereits begonnen hat und der Teilnehmer in diesem Fall per GCM informiert.

removeMemberFromEvent

Neben dem Teilnehmen an einem Event muss der Benutzer auch die Möglichkeit haben, das Event wieder zu verlassen. Die Methode *removeMemberFromEvent* realisiert diese Funktion und wird in der User-App aufgerufen.

getMembersFromEvent

Analog zur *getCurrentEventsForMember* -Methode ruft die Methode *getMembersFromEvent* die Liste der Teilnehmer eines Events ab. Zwar werden diese in der Klasse *Event* gespeichert, jedoch nur als Referenz über die Mail-Adresse. Diese Methode wandelt die Mail-Adressen in eine Liste von *Member* -Objekten um, die anschließend zurück gegeben werden.

addRoute

Die *addRoute* -Methode ist eine Methode, die nur von eingetragenen Veranstaltern aufgerufen werden kann. Sie speichert die übergebene Route in der Datenbank und wird aus der Veranstalter-App nach dem Anlegen einer neuen Route aufgerufen.

getRoutes

Diese Methode gibt eine Liste der auf dem Server gespeicherten Routen zurück. Sie wird sowohl aus der User- als auch aus der Veranstalter-App aufgerufen, da in beiden Apps eine Liste der Routen angezeigt werden kann.

deleteRoute

Die *deleteRoute* -Methode bildet das Gegenstück zur *addRoute* -Methode. Sie löscht die angegebene Route, falls es sich bei dem aufrufenden Benutzer um einen eingetragenen Veranstalter handelt.

getEvent

getEvent gibt das *Event* -Objekt mit der angegebenen ID zurück. Die Methode wird sowohl aus der Veranstalter- als auch aus der User-App aufgerufen, wenn der Anwender auf ein Event klickt.

deleteEvent

Die *deleteEvent* -Methode kann von eingetragenen Veranstaltern zum Löschen eines Events aus der Veranstalter-App heraus aufgerufen werden. Die Methode gibt als Boolean-Wert zurück, ob das Event gelöscht werden konnte oder nicht. Da durch die Google Cloud Endpoints die Menge der verwendbaren Rückgabetypen für Methoden eingeschränkt ist, sind unter anderem primitive Datentypen wie boolean oder der durch Java vorgegebene Standardwrapper Boolean nicht möglich. Aus diesem Grund haben wir einen eigenen Boolean-Wrapper implementiert, der als einzigen Zweck das Kapseln eines Boolean-Wertes erfüllt.

editEvent

Um ein Event anzupassen, können Veranstalter die *editEvent* -Methode aufrufen. Die Implementation dieser Methode löscht zunächst das alte Event und legt es anschließend über die *createEvent* -Methode mit den neuen Daten wieder an.

getAllEvents

Für Benutzer der beiden Apps ist die Methode *getAllEvents* ohne Authentifikation zugänglich. Sie gibt eine Liste aller auf dem Server gespeicherter Events zurück. Da durch die Nuztung von JDO und DataNucleus die in den Events eingebetteten Routen jedoch nicht abgerufen werden konnten, wird nach dem Ermitteln der Eventliste jedes Event nocheinmal einzeln abgerufen. So sind auch die Routen in den Event-Objekten initialisiert.

registerForGCM

Von der User-App wird bei jedem Appstart die *registerForGCM* -Methode aufgerufen. Sie übermittelt die aktuelle GCM-ID an den Server, die zur Kommunikation über den Google Cloud Messaging Dienst benötigt wird. Die GCM-ID eines Teilnehmers wird in seinem *Member* -Objekt gespeichert.

getAllUserGroups

getAllUserGroups ruft die Liste aller Benutzergruppen ab. Sie wird von der User-App benutzt.

fetchMyUserGroups

Diese Methode ruft ebenfalls eine Liste der Benutzergruppen ab, schränkt diese jedoch auf Benutzergruppen des angegebenen Benutzers ein. Zusätzlich prüft die Methode, ob *Member* -Objekte auf Gruppen verweisen, die nicht mehr existieren und bereinigt die Daten, falls entsprechende Gruppen gefunden werden.

createUserGroup

Die *createUserGroup* -Methode legt eine Benutzergruppe mit dem angegebenen Namen und dem aufrufenden Benutzer als Ersteller an. Zusätzlich werden die User-Apps der Benutzer per GCM über die neue Gruppe informiert, damit die Liste der Benutzergruppen in bereits geöffneten Apps aktualisiert werden kann. Für den Benutzer ist diese Nachricht nicht sichtbar.

deleteUserGroup

Die *deleteUserGroup* -Methode löscht die Benutzergruppe mit dem angegebenen Namen, sofern der aufrufende Benutzer Ersteller der Gruppe ist.

joinUserGroup

Über die *joinUserGroup* -Methode haben Benutzer der User-App die Möglichkeit, Benutzergruppen anderer Benutzer beizutreten.

leaveUserGroup

Die *leaveUserGroup* -Methode entfernt den Benutzer wieder aus der angegebenen Gruppe. Dies ist nur dann Möglich, wenn der aufrufende Benutzer nicht Ersteller der Gruppe ist, aus der er austreten möchte.

fetchGroupMembers

Mithilfe der *fetchGroupMembers* -Methode kann eine Liste aller Member abgerufen werden, die sich in der angegebenen Benutzergruppe befinden. Dies wird benötigt, wenn die Gruppenmitglieder der eigenen Gruppen auf der Karte während eines Events angezeigt werden sollen.

5.6 Google Cloud Messaging

von Richard

An mehreren Stellen der User-App wird die Google Cloud Messaging API genutzt. Sie ermöglicht es, Nachrichten vom Backend an registrierte Handys zu senden. Dies ist vor allem dann nützlich, wenn sich Daten auf dem Server ändern und der Benutzer darüber informiert werden muss. An folgenden Stellen benutzen wir das Google Cloud Messaging:

- Zur Benachrichtigung aller Benutzer, wenn ein neues Event erstellt wurde. In diesem Fall wird die Liste der Events in der User-App automatisch aktualisiert, falls sie geöffnet ist. Ebenfalls wird dem Benutzer eine Notification-Nachricht angezeigt, die von überall im Handy über die obere Statusleiste abrufbar ist.
- Zur Benachrichtigung aller Benutzer, wenn eine neue Benutzergruppe erstellt wurde. Dabei wird jedoch keine Notification erstellt, sondern lediglich die Liste der Gruppen aktualisiert.
- Zu Benachrichtigung aller Benutzer, wenn eine Benutzergruppe gelöscht wurde. Dies ist notwendig, damit gegebenenfalls die Einstellung beim Benutzer, dass die gelöschte Gruppe auf der Karte angezeigt werden soll, ebenfalls gelöscht wird. Bei dieser Benachrichtigung erhält der Benutzer außerdem eine Notification.

Die ID, die zur Identifikation und damit zur Kontaktaufnahme mit den Benutzern per GCM notwendig ist, wird beim Start der User-App auf dem Handy generiert. Dieses sendet die ID dann an den Server, der die ID für den aufrufenden Benutzer hinterlegt. Zur Verwaltung der GCM-IDs gibt es auf dem Server die Klasse *RegistrationManager* . Diese speichert zu jeder Nutzer-Mail-Adresse eine ID ab, kann aber auch eine Liste aller registrierter GCM-IDs zurückgeben.
Zur Unterscheidung der einzelnen Nachrichten-Typen, die vom Server zu der User-App gesendet werden, existiert die Enumeration *MessageType* . Alle vom Skatenight-Backend generierten Nachrichten enthalten als Payload mindestens das Attribut *type* , das als Wert den Namen des Enumeration-Eintrags hat. Anhand dieses Typs wird auf dem Handy die Nachricht dann durch den GcmIntentService weiter verarbeitet.
Um eine GCM-Nachricht vom Backend zu Senden, wird zunächst ein *Message* -Objekt erstellt. Dieses erhält Daten wie den *collapseKey* , über den mehrere Nachrichten zusammengefasst werden können, falls sie stoßweise beim Benutzer ankommen. Ebenfalls kann die Gültigkeitsdauer der Nachricht angegeben werden. Falls die Nachricht innerhalb dieser Zeitspanne nicht ausgeliefert wurde, wird sie verworfen. Als weitere Einstellung kann angegeben werden, ob die Nachricht direkt zugestellt werden soll oder erst, wenn der Benutzer sein Handy aktiviert. Über ein Objekt vom Typ *Sender* kann die Nachricht dann an die gewünschten GCM-IDs verschickt werden.

5.7 Feldberechnung

von Pascal

Die Feldberechnung soll einen schnellen Überblick über die aktuelle Lage der teilnehmenden Skater ermöglichen. Ursprünglich erschien es logisch den Teil der Strecke farbig hervorzuheben, der zwischen dem ersten und dem letzten Skater liegt. Bei diesem Ansatz ergaben sich jedoch

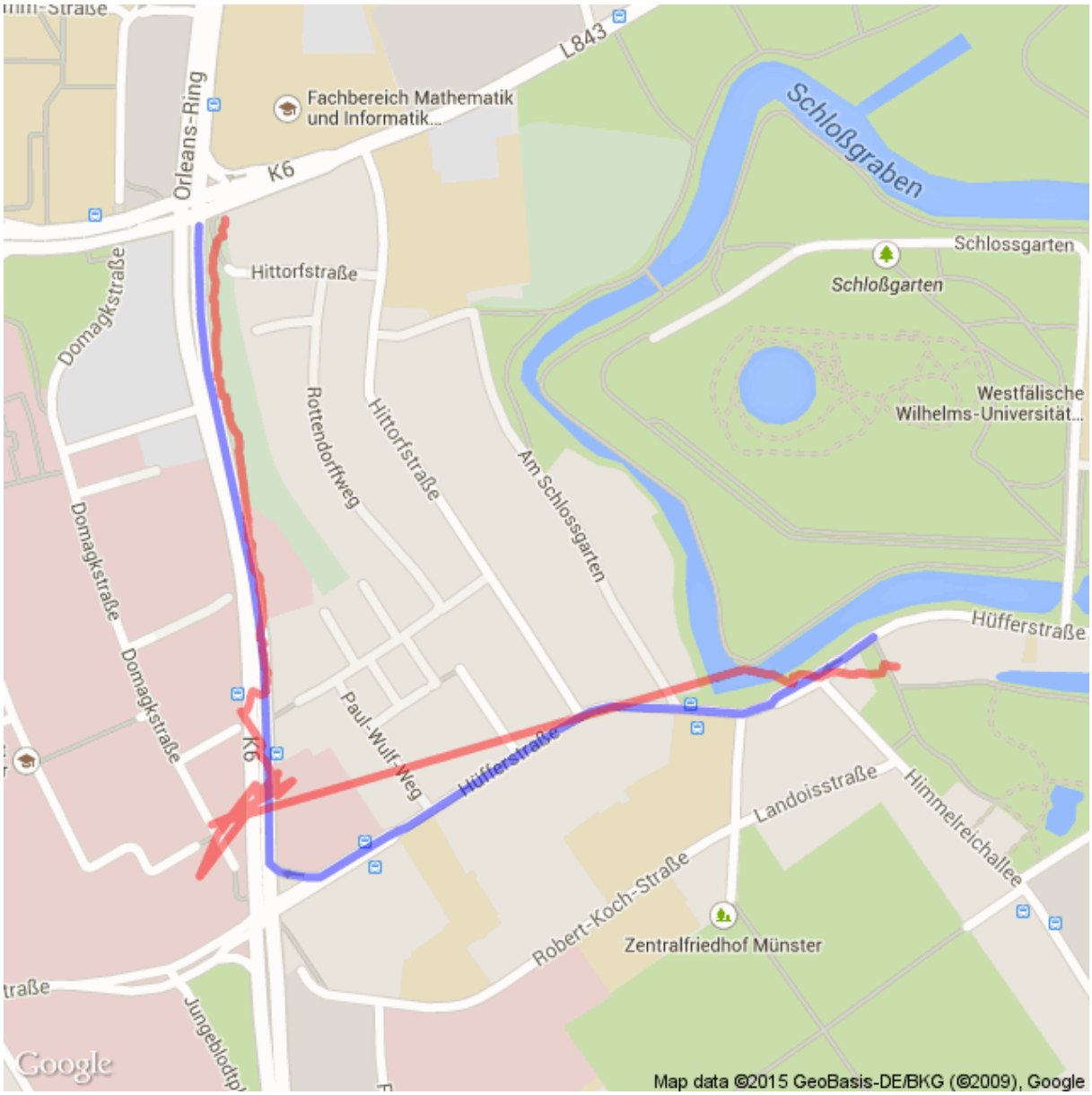


Abbildung 3: Rot: GPS-Aufzeichnung, Blau: tatsächlich zurückgelegte Strecke

einige Probleme welche die Aussagekraft dieser Daten infrage stellten. Zum Beispiel würde ein Skater, der die offizielle Strecke verlässt und diese später auf einem anderen Streckenabschnitt wieder betritt, das Feld künstlich in die Länge ziehen, da sich an der markierten Strecke keine Skater befinden. Ein ähnliches Phänomen würde auftreten, wenn ein Teilnehmer z.B. durch einen Unfall am Weiterfahren gehindert werden würde. Das errechnete Feld hätte in diesen Fällen keinen praktischen Nutzen. Es wird somit ein intelligenter Feldberechnungsalgorithmus benötigt, welcher eine automatische Fehlerkorrektur durchführt. Ein weiteres Problem ist die relative Ungenauigkeit der vom GPS ermittelten Positionen. Diese führt zu einer falschen Darstellung der Strecke (siehe Abb. 3). Um dieses Problem zu beheben liegt es nahe die ermittelten Positionen der Teilnehmer auf die tatsächliche Strecke zu projizieren.
Unser Algorithmus besteht aus zwei Teilen: Der Berechnung des aktuellen Routenpunktes auf der Strecke und der anschließenden Berechnung des Feldes.

5.7.1 Routenpunktberechnungsalgorithmus

Jeder Teilnehmer sendet in regelmäßigen Abständen mithilfe der *updateMemberLocation(String, double, double, long)* im *SkatenightServerEndpoint* seine Position an den Server. An dieser Stelle berechnet der Server mit der *calculateCurrentWaypoint(PersistenceManager, Member)* dessen aktuellen Routenpunkt.
Zunächst wird überprüft ob sich der Teilnehmer bereits an einem Routenpunkt befindet. Falls dies nicht zutrifft, wird er temporär dem ersten Routenpunkt der Strecke zugeordnet. Wenn der aktuelle Routenpunkt des Teilnehmers nicht der Endpunkt der Route ist, wird nun der tatsächliche Routenpunkt ermittelt. Es wird nun die Distanz zwischen den GPS-Koordinaten des Teilnehmers und denen des aktuellen und sequentiell nächsten Routenpunktes ermittelt. Ist die Distanz zum nächsten Routenpunkt kleiner als die zum Aktuellen und kleiner als *MAX_NEXT_WAYPOINT_DISTANCE* (10 m), wird der nächste Routenpunkt als Aktueller gesetzt. Falls dies nicht der Fall ist, wird überprüft ob sich der Teilnehmer noch am aktuellen Routenpunkt befindet indem die Distanz zu diesem ebenfalls mit der *MAX_NEXT_WAYPOINT_DISTANCE* verglichen wird. Ist beides nicht der Fall, wird unter allen noch folgenden Routenpunkten der Strecke nach dem am nächsten liegenden gesucht, der nicht mehr als *MAX_ANY_WAYPOINT_DISTANCE* (50) Meter entfernt ist. Andernfalls hat der Teilnehmer die Strecke verlassen.

5.7.2 Feldberechnungsalgorithmus

Der Feldberechnungsalgorithmus wird ebenfalls von der *updateMemberLocation(String, double, double, long)* Methode des *SkatenightServerEndpoint* aufgerufen. Nachdem der aktuelle Routenpunkt des Teilnehmers berechnet wurde, wird überprüft ob die letzte Berechnung des Feldes mehr als 30 Sekunden in der Vergangenheit liegt. Ist dies der Fall, wird das Feld neu berechnet. Da der Routenpunktberechnungsalgorithmus die Positionen der Teilnehmer auf die Strecke projiziert können wir nun das Feld anhand der Routenpunkte berechnen. Allerdings würden in diesem Zustand die Positionen einzelner Skater noch zu stark gewichtet. Um das Problem zu lösen nutzen wir für die Feldberechnung zusätzlich zu den Positionen auch die Anzahl der teilnehmenden Skater die sich gleichzeitig an einem Routenpunkt befinden. Zunächst wird eine Liste von Integern erstellt, welche an der Stelle n die Anzahl der Teilnehmer enthält die sich an dem Routenpunkt n befinden. Es wird nun der Routenpunkt zum Feld hinzugefügt, an dem sich die meisten Teilnehmer befinden. Von diesem Routenpunkt ausgehend werden umliegende Routenpunkte zum Feld hinzugefügt an denen sich mindestens *MIN_WAYPOINT_MEMBER_COUNT* (5) Teilnehmer befinden. Ist zwischen zwei Routenpunkten mit mindestens *MIN_WAYPOINT_MEMBER_COUNT* (5) Teilnehmern ein Punkt mit weniger Skatern, so wird dieser trotzdem im Feld berücksichtigt. Bei praktischen Tests ist uns aufgefallen, dass durch die kurze Distanz zwischen einigen Routenpunkten das Limit von einem Routenpunkt der weniger als *MIN_WAYPOINT_MEMBER_COUNT* (5) Skater beherbergt möglicherweise nicht ausreicht um die präziseste Berechnung des Feldes zu gewährleisten. Basierend auf dieser Berechnung werden jeweils der erste und der letzte Routenpunktindex des Feldes im Event als *routeFieldFirst* und *routeFieldLast* hinterlegt.

6 Frontend

6.1 Beschreibung App-Gerüst (Gemeinsamkeiten beider Apps)

von Martin und Bernd

Das Projekt umfasst zwei Apps, eine App für die Veranstalter(Veranstalterapp) von Skatenights und eine App für die Teilnehmer(Userapp) von Skatenights. In beiden Apps hat man

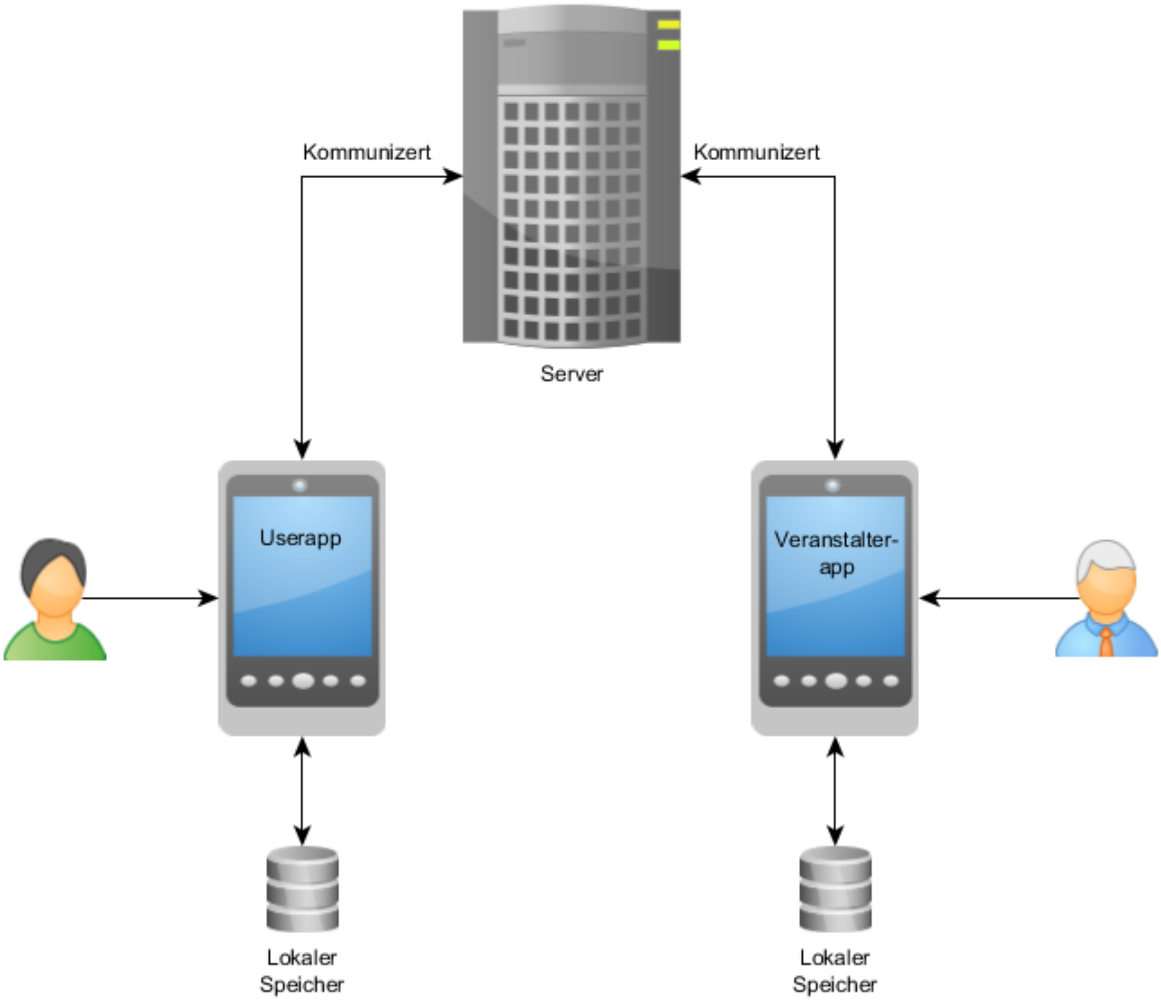


Abbildung 4: Ein Interaktionsmodell der User- und Veranstalter-App

eine Liste von Veranstaltungen gegeben und man kann einzelne Veranstaltungen aus dieser Liste betrachten. In der Veranstalterapp hält ein Fragment, das *ShowEventsFragment* , die Liste von Veranstaltungen und in der Userapp wird dies mit einer Activity, der *ShowEventsActivity* , realisiert. Die Liste ist ein *ListView* Element, welches in der dazugehörigen xml Datei des Fragments/ der Activity definiert wird, das über einen Adapter gefüllt wird. In beiden Apps wird dazu der gleiche Adapter, der *EventsCursorAdapter* , benutzt. Auf den Adapter wird näher in Abschnitt 4.1 eingegangen. Nach dem Auswählen einer Veranstaltung aus der Liste wird eine Activity gestartet, um die Informationen der Veranstaltung anzuzeigen. In den Informationen einer Veranstaltung kann man sich die Karte über eine neue Activity anschauen. Um auf den Server zuzugreifen benutzen beide Apps sogenannte Tasks. Ein Task ist eine Java Klasse, welche *AsyncTask* erweitert und bei Aufruf einen neuen Thread erstellt um mit dem Server zu kommunizieren. Ein Task hat meist zwei implementierte Methoden, die *doInBackground(...)* und die *onPostExecute(...)* . Der Server Zugriff erfolgt meist in der *doInBackground(...)* Methode, die *onPostExecute(...)* Methode wird erst nach kompletten Durchlauf der *doInBackground(...)* Methode aufgerufen. Diese Methode wird meistens dazu genutzt um das Ergebnis des Tasks an die Userapp bzw. an die Veranstalterapp zu übergeben. Folgende Tasks werden sowohl in der Userapp als auch in der Veranstalterapp genutzt:
GetEventTask : Dieser Task erwartet die ID einer Veranstaltung und ruft dann die dazu passende Veranstaltung vom Server ab, null falls keine Veranstaltung mit der angegebenen ID existiert.

Der Aufruf erfolgt, wenn ein Benutzer eine Veranstaltung in der Liste auswählt um sich die Informationen anzeigen zu lassen.
QueryEventsTask : Dieser Task ruft die komplette Liste von Veranstaltungen vom Server ab und gibt diese aus, null falls keine Veranstaltungen auf dem Server existieren. Der Aufruf erfolgt, wenn ein Benutzer die Userapp bzw. die Veranstalterapp startet oder die Liste in der Userapp aktualisiert wird.

6.2 Beschreibung Veranstalter-App

6.2.1 Gerüst

von Martin und Bernd

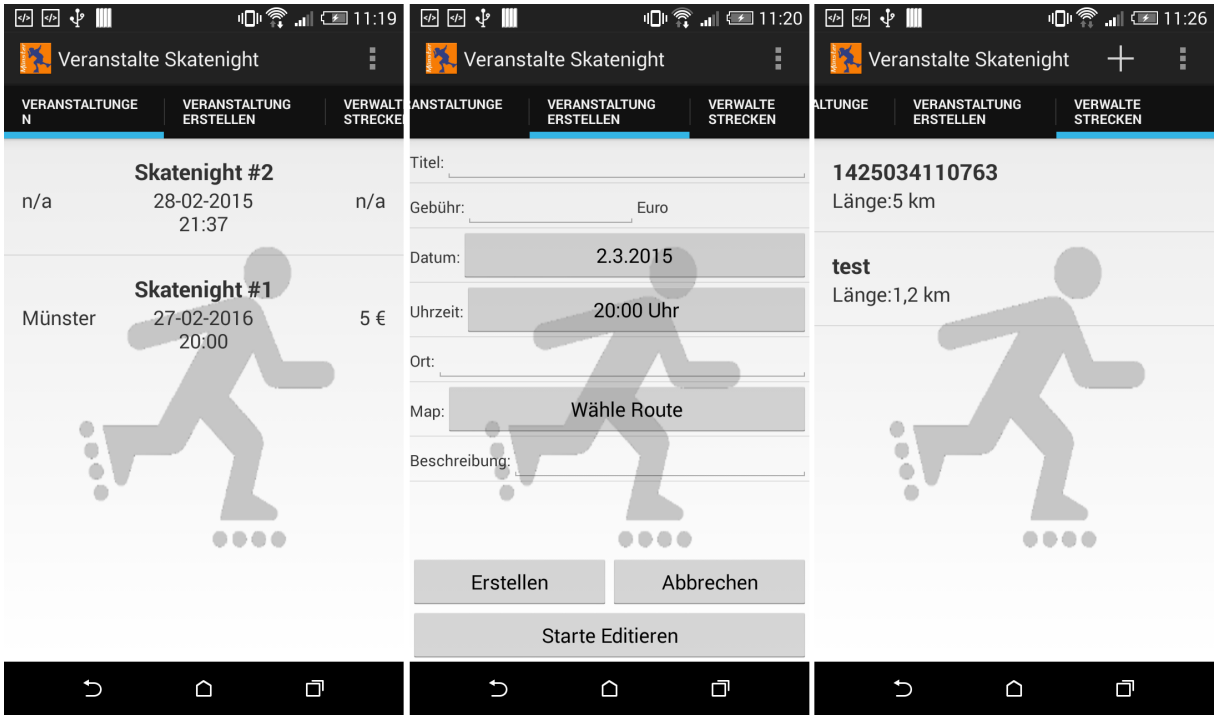


Abbildung 5: Eine Übersicht über die in der Veranstalter-App angebotenen Fragments

Um zu den eigentlichen Funktionen der Veranstalterapp zu gelangen muss man sich zu Beginn in der LoginActivity mit seinem Google-Account einloggen. Ist der Login erfolgreich, so wird die *HoldTabsActivity* gestartet.
Die Veranstalterapp besteht aus einer Activity, der *HoldTabsActivity* , die einen Viewpager nutzt um zwischen den Funktionalitäten der App zu navigieren. Ein Viewpager ist eine UI-Komponente, welche dafür sorgt, dass man zwischen verschiedenen Tabs über horizontales Scrollen bzw. über Klicken der Tabs in der Tab-Leiste (siehe Abbildung) wechseln kann. Jeder einzelne Tab ist in dem Fall ein einzelnes Fragment. Dazu ist ein Adapter erforderlich, der entscheidet welches Fragment angezeigt werden soll, wenn gescrollt wird, bzw. wenn auf einen der Tabs geklickt wird. Dieser Adapter(*TabsPagerAdapter*) ist eine Klasse, welche *FragmentPagerAdapter* erweitert und zwei Methoden benötigt um Fragmente zu verwalten. Der *TabsPagerAdapter* hält eine Liste der Fragmente die im Konstruktor dieser hinzugefügt werden. Eine der beiden Methoden ist die *getItem(...)* Methode. Diese wird aufgerufen, wenn ein Tab berührt wird oder in einem Fragment horizontal gescrollt wird und liefert das Fragment mit dem übergebenen Index in der Liste zurück. Durch die *getCount(...)* Methode weiß der Adapter wie viele Tabs höchstens

angezeigt werden sollen. Wenn die Activity gestartet wird, dann wird in der *onCreate(...)* Methode der ViewPager, der Adapter(*TabsPagerAdapter*) und die Namen der Tabs initialisiert. Anschließend muss dieser Adapter dem ViewPager hinzugefügt werden, damit die Fragmente angezeigt werden können. Es werden insgesamt drei Fragmente dem Adapter hinzugefügt. Das *ShowEventsFragment* dient dem Anzeigen aller Veranstaltungen und ist das Fragment, das beim Start der Veranstalterapp angezeigt wird. In der Liste von Veranstaltungen kann man genauere Informationen einzelner Veranstaltungen durch das Auswählen betrachten, ändern und auch löschen. Das *AnnounceInformationFragment* dient zum Erstellen neuer Veranstaltungen. Man hat hier die Möglichkeit weitere Angaben zu der Veranstaltung hinzuzufügen, als die standardmäßig festgelegten. Dies wird in Abschnitt 6.2.4 genauer beschrieben. Das *ManageRoutesFragment* dient zum Verwalten aller auf dem Server vorhandenen Routen. Man hat hier die Möglichkeit Routen hinzuzufügen, zu löschen und als Vorlage für neue Routen zu benutzen. Des weiteren ist nur wenn dieses Fragment angezeigt wird der „Routen hinzufügen"-Button in der ActionBar sichtbar. Dazu erweitert das *ManageRoutesFragment* das OptionsMenu der *HoldTabsActivity* um den zusätzlichen Button in der *onCreateOptionsMenu(...)* Methode. Damit dies umgesetzt wird muss beim Erstellen des Fragmentes in der *onCreate(...)* Methode mit dem Aufruf *setHasOptionsMenu(...)* signalisiert werden, dass eine Erweiterung der ActionBar vorliegt. Die *HoldTabsActivity* bietet die Möglichkeit über das Menu der ActionBar die Rechteverwaltung aufzurufen.

6.2.2 Hosts

von Bernd

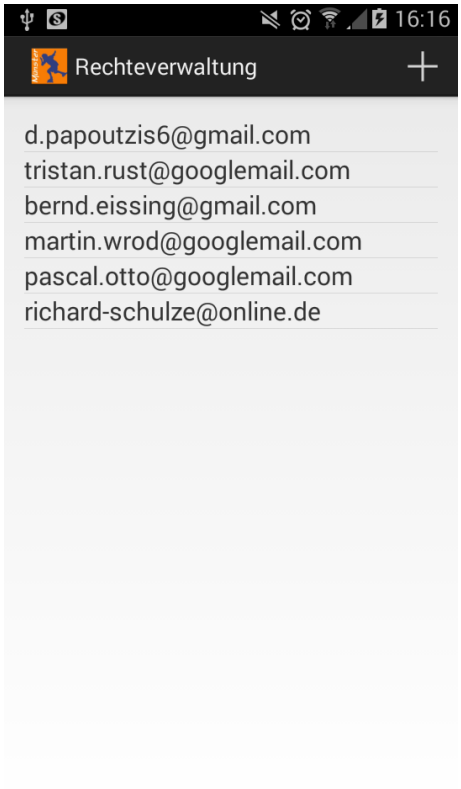


Abbildung 6: Die Liste der eingetragenen Veranstalter

Ein Host ist ein Veranstalter, welcher autorisiert ist die Veranstalterapp zu starten und die dor-

tigen Funktionen zu nutzen. Ein Host ist eine Klasse auf dem Server, mit einem Attribut Key für den Schlüssel eines Hostobjekts und einem String für die E-Mail. In der Veranstalterapp ist es Hosts erlaubt andere Hosts hinzuzufügen und auch zu löschen. Um dies zu tun muss man die Veanstalterapp starten und dann das Optionsmenü aufrufen. Dort kann man dann „Rechte“ auswählen und man wird zu der Recheverwaltung weitergeleitet. Die Rechteverwaltung ist lediglich eine Activity(*PermissionManagementActivity*) mit einer ListView für die Liste der Hosts, die zum Zeitpunkt des Aufrufs der Activity auf dem Server existieren. Wenn die Activity gestartet wird, fragt diese in der *onCreate(..)* Methode alle Hosts vom Server ab. Dazu wird der *QueryHostsTask* aufgerufen, welcher die Liste vom Server in der *doInBackground(...)* Methode abfragt, und diese dann in der *onPostExecute(...)* an die Activity weiter gibt, indem der Task die *setHostsToListView(...)* in der Activity aufruft. Diese Methode in der Activity erstellt einen neuen Adapter(*HostCursorAdapter*) welcher der ListView hinzugefügt wird.

Hinzufügen und Löschen von Hosts

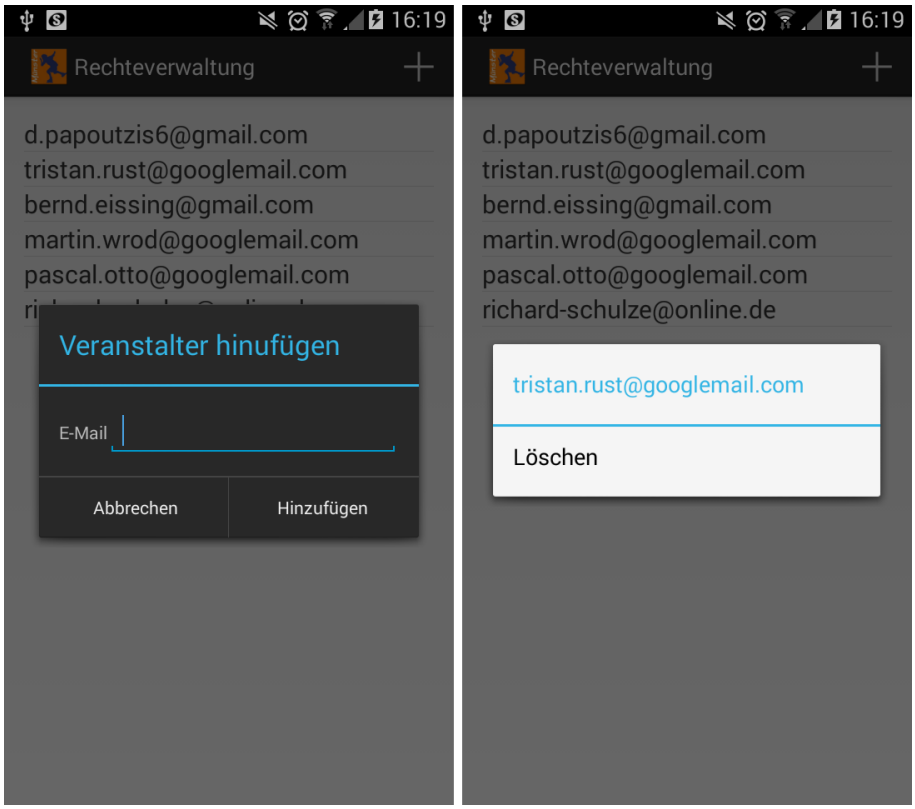


Abbildung 7: GUI zum Hinzufügen und Löschen von Veranstaltern

Es sind Optionen zum Hinzufügen und Löschen von Hosts vorhanden. Zum Löschen eines Hosts muss man im Actionmenu den Plusbutton betätigen. Dadurch wird eine neue Activity(*AddHostDialog*) gestartet, welche einen EditText für die E-Mail des neuen Host und zwei Buttons für das Abbrechen und das Bestätigen des Vorgangs hat. An dieser Stelle wird nicht geprüft, ob es sich bei dem angegebenen Text um eine E-Mail handelt oder nicht; es wird lediglich abgefragt, ob der angegebene Text nicht leer ist. Der Abbrechen-Button beendet den Vorgang. Bei Eingabe und Bestätigung durch Drücken des Hinzufügen Buttons wird die *apply(...)* Methode aufgerufen, welche den *AddHostTask* startet. Dieser Task erwartet einen String und ruft dann auf dem Server die *addHost(...)* Methode auf. Veranstalter können auch aus der Liste von Veranstalter gelöscht werden. Dazu muss man die gewünschte E-Mail eines Veranstalters

in der Liste berühren. Es wird ein AlertDialog gestartet, welcher als Überschrift die E-Mail des ausgewählten Veranstalters hat und einen Löschen Button. Bei Betätigung des Löschen Buttons wird der *DeleteHostTask* aufgerufen, welcher einen String für die E-Mail eines Hosts erwartet und damit dann in der *doInBackground(...)* Methode die *removeHost(...)* Methode auf dem Server aufruft. In der *onPostExecute(...)* Methode wird dann die Liste von Veranstaltern in der Activity durch den Aufruf von *refresh(...)* aktualisiert. Momentan kann jeder Veranstalter alle möglichen Strings als Veranstalter hinzufügen, da nicht geprüft wird ob es sich um den angegebenen Text um eine E-Mail handelt. Des weiteren kann ein Veranstalter sich selber löschen, was dazu führt, dass dieser in der Veranstalterapp keine Funktionen mehr betätigen kann aber dennoch eingeloggt ist.

6.2.3 Routen

von Pascal

Das Erstellen und Bearbeiten von Routen ist ein integraler Bestandteil der Veranstalter-App. Dem Nutzer sollte es möglich sein auf intuitive Weise Strecken zu erstellen ohne diese selber abzulaufen oder umständlich manuell auf der Karte zu zeichnen. Stattdessen sollte die Strecke möglichst automatisch über die bestehenden Straßen gelegt werden. Da das Erstellen einer Strecke eine einzelne unabhängige Aktivität darstellt, wird mit der *RouteEditorActivity* eine eigene Activity bereitgestellt die sich allein mit dieser Aufgabe beschäftigt. Diese Activity beinhaltet einen ViewPager welcher zwei Fragments verwaltet zwischen denen sowohl über die Tab Leiste, als auch durch horizontales wischen gewechselt werden kann (genauere Beschreibung in Abschnitt 6.1). Verwaltet werden das *EditorMapFragment* , welches die Darstellung und Bearbeitung der Strecke auf der Karte ermöglicht und das *EditorWaypointsFragment* , welches die Bearbeitung einzelner Wegpunkte in der Liste ermöglicht. Die Fragment Klassen beinhalten lediglich die Methoden zur Abstraktion des UI welche von der eigentlichen Programmlogik in der *RouteEditorActivity* aufgerufen werden.
Von der *RouteEditorActivity* verwaltete Daten werden hauptsächlich durch zwei statische innere Klassen repräsentiert. Diese kombinieren die von der „Google Maps Android API v2“ gestellten Klassen zur Darstellung der Daten auf der Strecke mit zusätzlichen Metainformationen. Um die Verwendung dieser Klassen zu erleichtern implementieren sie das Parcelable Interface. Somit können sie beispielsweise in der *onSaveInstanceState(Bundle)* problemlos im Bundle hinterlegt werden. Referenzen auf UI Elemente werden jedoch nicht gespeichert (siehe Abschnitt 6.2.3). Die erste dieser Klassen ist die *RouteEditorActivity.Route* , welche den Streckenverlauf der berechneten Route darstellt. Sie kombiniert die von Google Maps genutzten *PolylineOptions* und *Polyline* mit der als int gespeicherten Länge der Strecke in Metern. Die zweite der Klassen ist die *RouteEditorActivity.Waypoint* , welche einen einzelnen Wegpunkt auf der Strecke darstellt. Diese bestehen aus den von Google Maps genutzten *MarkerOptions* und *Marker* .

Google Maps Android API v2

Die Google Maps Android API v2 ist Teil der Google Play Services und bietet die Möglichkeit einfach zu verwendende, interaktive Karten in Android Apps zu verwenden. Kernstück der API ist die *GoogleMap* Klasse, welche Interaktion mit der dargestellten Karte ermöglicht. Zugriff erfolgt wahlweise über die *getMap()* Methoden der *MapView* oder *MapFragment* Klassen, welche zur Darstellung der Karte verwendet werden. Da die Karte Zugriff auf den Activity-Lifecycle benötigt, wird von Google die Verwendung des *MapFragment* empfohlen. Die *GoogleMap* stellt eine Vielzahl von Möglichkeiten bereit die Karte der erforderten Nutzung anzupassen. Insgesamt werden 6 mögliche Varianten für Kartenannotationen bereitgestellt: Circle, GroundOverlay, Marker, Polygon, Polyline und TileOverlay. Für die Planung der Route benötigen wir jedoch lediglich *Marker* für die Darstellung von Wegpunkten und eine *Polyline* für die Darstellung des Streckenverlaufs. Diese Annotationen bestehen jeweils aus zwei Teilen. So repräs-

sentiert die *MarkerOptions* Klasse Anweisungen um einen *Marker* auf der Karte darstellen zu können. Um diesen tatsächlich darzustellen wird die *addMarker(MarkerOptions)* Methode der *GoogleMap* aufgerufen. Diese liefert eine Referenz auf das gezeichnete *Marker* Objekt zurück. Da die *MarkerOptions* lediglich die Anweisungen zum Zeichnen des Markers repräsentieren, müssen Änderungen wie z.B. ein neuer Titel an das *Marker* Objekt übermittelt werden. Da die *addMarker(MarkerOptions)* die einzige Möglichkeit darstellt auf dieses zuzugreifen, ist es wichtig das Objekt für eine mögliche spätere Verwendung zu behalten. Zudem muss bedacht werden, dass die *Marker* nur zeitlich begrenzt gültig sind. Wird z.B. durch drehen des Gerätes das UI neu gezeichnet, werden die Marker ungültig. Für diesen Fall sollten die *MarkerOptions* gespeichert werden um die erneute Darstellung zu ermöglichen. Dieses Konzept überträgt sich auf die anderen Annotationstypen z.B. mit *PolygonOptions* und *Polygon* , *PolylineOptions* und *Polyline* , usw. Diese bieten jeweils eigene Schnittstellen um sie auf die Bedürfnisse der Anwendung anzupassen.
Ein Marker besteht aus einem Bild, welches auf der Karte angezeigt wird. Bei Berührung des Bildes wird der Titel des Markers angezeigt. Die Position des Bildes kann mithilfe eines *LatLng* Objektes gesetzt werden, welches aus jeweils einem double Wert für Längen- und Breitengrad besteht. Marker können zudem, falls dies erfordert wird, vom Nutzer nächträglich per drag-and-drop verschoben werden. Die Funktionalität wird über ein einzelnes Flag gesteuert, auf welches man mithilfe der *draggable(boolean)* Methode des *Marker* bzw. *MarkerOptions* Objektes zugegreifen kann. Auf die neue Position kann über die *getPosition()* Methode zugegriffen werden. Zudem besteht die Möglichkeit vom *GoogleMap* Objekt über Veränderungen der Marker durch den Nutzer benachrichtigt zu werden. Dies wird mithilfe des *GoogleMap.OnMarkerDragListener* Interfaces realisiert, welches beim *GoogleMap* Objekt über die *setOnMarkerDragListener(GoogleMap.OnMarkerDragListener)* Methode registriert werden kann. Bei Änderungen eines Markers werden danach die *onMarkerDragStart(Marker)* , *onMarkerDrag(Marker)* und *onMarkerDragEnd(Marker)* Methoden aufgerufen.
Eine *Polyline* stellt eine Linie dar die auf der Karte gezeichnet wird. Sie besteht hauptsächlich aus einer Farbe, welche die Farbe der Linie kontrolliert und einem float Wert, der die Breite der Linie kontrolliert. Der Verlauf der Linie kann mithilfe der *addAll(List<LatLng>)* Methode festgelegt werden. Zugriff auf diese Liste ist über die *getPoints()* Methode möglich.

Google Directions API[1]

Über die Google Directions API stellt Google die in Google Maps integrierte Funktionalität zur Planung von Routen zur Verfügung. Per HTTP-Anfrage kann die benötigte Route wahlweise im XML oder JSON Format angefordert werden. Android verfügt bereits über die Funktionalität zum parsen von XML und JSON, sodass wir uns nach der Empfehlung von Google richten und das JSON Format verwenden. In der kostenlosen Variante sind die Anzahl der Anfragen auf 2500 pro Tag limitiert. Da die API jedoch ausschließlich beim Eintragen der Route durch die Veranstalter genutzt wird, kann dies vernachlässigt werden. API Anfragen richten sich an „http://maps.googleapis.com/maps/api/directions/“ gefolgt von dem gewünschten Ausgabeformat. In diesem Fall folgt also „.../directions/json?“. Anschließend können eine Reihe von Parametern in beliebiger Reihenfolge übergeben werden, die durch „&“ getrennt werden. Zunächst gibt es jeweils einen Parameter für Ausgangs- und Zielpunkt. Diese können als „origin=“ und „destination=“ gefolgt von den gewünschten Adressen übergeben werden. Adressen werden automatisch zu Koordinaten umgewandelt. Alternativ können die Koordinaten auch direkt als Längen- und Breitengrad getrennt durch ein „,“ übergeben werden. Um den Routenverlauf genauer zu Steuern, können zusätzlich Wegpunkte übermittelt werden. Dies erfolgt mithilfe des „waypoints=“ Parameters gefolgt von einer Reihe von Adressen, die das gleiche Format wie Start- und Endpunkt unterstützen. Die einzelnen Adressen werden durch ein „|“ getrennt. Übergebene Wegpunkte werden in der angegebenen Reihenfolge in die Route integriert. An dieser Stelle fällt bei der

[1] https://developers.google.com/maps/documentation/directions/

kostenlosen Variante der API eine weitere Einschränkung an. So dürfen neben dem Start und Endpunkt maximal 8 Wegpunkte angegeben werden. Zuletzt gibt es noch den „sensor=“ Paramter welcher der API über einen Boolean Wert mitteilt ob das Gerät über die Möglichkeit verfügt seinen Standort zu ermitteln. Eine einfache Route an der Straße um das Hörsaalgebäude an der Einsteinstraße über zwei Wegpunkte lässt sich somit über die folgende Anfrage planen:
http://maps.googleapis.com/maps/api/directions/json?origin=51.965516,7.603414
&destination=51.965674,7.604188&waypoints=51.968710,7.603574|51.967477,7.606374
&sensor=true
Der zurückgegebene JSON String enthält ein Objekt „status“, welches den Statuscode der Anfrage enthält, und ein Array „routes“, welches die gefundenen Routen enthält. Konnte keine Route gefunden werden, ist „routes“ leer und Status enthält den Grund für den Fehler. Die genauen Fehlercodes finden sich in der Dokumentation unter „Statuscodes“. Gefundene Routen bestehen aus einer Anzahl von Metainformationen und einer detaillierten Beschreibung der Route. Da für die Anwendung lediglich der Streckenverlauf relevant ist, können die meisten dieser Informationen ignoriert werden. Die Beschreibung der Route ist unterteilt in mehrere Streckenabschnitte, die sich im „legs“ Array der Route befinden. Diese Streckenabschnitte enthalten neben einer Zusammenfassung des Abschnittes die genauen Anweisungen im „steps“ Array. Jede Anweisung besteht unter anderem aus dem genauen Straßenverlauf („polyline.points“), Informationen über den Abschnitt (z.B. „distance.value“) und einer textuellen Fahranweisung („html_instructions“).

RouteEditorActivity Funktionalität

Die *RouteEditorActivity* ist die Zentrale Klasse des Interface zum Erstellen von Routen. Beim starten der Activity wird der Name der Route als String Extra (*RouteEditorActivity.EXTRA_NAME*) erwartet. Dieser wird im Member *name* gespeichert. Mithilfe des *RouteEditorActivity.EXTRA_WAYPOINTS* können außerdem Wegpunkte als Ausgangspunkt für die Strecke übergeben werden. Zur Verwaltung der Wegpunkte wird mit dem *WaypointAdapter* ein modifizierter ArrayAdapter verwendet, welcher angepasst wurde um die Namen der Wegpunkte aus den *MarkerOptions* des *Waypoint* zu lesen. Das *Route* Objekt, welches die aktuell berechnete Strecke repräsentiert wird im Member *route* hinterlegt. Falls keine Route vorhanden ist, nimmt dieses den Wert null an. Damit beim Drehen des Bildschirms keine Berechnungen verloren gehen, werden Name, Route und Wegpunkte im Bundle gespeichert. Da die *Route* und *Waypoint* Klassen das Parcelable Interface implementieren, können diese im Bundle mithilfe der *putParcelable(String, Parcelable)* bzw. *putParcelableArray(String, Parcelable[])* gespeichert werden. Um eine Strecke zu erstellen müssen zunächst mindestens zwei Wegpunkte hinzugefügt werden. Das Erstellen neuer Wegpunkte geschieht mithilfe der *addWaypoint()* Methode. Diese kann von beliebiger Stelle aufgerufen werden. In der App geschieht dies ausschließlich über einen Button in der ActionBar. Der neue *Waypoint* wird mithilfe der *addWaypoint(int)* Methode des *EditorMapFragment* erstellt, welche den Mittelpunkt der Karte ermittelt und diesen als Position des Waypoints übernimmt. Außerdem wird der Titel des *Waypoint* gemäß des übergebenen Index auf „Wegpunkt X“ gesetzt. Mithilfe des *MarkerOptions* Objektes des *Waypoint* kann der Wegpunkt nun der Karte hinzugefügt werden. Die Referenz für das entstehende *Marker* Objekt wird im *Waypoint* gespeichert. Anschließend versucht die *RouteEditorActivity* die Strecke durch den Aufruf der *loadRoute()* Methode zu berechnen. Diese entfernt zunächst mithilfe der *setRoute(Route)* Methode die aktuell angezeigte Strecke von der Karte und berechnet anschließend mithilfe des *RouteLoaderTask* eine aktuelle Route, wenn nach der Änderung mindestens zwei Wegpunkte vorhanden sind. Damit der Nutzer weiß, dass die Route berechnet wird, wird in der ActionBar eine auf *indeterminate* gesetzte ProgressBar angezeigt. Beim Beenden des Task, sei es durch Erfolg oder Fehler, wird diese ausgeblendet und die neue Strecke bzw. eine Fehlermeldung angezeigt. Erfolgt ein weiterer Aufruf der Methode, bevor der Task beendet wird, so wird nicht auf dessen Ende gewartet. Stattdessen wird sein Ergebnis ignoriert und auf das Ergebnis des neuen Tasks gewartet. Im Member *currentTask*

wird dafür eine Referenz auf den Task behalten, dessen Ergebnis dem zeitlich neuesten Aufruf der *loadRoute()* Methode entspricht.
Da die Wegpunkte im *WaypointAdapter* gespeichert werden, spiegeln sich Änderungen direkt in der ListView im *EditorWaypointsFragment* wieder.
Das Fragment behandelt insgesamt nur zwei Funktionalitäten. Zum einen wird ein *AdapterView.OnItemClickListener* verwendet, welcher beim Berühren eines Wegpunktes in der Liste das *EditorMapFragment* im ViewPager aktiviert und mithilfe dessen *showWaypoint(Waypoint)* Methode den Wegpunkt auf der Karte vergrößert und dessen Titel anzeigt. Außerdem wird ein *AdapterView.OnItemLongClickListener* verwendet, welcher bei seiner Aktivierung durch ein langes drücken auf einen der Wegpunkte in der Liste die Option zum Benennen und Löschen dieses Wegpunktes anzeigt. Beim Löschen wird die *removeWaypoint(int)* Methode der *RouteEditorActivity* aufgerufen welche den Wegpunkt aus dem Adapter entfernt, ihn beim *EditorMapFragment* mithilfe der *removeWaypoint(Waypoint)* von der Karte entfernt, die Indizes der folgenden Wegpunkte aktualisiert und mithilfe der *loadRoute()* falls möglich eine neue Route berechnet.
Auch das *EditorMapFragment* bietet eine Möglichkeit den Streckenverlauf zu ändern. So implementiert dieses das *GoogleMap.OnMarkerDragListener* Interface, welches von der *GoogleMap* über das Verschieben eines der Marker benachrichtigt wird. Wir müssen lediglich auf den Aufruf der *onMarkerDragEnd(Marker)* Methode warten, um den *Waypoint* über die Positionsänderung zu benachrichtigen. Da beim Verschieben des Marker lediglich die Position des Marker beeinflusst wird, muss diese Änderung auf das *PolylineOptions* Objekt übertragen werden. Anschließend wird noch die *loadRoute()* Methode der *RouteEditorActivity* aufgerufen damit diese die geänderte Route berechnen kann.

RouteLoaderTask Funktionalität

Der *RouteLoaderTask* ist für die eigentliche Routenberechnung verantwortlich. Er interagiert dafür mithilfe von HTTP-Anfragen mit der Google Directions API. Da es sich bei diesen Interaktionen um Netzwerkanfragen handelt, die möglicherweise längere Ladezeiten mit sich bringen, dürfen diese nicht auf dem UI-Thread ausgeführt werden, damit dieser nicht unnötig blockiert wird. Das Android SDK stellt hier in Form des *AsyncTask* eine nützliche Klasse bereit, welche eine zusammenhängende Berechnung auf einem Hintergrund-Thread repräsentiert. In seiner ursprünglichen Form bietet der *AsyncTask* mithilfe der *onProgress(Progress)* und *onPostExecute(Result)* Methoden die Interaktion mit dem GUI. In der *common* Bibliothek findet sich mit dem *ExtendedTask* eine Unterklasse des *AsyncTask* , welcher um die Funktionalität erweitert wurde den Fortschritt, das Ergebnis und mögliche Fehlermeldungen direkt an ein *ExtendedTaskDelegate* zu übertragen. Die *RouteEditorActivity* implementiert dieses Interface zu diesem Zweck.
In der *doInBackground(ArrayAdapter<Waypoint>...)* Methode findet die Routenberechnung und die anschließende Auswertung statt. Diese wird im Hintergrund-Thread des *ExtendedTask* ausgeführt. Mithilfe der *publishProgress(Progress)* und *publishError(String)* Methoden kann mit der *ExtendedTaskDelegate* interagiert werden. Dabei ist zu beachten, dass nach einem Aufruf der *publishError(String)* Methode ein eventuell übergebenes Ergebnis nicht an die Delegate weitergeleitet wird. Die erste Aufgabe des Task ist die Erstellung der HTTP-Anfrage, nach dem in Kapitel 6.2.3 beschriebenen Schema. Dafür wird der von Android bereitgestellte *Uri.Builder* verwendet. Dieser kann von einer Base-URL ausgehend Parameter automatisch korrekt formatieren. Es werden dafür lediglich der Name des Parameters und der zugehörige Wert als String benötigt. In diesem Fall werden automatisch der erste Wegpunkt des übergebenen ArrayAdapter als „origin“ und der letzte als „destination“ verwendet. Für die Umwandlung von *LatLng* zu String steht die *positionToString(LatLng)* Methode zur Verfügung. Sie wandelt Längen- und Breitengrad von double Werten in Strings um und kombiniert diese, getrennt durch ein „,“ zu einem einzelnen String. Die *waypointsToString(ArrayAdapter<Waypoint>)* Methode kombiniert mit-

hilfe der *positionToString(LatLng)* Methode die enthaltenen Wegpunkte zu einem String, indem sie diese mit dem Trennzeichen „|“ kombiniert. Da es sich beim 1. und letzten Wegpunkt um den Start- und Endpunkt handelt, werden diese nicht berücksichtigt. Der resultierende String wird im *Uri.Builder* für den Parameter „waypoints“ eingetragen. Zuletzt wird durch „sensor=true“ signalisiert, dass das Gerät über einen Standortsensor verfügt. Mit der entstandenen Uri kann nun ein URL Objekt erstellt werden, welches über die *openConnection()* Methode eine HTTP Verbindung öffnet. An diese Verbindung wird nun eine „GET“ Anfrage geschickt. Als Antwort erhalten wir das Ergebnis der geforderten Routenberechnung formatiert als JSON-String. Dieser kann mithilfe der *parseJSONString(String)* Methode geparst werden. Als Ergebnis erhalten wir ein Route Objekt mit dem berechneten Streckenverlauf. Die Methode legt zunächst ein JSON-Object mit dem empfangenen String an, welches Teil des Android SDK ist. Dieses Objekt stellt unter anderem zwei Methoden zur Verfügung, die entsprechend der Objekthierarchie Zugriff auf unterliegende JSON Objekte bzw. Arrays bietet. Mit der *getJSONArray(String)* Methode erhalten wir das JSONArray, welches die einzelnen Routen beinhaltet. Hier reicht es aus die erste enthaltene Route zu verwenden. Die Route enthält in „overview_polyline.points“ eine als String codierte Linie, welche den gesamten Routenverlauf umfasst. Diese Linien werden mithilfe des Polyline Algorithmus codiert, welcher es ermöglicht diese kompakt zu speichern. Der Algorithmus ist in der Google Maps Dokumentation[2] genauer beschrieben; Die Implementation in der *LocationUtils* liefert zudem eine genaue Beschreibung der Funktion. Bei kurzen Strecken entspricht diese Übersichtsstrecke dem tatsächlichen Streckenverlauf, bei längeren Strecken ist dies jedoch nicht garantiert. Aus diesem Grund decodieren wir stattdessen die Strecken der einzelnen Routenanweisungen. Die Anweisungen befinden sich als Array „steps“ in den Streckenabschnitten, die als Array „legs“ vorhanden sind. Um also den gesamten Routenverlauf zu erhalten wird im JSONObject welches die Route repräsentiert jedes „legs[i].steps[j].polyline.points“ decodiert und einer Liste mit *LatLng* Objekten hinzugefügt. Diese wird anschließend dem *PolylineOptions* Objekt des Route Objekts hinzugefügt. Um die Gesamtlänge der Strecke zu erhalten wird dabei auch die im „legs[i].steps[j].distance.value“ enthaltene Strecke in Metern aufaddiert. Das entstehende Route Objekt wird der *RouteEditorActivity* anschließend als Ergebnis übermittelt.

Routen als Vorlage nutzen und das Benennen von Wegpunkten

von Bernd

Man kann bestehende Routen benutzen um aus diesen neue Routen zu erstellen. Dazu muss man im *ManageRoutesFragment* eine Route aus der Liste auswählen und einen Longclick darauf machen. Es erscheint ein AlertDialog dessen Überschrift den Namen der Route darstellt. Man hat nun als Auswahl die Route zu löschen oder diese als Vorlage zu nutzen. Wählt man letzteres, so wird die Activity *AddRouteDraftDialog* gestartet. Des weiteren werden als Extras dem Intent die Waypoints der ausgewählten Route beigefügt. Ein Waypoint einer Route ist eine Klasse auf dem Server mit dem Namen *ServerWaypoint* . Ein *ServerWaypoint* hat die Attribute für den Titel eines Waypoints und dessen Latitude und Longitude. Eine Route hat eine Liste von *ServerWaypoints* und diese werden der Route hinzugefügt, wenn die Route erstellt wird, das heißt wenn der AlertDialog zum Speichern einer Route aufgerufen wird. Dort wird dann der *ArrayAdapter* durchlaufen und für jeden Waypoint wird ein *ServerWaypoint* erstellt. Dem ServerWaypoint werden dann der Titel, die Longitude und die Latitude gesetzt und er wird der Liste von ServerWaypoints hinzugefügt. Der Dialog ist eine Activity, in der man ein Feld für den Namen der Route zusammen mit zwei Buttons zum Abbrechen und Bestätigen der Aktion. Gibt man einen Namen für die neue Route ein und bestätigt dies, so wird in der Activity ein Intent gestartet und ihm in den Extras die Waypoints der Route übergeben. Dann wird die *RouteEditorActivity* mit diesem Intent gestartet. In der *onCreate(...)* Methode von

[2] https://developers.google.com/maps/documentation/utilities/polylinealgorithm

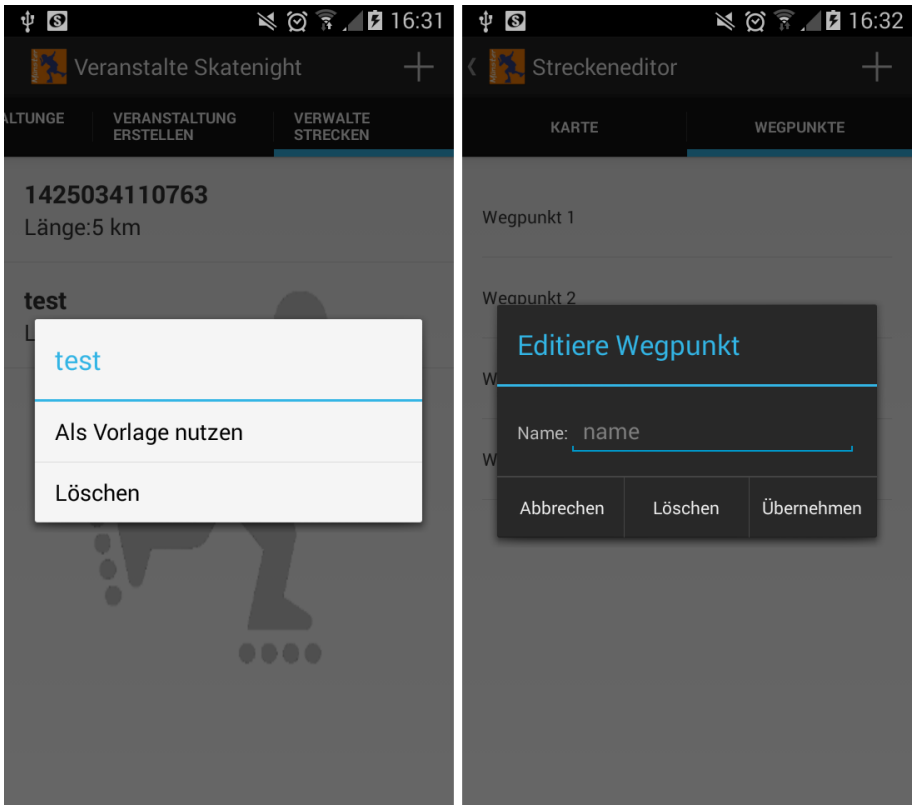


Abbildung 8: Erstellen einer Route aus einer Vorlage und das Umbenennen von Wegpunkten

der *RouteEditorActivitiy* wird dann abgefragt, ob der Intent Extras besitzt und auch, ob dies Waypoints sind. Ist dies der Fall, so werden die übergebenen Waypoints aus den Extras in ein Attribut gespeichert. Für jeden Waypoint wird dann ein neuer Waypoint erstellt und diesem dann die Latitude, die Longitude und der Titel gesetzt. Der neue Waypoint wird dann auch noch dem WaypointAdapter hinzugefügt. Der WaypointAdapter erweitert den ArrayAdapter mit dem generischen Typ Waypoint und überschreibt die *getView(...)* Methode, sodass für jeden Waypoint eine custom xml Datei angezeigt wird. In dieser Datei bedindet sich eine TextView zum Anzeigen des Namens des Wayoints und dieser wird für jeden Waypoint gesetzt.

6.2.4 Events

von Martin

Dieser Abschnitt beschäftigt sich mit der Realisierung der Veranstaltungen in der Veranstalter-App. Veranstaltungen können nur von eingetragenen Veranstaltern in der Veranstalter-App erstellt und verwaltet werden und können nach der Erstellung von den Benutzern in der User-App und von allen Veranstaltern in der Veranstalter-App betrachtet werden. Dazu werden die erstellten Veranstaltungen persistent auf dem Server abgespeichert. Zu Beginn des Projekts hatte eine Veranstaltung noch eine feste Anzahl an Informationen, d.h. die Model-Klasse „Event“, die eine Veranstaltung beschreibt, hatte Attribute für den Titel, für die Gebühr, für die Beschreibung usw. Durch Einführung der Features „Veröffentlichen von Informationen 2.0“ und „Anzeigen von Informationen 2.0“ musste die Umsetzung überarbeitet werden, da die Anzahl der Informationen zu einer Veranstaltung variabel wurde. Die Realisierung erfolgte nun über dynamische Felder, die eine abstrakte Darstellung einer Information einer Veranstaltung ist. Diese Umsetzung wird im Folgenden behandelt.

Model-Klasse Event

Die Klasse Event in unserem Projekt stellt eine Veranstaltung dar. Ein Objekt dieser Klasse beinhaltet alle Informationen einer Veranstaltung.

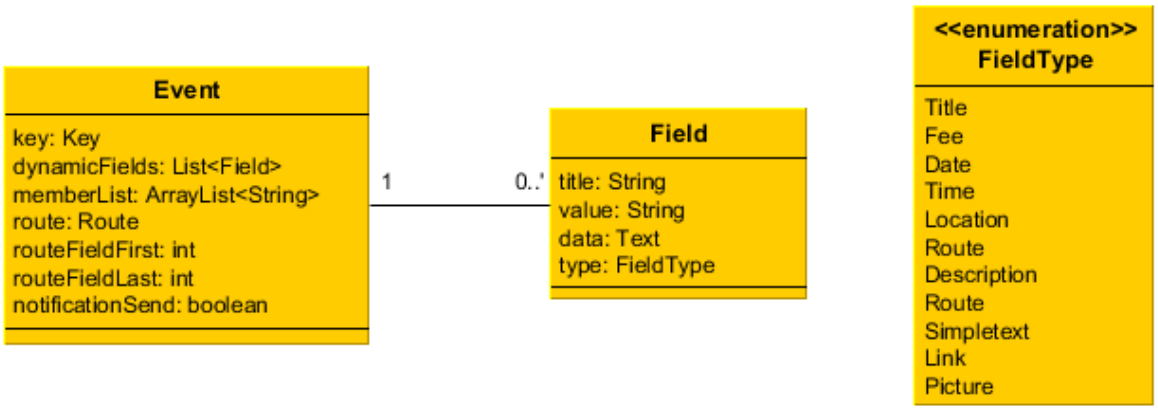


Abbildung 9: Die verwendeten Klassen zur Repräsentation von Veranstaltungen

Zu Beginn werde ich auf die dynamischen Felder und auf die Arten der dynamischen Felder eingehen, die durch die Klassen *Field* und *FieldType* realisiert wurden.

Klasse Field:

Ein Objekt der Klasse *Field* stellt ein dynamisches Feld dar. Ein dynamisches Feld wird genutzt um eine Veranstaltung zu beschreiben. Jedes dynamische Feld stellt genau eine Information zu einer Veranstaltung dar und besteht aus:

- title: Bezeichnung der Information (z.B. „Titel“, „Gebühr“)
- value: Wert der Informationen in Form einer Zeichenkette
- data: Feld, das nur für die Speicherung einer Routeninformation genutzt wird, leer bei jeder anderen Information
- type: Art des Feldes, ein Enum-Objekt des Enums *FieldTypes*

Enum FieldType:

Es existieren zehn verschiedene Arten von dynamischen Feldern, die durch das Enum *FieldType* identifiziert werden können. Folgende Arten von dynamischen Feldern existieren:

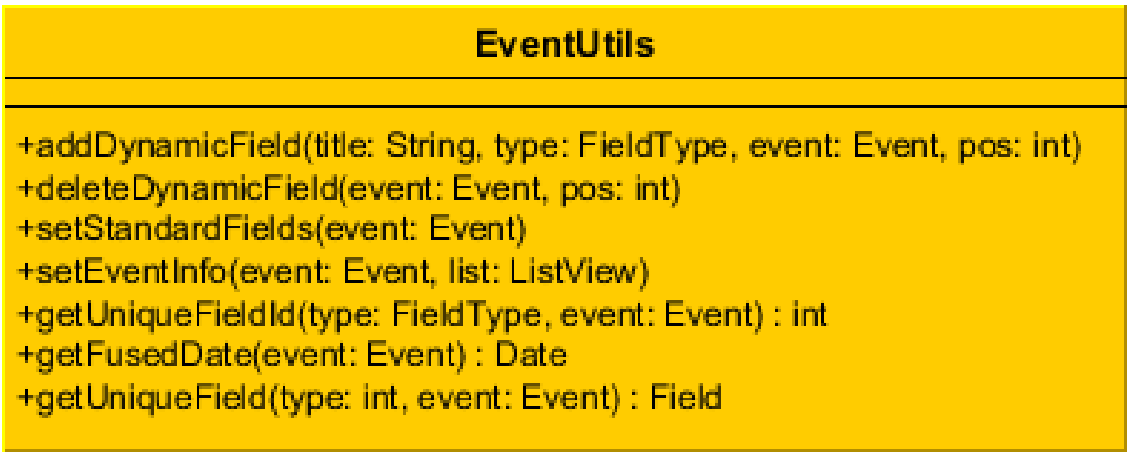
- Title: Titel
- Fee: Gebühr
- Date: Datum des Startzeitpunkts
- Time: Uhrzeit des Startzeitpunkts
- Location: Startpunkt
- Description: Beschreibung
- Route: Route, die in der Veranstaltung abgefahren wird
- Simpletext: Ergänzende Text-Information
- Link: Verweis auf Webseite

- Picture: Bild

Klasse Event:
Ein Event-Objekt beinhaltet alle Information zu einer Veranstaltung. Folgende Informationen ergeben sich:

- key: Eindeutige Nummer, die eine Veranstaltung identifiziert
- dynamicFields: Liste von dynamischen Feldern
- memberList: Liste von E-Mail Adressen der Teilnehmer
- routeFieldFirst: erster Wegpunkt des Feldes(siehe Abschnitt 5.7)
- routeFieldLast: letzter Wegpunkt des Feldes(siehe Abschnitt 5.7)
- notificationSend: identifiziert, ob die GCM-Nachricht für das Starten einer Veranstaltung gesendet wurde

Wird ein neues Event auf dem Server erstellt, wird automatisch eine Key-ID generiert sodass wir uns nicht um die Generierung einer ID kümmern müssen. Da wir eine variable Anzahl an dynamischen Feldern abspeichern wollen, haben wir uns dazu entschieden, die dynamischen Felder in einer Liste abzuspeichern. Jedes dynamische Feld hat einen eigenen Typen, wobei sich die Typen „Simpletext“, „Link“ und „Picture“ von den restlichen Typen unterscheiden, da ein Objekt mit diesen Typen mehrmals in der Liste auftreten können. Die restlichen Typen sind einzigartig und treten nur ein einziges Mal in der Liste auf. Die beiden Felder *routeFieldFirst* und *routeFieldLast* werden für die Feldberechnung genutzt, um den Start und das Ende des Feldes zu speichern (siehe Abschnitt 5.7). Das letzte Attribut *notificationSend* speichert ob jemals schon eine Notification, die den Start der Veranstaltung ankündigt, gesendet wurde.
EventUtils:
Um den Zugriff und die Verwaltung der dynamischen Felder einer Veranstaltung zu vereinfachen haben wir uns dazu entschieden eine Singleton-Klasse *EventUtils* zu erstellen.



Folgende Funktionen bietet die Klasse *EventUtils* an:

- addDynamicField: Fügt ein dynamisches Feld mit dem übergebenen Titel und Typ dem Event an der Position „pos“ hinzu.

- deleteDynamicField: Löscht ein dynamisches Feld an der Position „pos“ aus dem Event.
- setStandardFields: Setzt die dynamischen Felder die zu Beginn der Erstellung initialisiert werden sollen(Titel, Gebühr, Startdatum, Startuhrzeit, Ort, Beschreibung, Route).
- setEventInfo: Setzt die dynamischen Felder der übergebenen ListView in das übergebene Event-Objekt.
- getUniqueFieldId: Gibt die Position des eindeutigen dynamischen Feldes mit dem Field-Type „type“ in der Liste der dynamischen Felder des übergebenen Events zurück.
- getFusedDate: Bestimmt aus Startdatum und Startuhrzeit das Datum des Starts mit der Uhrzeit.
- getUniqueField: Gibt das einzigartige dynamisches Feld mit dem *FieldType* „type“ zurück.

Erstellen einer Veranstaltung

Das Erstellen einer neuen Veranstaltung ist nur in der Veranstalter-App durch einen eingetragenen Veranstalter möglich. Die Klasse, die die Erstellung verwaltet, ist ein Fragment, das *AnnounceInformationFragment* , welches man über das Navigieren des *ViewPagers* erreichen kann. Das angezeigte Fragment ist auf Abbildung 10 dargestellt.

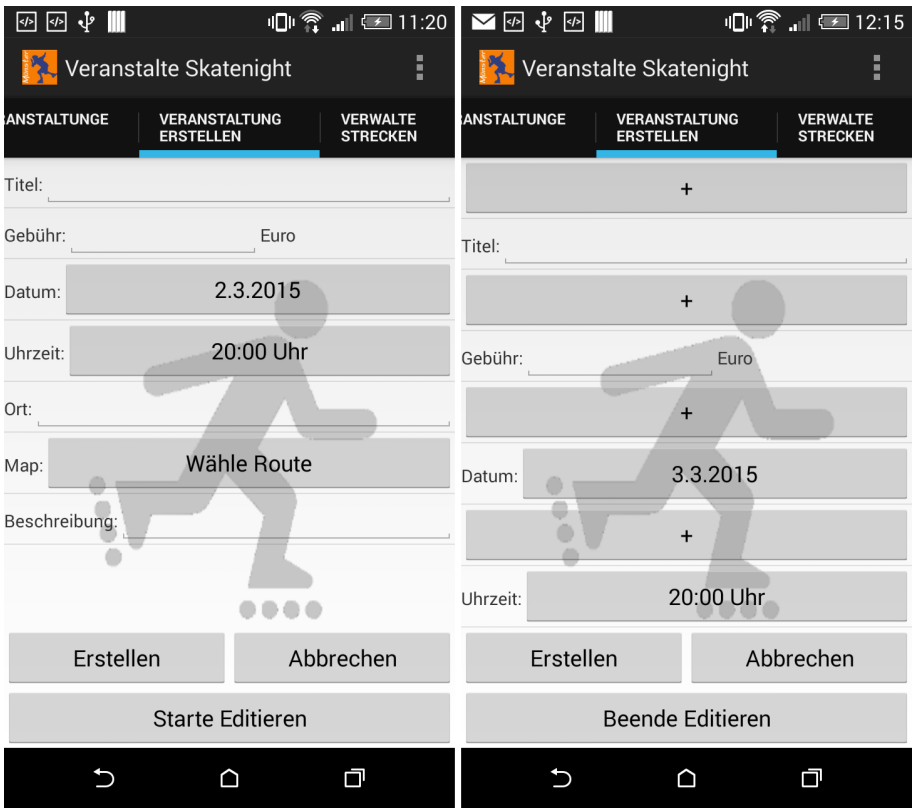


Abbildung 10: Das Fragment zur Erstellung von Veranstaltungen(links ohne, rechts mit aktiviertem Editiermodus

Vor der Einführung der 2.0 Features war die grafische Umsetzung der Erstellung einfach, da wir eine feste Anzahl von grafischen Elementen hatten, die angezeigt werden mussten um die eingegebenen Informationen vom Veranstalter zu registrieren. Nach der Einführung musste es auch möglich sein grafische Elemente zu ergänzen, da weitere Angaben, wie z.B. zusätzliche Textfelder möglich waren. Wir haben uns dazu entschieden, dies mit einer ListView und einem

Adapter, dem *AnnounceCursorAdapter* , zu realisieren, da ein Adapter relativ einfach um eine Datenmenge erweitern kann. Der Adapter hat als Datenmenge eine Liste von dynamischen Feldern, da diese genau eine Information zu einer Veranstaltung darstellen, und erstellt zu jedem dynamischen Feld ein grafisches View-Element und fügt es in die ListView ein. Da verschiedene Typen von dynamischen Feldern existieren, haben wir uns vor der codetechnischen Umsetzung erst verschiedene Layouts mit Hilfe von xml-Dateien erstellt, sodass der Adapter Typ-abhängig die Layouts für die Elemente der ListView setzen kann. Damit die Felder jetzt in der ListView angezeigt werden, wird in der *getCount* -Methode die Anzahl der angezeigten Elemente auf die Größe der Liste der dynamischen Felder gesetzt. Anschließend wird für jedes dynamische Feld die *getView* -Methode aufgerufen, die den Typ des dynamischen Feldes an der aktuellen Position in der Liste abfragt, wurde der richtige Typ gefunden, wird die passende Layout xml-Datei gesetzt. Die Realisierung der Erweiterung von Informationen haben wir mit einem Editiermodus gelöst. Wird der Editiermodus über den Button „Starte Editieren“ gestartet erscheint über und unter jedem Element ein „+“-Button. Dazu wird in der *getCount* -Methode im Adapter die Anzahl verdoppelt und in der *getView* -Methode wird bei jedem zweiten Element dieser Button gezeichnet. Wird dieser gedrückt, so kann man Auswählen welches dynamische Feld an diese Stelle ergänzt werden soll. Nach der Auswahl wird die Bezeichnung des dynamischen Feldes festgelegt. Anschließend wird im Adapter die Liste mit den dynamischen Feldern um das neue Feld an der bestimmten Position ergänzt und der Adapter zeichnet die Felder neu.
Wird nun also das Fragment erstellt wird ein neues Event erstellt und die dynamischen Standard-Felder werden mit Hilfe der *EventUtils* in dem Event gespeichert. Dann wird der Adapter mit den dynamischen Feldern des Events initialisiert und dieser wird dann in der ListView gesetzt. Am Ende zeichnet der Adapter wie zuvor beschrieben die dynamischen Felder, indem er die *getView* -Methode iteriert bis die Anzahl von der *getCount* -Methode erreicht wird. Hat man nun alle notwendigen Informationen angegeben wird mit Hilfe der *setEventInfo* -Methode aus den *EventUtils* die Informationen des Adapters an das Event übergeben. Danach wird der *CreateEventTask* aufgerufen, der im Hintergrund-Thread auf dem Server die neue Veranstaltung erstellt. Am Ende wird das Fragment in den Anfangszustand gesetzt und die Liste der Veranstaltungen im *ShowEventsFragment* wird aktualisiert.

Veranstaltungen editieren und löschen

Um eine Veranstaltungen zu editieren, muss man die zu bearbeitende Veranstaltung lange berühren. Dann öffnet sich ein Menü, in dem man auswählen kann, ob man die Veranstaltung editieren oder löschen möchte. Wählt man „Löschen“ so wird der *DeleteEventTask* aufgerufen, der die Veranstaltung im Hintergrund-Thread vom Server löscht und anschließend aus der ListView entfernt. Wählt man „Editieren“ so wird ein Intent mit der *EditEventActivity* erstellt und dem Intent wird die ID der Veranstaltung übergeben. Anschließend wird die Activity mit dem Intent gestartet. In der Activity wird zunächst in der *onCreate* -Methode die Veranstaltung mit der übergebenen ID vom Server mit Hilfe der *getEventTask* Methode abgerufen. Diese Activity sieht genau so aus wie das *AnnounceInformationFragment* und hat die gleiche Funktionsweise. Auch hier wird der *AnnounceCursorAdapter* verwendet um die Informationen der Veranstaltung anzuzeigen. Jedoch werden hier in der *EditEventActivity* nicht die Standard-Felder in den *AnnounceCursorAdapter* geladen, sondern die dynamischen Felder der abgerufenen Veranstaltung. Nun kann man wie auch schon beim Erstellen einer Veranstaltung weitere Felder über den Editiermodus ergänzen. Ist man fertig mit dem Bearbeiten der Veranstaltung und berührt den Button „Speichern“ wird mit Hilfe der *EventUtils* die Daten der dynamischen Felder in das Event-Objekt gespeichert, das nach dem Abrufen der Veranstaltung vom Server erstellt wurde. Dann wird ein *EditEventTask* mit dem Event-Objekt gestartet, der die neuen Informationen der Veranstaltung im Hintergrund-Thread in das Event-Objekt auf dem Server schreibt.

Veranstaltungen anzeigen

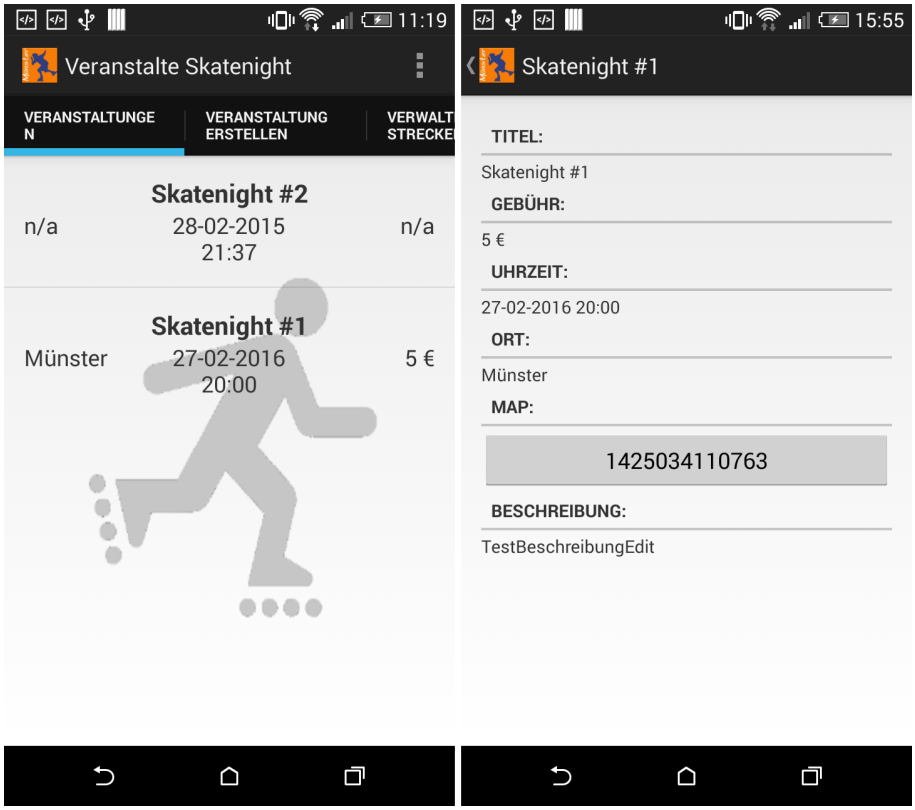


Abbildung 11: Anzeige von Veranstaltungen in der Veranstalter-App

Die Veranstaltungen werden in einer ListView angezeigt, die mit einem Adapter, dem *EventsCursorAdapter* , mit Inhalt gefüllt wird. In der Veranstalter-App haben wir ein Fragment, das *ShowEventsFragment* , das die ListView anzeigt. Dieses Fragment wird geladen wenn man den ersten Tab in der Tab-Leiste auswählt und wird beim Start der App automatisch ausgewählt. Die Layout xml-Datei eines Elements der ListView enthält die wichtigsten Informationen der Veranstaltung: den Titel, der Ort, der Startzeitpunkt und die Gebühr. Wird nun das Fragment angezeigt wird der *QueryEventsTask* ausgeführt, der im Hintergrund-Thread die Liste der Veranstaltungen vom Server abruft. Hat der Thread seine Arbeit getan, wird ein Adapter-Objekt erstellt, die Liste der Veranstaltungen wird diesem übergeben und die ListView wird mit dem Adapter gesetzt. Anschließend wird über die *getView* -Methode iteriert, die die passenden View-Elemente zu den Veranstaltungen erstellt und diese werden dann in der ListView angezeigt. Zusätzlich wird ein *OnItemClickListener* auf die ListView gesetzt, der prüft ob ein Element der ListView gedrückt wird. Wird ein Element gedrückt wird eine neue Activity, die *ShowInformationActivity* , gestartet. Dem Intent wird zusätzlich noch die Event-Id übergeben, da die Activity alle Informationen der Veranstaltung anzeigen soll. Da die Anzahl variabel ist, haben wir uns wieder dazu entschieden, diese Angaben in einer ListView mit einem Adapter anzuzeigen. Nach Start der Activity wird zunächst über den Intent die Event-Id ermittelt. Mit der Id wird der *GetEventTask* ausgeführt, der das Event-Objekt zu der Veranstaltung vom Server abruft. Abschließend wird ein *EventsCursorAdapter* -Objekt mit den dynamischen Feldern der Veranstaltung erstellt und diese ListView wird mit dem Adapter gesetzt. Der *EventsCursorAdapter* funktioniert ähnlich wie der *AnnounceCursorAdapter* . Wir haben wieder verschiedene Layout xml-Dateien für die verschiedenen Typen, wobei eine Layout Datei jeweils immer eine Header-TextView mit der Bezeichnung der Information hat. Dann wird in der *getView* -Methode abgefragt um welchen Typ es sich bei dem dynamischen Feld handelt und es wird die entspre-

chende Layout Datei geladen und die Informationen werden gesetzt. Diese Methode wird für jedes dynamische Feld in der Liste aufgerufen. Eine spezielles dynamisches Feld ist das Feld für die Route, da in dem Fall eine Layout-Datei geladen wird, die einen Button enthält, der beim Auslösen eine neue Activity, die *ShowRouteActivity* , startet, die die Route der Veranstaltung anzeigt.

6.3 Beschreibung User-App

6.3.1 Gerüst

von Martin

Abbildung 12: Der ausgeklappte Navigation Drawer

Die User-App besteht aus einer Activity, der *ShowEventsActivity* , die alle Veranstaltungen auf dem Server anzeigt und aus einer FragmentActivity, der *UsergroupActivitiy* , die alle Gruppen auf dem Server und alle Gruppen denen man beigetreten ist in zwei Tabs anzeigt. Um zwischen den beiden Activities zu navigieren haben wir uns entschieden einen Navigation Drawer zu verwenden. Ein Navigation Drawer ist ein Fragment, das sich von der linken Kante des Bildschirms in das Hauptfenster schiebt und die wichtigsten Navigationsmöglichkeiten der App anzeigt. Damit der Drawer angezeigt wird, muss man entweder vom linken äußersten Rand zur Mitte swipen oder das App-Icon in der ActionBar berühren. Klickt man auf eines der Felder im Navigation Drawer startet die dazugehörige Activity oder sie wird in den Vordergrund geschoben wenn bereits eine Instanz der Activity existiert. Um dies zu realisieren, haben wir eine *BaseActivity* und *BaseFragmentActivity* erstellt, die als Oberklasse jeder Activity bzw. FragmentActivity dient, die einen Navigation Drawer nutzt. In diesen Oberklassen wird die *setContentView(int layoutResId)* Methode überschrieben. Diese Methode wird von jeder Activity in der *onCreate* -Methode

aufgerufen um das Layout zu setzen. In der überschriebenen *setContentView* -Methode wird ein DrawerLayout geladen, in dem eine ListView existiert, die bei Start nicht sichtbar ist und außerhalb des Bildschirms liegt und ein FrameLayout, das den ganzen Platz des Bildschirms nutzt. Das Layout das von der Unterklasse mit *setContentView* gesetzt wird, wird in dieses FrameLayout geladen sodass am Ende dann der Inhalt der Activity angezeigt wird. In der Base- bzw. BaseFragmentActivity wird in der *setContentView* -Methode auch noch die ListView initialisiert, d.h. es werden die Menü-Titel und Menü-Icons der ListView hinzugefügt und ein ItemClickListener gesetzt, der prüft, ob ein Item berührt wurde, sodass die dazugehörige Activity geladen werden kann.

6.3.2 Eventanzeige

von Martin

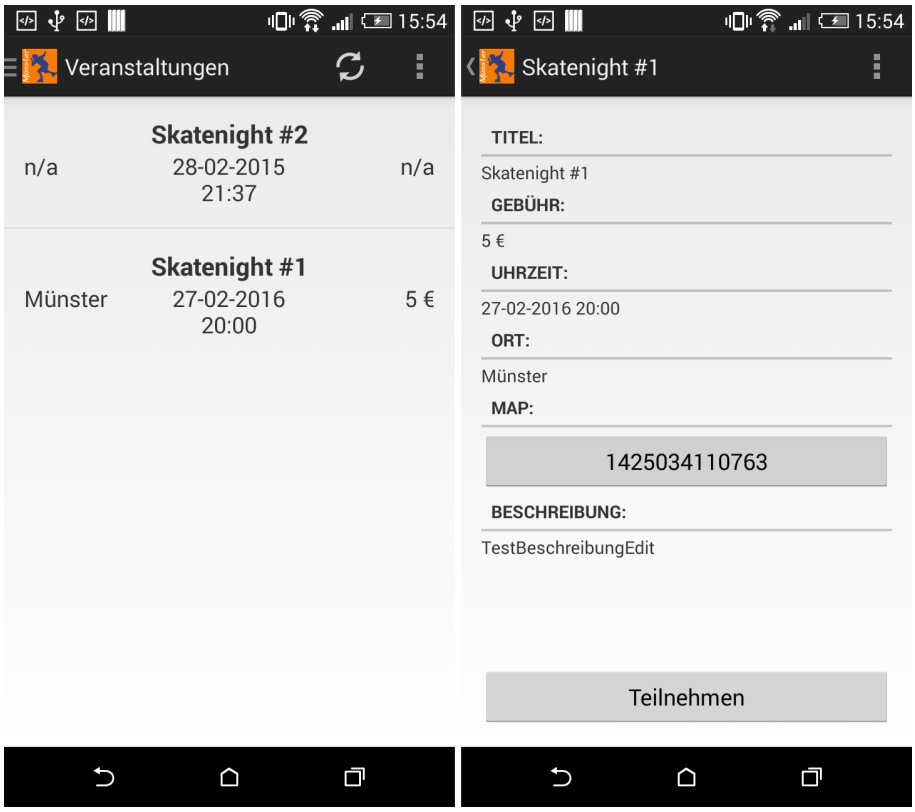


Abbildung 13: Die Event-Anzeige der User-App

Veranstaltungen werden in der User-App in der *ShowEventsActivity* angezeigt. Diese wird bei Start der User-App angezeigt. Zusätzlich kann man zu ihr über den Navigation Drawer navigieren. Die Anzeige der Veranstaltungen funktioniert genau so wie in der Veranstalter-App(Kapitel ...), jedoch haben wir eine Activity und kein Fragment in der die ListView angezeigt wird. Wird also die Activity gestartet, wird in der *onCreate* -Methode die Liste der Veranstaltungen vom Server mit Hilfe des *QueryEventsTask* abgerufen und anschließend wird die ListView mit den Informationen gefüllt. Auch hier haben wir eine *ShowInformationActivity* , die gestartet wird, wenn man ein Element der ListView berührt. Die *ShowInformationActivity* unterscheidet sich aber in der User-App von der Veranstalter-App, da man in der User-App auch noch zusätzlich in der Activity entscheiden kann ob man an der Veranstaltung teilnehmen möchte. Dazu existiert ein Button unter der ListView, der „Teilnehmen“-Button, der im Fall der Teilnahme „Verlassen“

heißt. Wird der „Teilnehmen“-Button ausgelöst, wird der Benutzer mit seiner Email-Adresse in der Veranstaltung eingetragen. Außerdem wird in dieser Activity auch die Lokale Auswertung umgesetzt. Wurde an der Veranstaltung teilgenommen und der Veranstaltungszeitpunkt wurde erreicht wird hier die lokale Auswertung geladen. Dies wird im Abschnitt der lokalen Auswertung (6.3.4) näher behandelt.

6.3.3 Nutzergruppen

von Bernd

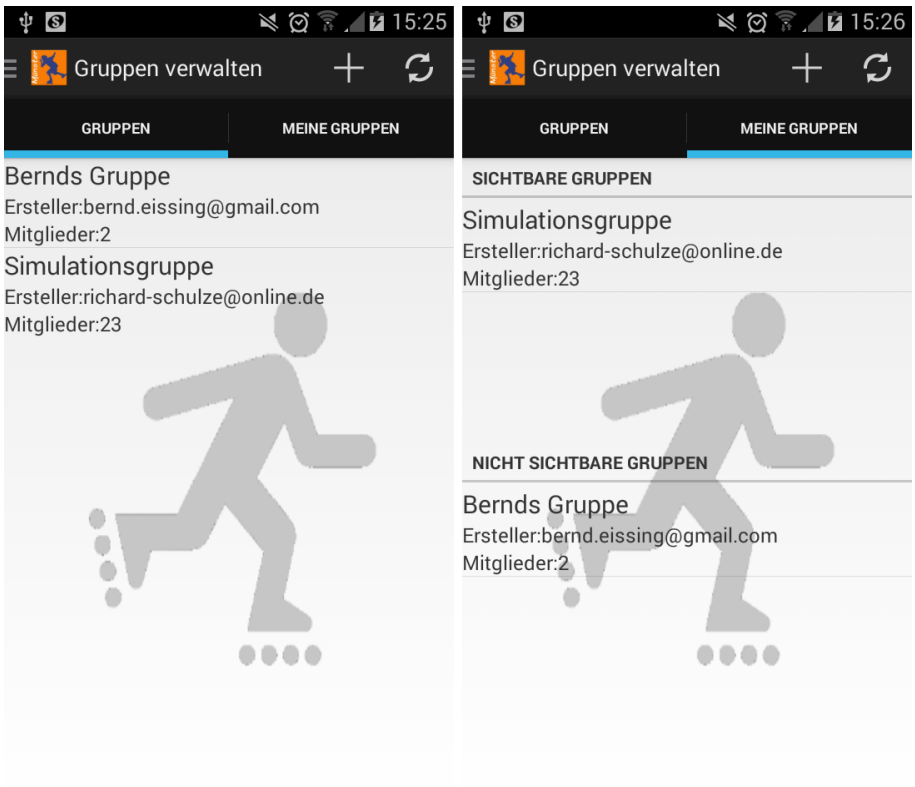


Abbildung 14: Verwaltung von Nutzergruppen

In der Userapp kann ein Benutzer eigene Nutzergruppen verwalten, anderen Nutzergruppen beitreten oder diese verlassen. Um in der Userapp zu den Nutzergruppen zu gelangen, muss der NavigationDrawer bedient werden und dort kann man dann Nutzergruppen auswählen. Die Nutzergruppen werden über eine Activity(*UsergroupActivity*) angezeigt, welche zwei Fragmente(das *AllUsergroupsFragment* und das *MyUsergroupsFragment*) hält. Das *AllUsergoupsFragment* hält eine Liste mit allen auf dem Server vorhandenen Nutzergruppen. Wenn dieses Fragment aufgerufen wird, dann wird der *QueryUserGroupsTask* aufgerufen, welcher in der *doInBackground(...)* Methode alle Nutzergruppen vom Server abruft und diese dann in der *onPostExecute(...)* Methode der ListView des Fragments hinzufügt. Der Adapter für die ListView ist der *UsergroupAdapter* . Dieser setzt für jede Nutzergruppe, die der ListView hinzugefügt wurden, deren Namen, Ersteller und Anzahl an Mitgliedern im Item(*list_view_item_usergroup.xml*). Der Adapter wird in der *onResume(...)* Methode des *AllUsergroupsFragment* gesetzt.

Neue Nutzergruppe erstellen

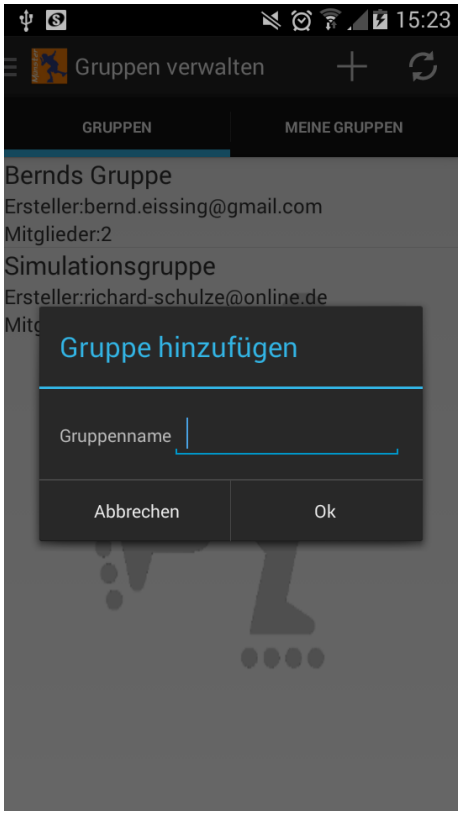


Abbildung 15: GUI zum Hinzufügen von Benutzergruppen

Man hat im *AllUsergroupsFragment* die Möglichkeit neue Nutzergruppen zu erstellen, indem man im Optionsmenü den Plus Button betätigt. Dies führt zum Aufruf von der *onOptionsItemSelected(...)* Methode in der *UsergroupActivity* . Hier wird die *AddUsergroupActivity* aufgerufen, was lediglich einen Dialog darstellen soll. Die Activity hat ein Feld für den Namen der neuen Nutzergruppe, einen Button zum Abbrechen und einen Button zum Bestätigen der Aktion. Wird ein Name einer Nutzergruppe eingegeben, der nicht leer ist so wird die *apply(...)* Methode in der *AddUsergroupActivity* aufgerufen. Diese Methode führt den *QueryUserGroupsTask* aus, welcher eine Instanz vom *AllUsergropusFragment* erwartet. In dem hier übergebenen Fragment wird die *setUserGroupsToListView(...)* Methode neu definiert da diese von dem *QueryUserGroupsTask* in der *onPostExecute(...)* Methode aufgerufen wird. Es werden alle Nutzergruppen nach Namen durchsucht und überprüft, ob der angegebene Name bereits vergeben ist. Ist dies nicht der Fall wird der *AddUserGroupTask* aufgerufen, welcher eine neue Nutzergruppe in der *doInBackground(...)* Methode erstellt und dann die *refresh(...)* Methode in der *UsergroupActivity* aufruft um die Liste mit den Nutzergruppen zu aktualisieren.

Listen aktualisieren

Die *referesh(...)* Methode kann auch manuell aufgerufen werden, dazu muss man im Optionsmenü den Refresh Button betätigen. Die *refresh(...)* Methode in der *UsergroupActivity* ruft die *refresh(...)* Methode in dem *AllUsergroupsFragment* und in dem *MyUsergroupsFragment* auf. Dort wird der *QueryUserGroupsTask* im *AllUsergroupsFragment* und der *QueryMyUserGroupsTask* im *MyUsergroupsFragment* aufgerufen um die Liste von Nutzergruppen zu aktualisieren.

Nutzergruppen beitreten und verlassen

Mit einem einfachen Klick auf eine Nutzergruppe in der Liste kann man dieser beitreten oder diese verlassen, solange man nicht der Ersteller ist. Beim Verlassen und beim Beitreten wird ein AlertDialog angezeigt dessen Überschrift den Namen der Gruppe darstellt zusammen mit der Frage, ob man die ausgewählte Nutzergruppe verlassen möchte und zwei Buttons zum Bestätigen und zum Abbrechen der Vorgangs. Verlässt man eine Nutzergruppe so führt dies zu zum Aufruft der *createDialogLeave(...)* Methode in der *GroupUtils* Klasse. Beim Bestätigen wird der *LeaveUserGroupTask* mit dem Namen der ausgewählten Nutzergruppe aufgerufen. Dieser Task ruft auf dem Server die *leaveUserGroup(...)* Methode in der *doInBackground(...)* Methode und anschließend in der *onPostExecute(...)* Methode die *refresh(...)* Methode der *UsergroupActivity* auf, um die Änderung sichtbar zu machen. Beim Verlassen wird die *createDialogJoin(...)* Methode in *GroupUtils* aufgerufen. Diese ruft bei Bestätigung den *JoinUserGroupTask* auf welcher die *joinUserGroup(...)* Methode auf dem Server aufruft und abschließend die *refresh(...)* Methode ausführt, um die Änderung sichtbar zu machen. Ist man der Ersteller der augewählten Nutzergruppe so wird die *createDialogOwner(...)* Methode aufgerufen welche einen AlertDialog erstellt der darüber informiert, dass man der Ersteller der Nutzergruppe ist und man diese weder verlassen noch dieser beitreten kann.

Nutzergruppen auf der Karte sichtbar machen und löschen

Abbildung 16: Das Auswahlmenü auf einer Gruppe

Das *MyUsergroupsFragment* enthält zwei ListViews, welche sich jeweils die Hälfte des Bildschirms teilen. Die erste ListView kann maximal fünf Nutzergruppen halten und dient dazu Nutzergruppen auf der Karte sichtbar zu machen. Die zweite ListView zeigt alle Nutergruppen an bei denen man entweder Mitglied oder Ersteller ist. Beide ListViews benztzen den *UsergroupAdapter* als Adapter, sie unterscheiden sich jedoch in ihren Namen. Der eine ist für die

sichtbaren und der andere für die nicht sichtbaren Nutzergruppen. Mit einem Long Click auf eine Nutzergruppe in entweder der Liste der sichtbaren oder der Liste aller Nutzergruppen führt dies zu einem Aufruft der *createSelectionsMenu(...)* Methode im *MyUsergroupsFragment* . Diese Methode überprüft zunächst aus welchem Adapter der Aufruft kommt und ob der Aufrufer auch gleichzeitig der Ersteller der Nutzergruppe ist. Dann wird ein AlertDialog angezeigt mit dem Namen der Nutzergruppe als Überschrift und zwei weiteren Optionen, falls man nicht der Ersteller dieser ist. Die erste Option ist es die Nutzergruppe auf der Map sichtbar oder unsichtbar zu machen. Wählt man diese Option, so wird im *PrefManager* die *setGroupVisibility(...)* Methode aufgerufen, welche die Nutzergruppe sichtbar oder unsichtbar macht. Diese Methode speichert in den SharedPreferences des Smartphones die Sichtbarkeit jeder Gruppe. Die SharedPreferences sind ein lokaler Speicher der zu Key-Einträgen Werte lokal auf dem Smartphone speichert. Die zweite Option ist es die Gruppe zu verlassen; das Auswählen dieser Funktion wurde bereits in Abschnitt 6.3.3 erklärt. Die dritte Option ist es die Nutzergruppe zu löschen, welche nur angezeigt wird wenn man der Ersteller dieser ist. Wenn diese Option ausgewählt wird, wird die *createDialogDelete(...)* Methode in *GroupUtils* aufgerufen. Hier wird ein neuer AlertDialog erstellt, dessen Überschrift der Name der Nutzergruppe ist und als Text die Frage ob man die Nutzergruppe löschen möchte zusammen mit Buttons zum Bestätigen und Abbrechen der Aktion enthält. Wird das Löschen bestätigt, so wird der *DeleteUserGroupTask* aufgerufen, welcher in der *doInBackground(...)* Methode die Nutzergruppe vom Server löscht und in der *onPostExecute(...)* Methode die Änderungen aktualisiert.

6.3.4 Lokale Auswertung

von Pascal und Daniel

Die Lokale-Auswertung soll dem Benutzer ermöglichen statistische Daten zu erfassen, visualisieren und auszuwerten. Zu diesen Daten zählen:

- Der aktuelle Fortschritt auf der Strecke
- Die Zeit die seit dem Start des Events vergangen ist
- Die bereits zurück gelegte Distanz, sowie die Gesamtdistanz
- Die aktuelle, maximale und durchschnittliche Geschwindigkeit
- Die aktuelle Höhe und die zurückgelegten Höhenmeter
- Das Geschwindigkeitsprofil

Diese Daten sollen dem Benutzer auf intuitive Weise präsentiert werden, sodass dieser sie einfach während des Events ablesen kann. Eine besondere Herausforderung ist zu garantieren, dass die Daten unabhängig vom Status der App erfasst werden. So sollen die Daten auch aufgezeichnet werden wenn die App nicht gestartet wurde oder vorzeitig beendet wird. Desweiteren sollten sie möglichst kompakt und persistent gespeichert werden um sicherzustellen, dass die Daten auch nach Abschluss des Events analysiert werden können. Weiterhin soll die Auswertung unabhängig von der globalen Auswertung (siehe Kapitel 5.7) stattfinden, ohne die Notwendigkeit Positionsdaten an den Server zu übermitteln.

LocationTransmitterService

Der *LocationTransmitterService* ist ein Vordergrundservice der für die Erhebung und Analyse der Positionsdaten verantwortlich ist. Dieser wird automatisch zu Beginn des Events gestartet und vom Nutzer gestoppt, sobald dieser seine Teilnahme beendet. Hier muss der Lifecycle von

Services beachtet werden. Einige unerwünschte Ereignisse könnten den Service vorzeitig beenden, was den Verlust der Daten herbeiführen würde. Um dies zu verhindern muss der *LocationTransmitterService* ein Vordergrundservice sein, der bei der Resourcenfreigabe durch das System besonders berücksichtigt wird. Der Service wird um eine Notification erweitert, welche den Nutzer über seine Aktivität informiert und in Form eines „Stopp“ Buttons eine Interaktion mit dem Benutzer ermöglicht (Abb. 17).

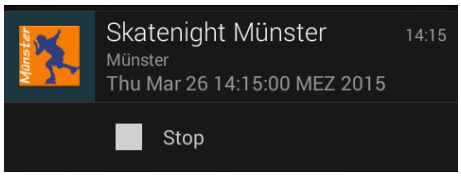


Abbildung 17: Anzeige der Notification zur Kontrolle des *LocationTransmitterService*

Durch den Aufruf der *startForeground(int, Notification)* Methode, mit einer einzigartigen ID und besagter Notification, wird der Service in einen Vordergrund Status versetzt. Dies geschieht nach der Initialisierung in der *onStartCommand(Intent, int, int)* Methode des Services.
Der Service sollte ursprünglich mit Hilfe eine Push-Notification gestartet werden, dies konnte allerdings nicht verlässlich realisiert werden, da diese teilweise garnicht oder zu häufig gesendet wurden. Um dennoch eine konsistente Datenerhebung zu gewährleisten, ohne einen Start der App vorauszusetzen, muss der Service automatisch vom System zur passenden Zeit gestartet werden. Dies ist mit Hilfe des Android AlarmManager eingeschränkt möglich. Da keine Referenzen auf registrierten Alarmzeiten gespeichert werden können, ist es nicht möglich diese nachträglich zu ändern. Da jedoch die Möglichkeit besteht, den Startzeitpunkt eines Events zu verändern, muss diese Funktionalität vorhanden sein. Gelöst haben wir dieses Problem, indem bei jeder Startzeitänderung ein neuer Alarm mit einzigartiger ID erstellt wird welche in den SharedPreferences für das Event gespeichert wird. Der ausgelöste Alarm ruft den *StartServiceReceiver* auf welcher zunächst überprüft ob die AlarmID mit der für das Event gespeicherten ID übereinstimmt. Ist dies nicht der Fall, so bedeutet dies eine Änderung der Alarmzeit. Es existiert also bereits ein aktuellerer Alarm. Falls der Alarm verantwortlich ist und der Service noch nicht gestartet wurde, wird dies nun erledigt.
Das betätigen des „Stopp“ Buttons der Notification ruft den *CancelServiceReceiver* auf, welcher den Service beendet.
Der *LocationTransmitterService* benötigt diverse Metainformationen, wie z.B. die Routendaten und Eventinformationen, zum Ausführen der Datenanalyse. Diese werden beim Starten des Services als Extras übergeben. Zur Ermittlung der Position wird die LocationServices API der Google Play Services verwendet, bei der wir möglichst genaue Positionen im 1-Sekunden Intervall abfragen. Im Falle eines erfassten Positionswechsels ruft die LocationServices API die *onLocationChanged(Location)* Methode des *LocationTransmitterService* auf.
Zunächst wird mithilfe des durch die *calculateCurrentWaypoint(LatLong)* Methode aufgerufenen Routenpunktberechnungsalgorithmus (siehe Kapitel 5.7.1) der aktuelle Routenpunkt des Nutzers ermittelt und anschließend, falls sich der neu berechnete Routenpunkt vom bereits hinterlegten unterscheidet, zusammen mit der aktuellen Systemzeit, in Millisekunden seit Mitternacht des 1. Januar 1970 (UTC), abgespeichert. Nun werden die optional in der Location enthaltenen Informationen, Geschwindigkeit und Höhe, verarbeitet. Falls die Location diese Informationen enthält, wird zunächst die Geschwindigkeit von m/s auf Km/h umgerechnet und mit der bisher maximal festgestellten Geschwindigkeit verglichen. Wenn bei der Höhe eine positive Änderung erfasst wird, so wird diese Änderung zu den zurückgelegten Höhenmetern addiert. Die durchschnittliche Geschwindigkeit wird berechnet indem die bereits zurückgelegte Strecke durch die Zeit seit dem Start des Events dividiert wird. Die so ermittelten Daten werden an die registrierten BroadcastReceiver gesendet.

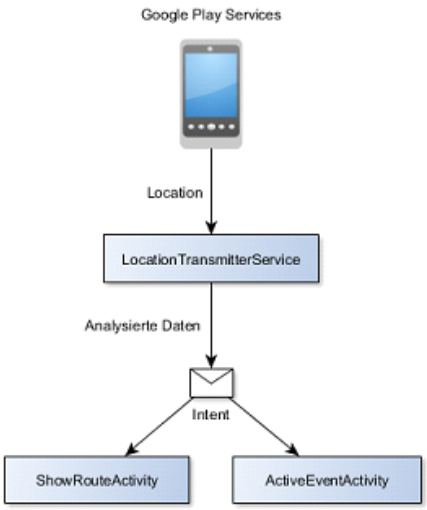


Abbildung 18: Verarbeitung einer Positionsänderung

Beim ordentlichen Beenden des *LocationTransmitterService* wird dessen *onDestroy()* Methode aufgerufen. Hier werden die erhobenen Daten in ein *LocalAnalysisData* Objekt überführt und anschließend mithilfe der *LocalStorageUtil* Klasse in den SharedPreferences persistent gespeichert. Die *LocalStorageUtil* verwendet hierfür die externe Bibliothek *gson-2.2.2.jar* welche beliebige Objekte zu JSON-Strings umwandeln kann.

ActiveEventActivity

Die *ActiveEventActivity* ist für das Anzeigen der sowohl in Echtzeit ermittelten als auch der gespeicherten Daten verantwortlich. Sie wird von der *ShowEventsActivity* als Alternative für die *ShowInformationActivity* aufgerufen, wenn das gewünschte Event bereits stattfindet und der Benutzer als Teilnehmer für dieses Event eingetragen ist.
Da die Berechnung der Daten im *LocationTransmitterService* stattfindet müssen diese lediglich in das GUI eingetragen werden. Alle Daten, bis auf die Fortschrittsanzeige, werden als einfache TextViews dargestellt, die Fortschrittsanzeige hingegen wird als Progressbar realisiert. Die beiden Buttons „Geschwindigkeitsprofil“ und „Karte“ rufen die *ShowRouteActivity* mit jeweils anderen Parametern auf.

Geschwindigkeitsprofil

Das Geschwindigkeitsprofil gibt sowohl Auskunft über die tatsächlich zurückgelegte Strecke, als auch über die Durchschnittsgeschwindigkeit welche der Teilnehmer zwischen den einzelnen Streckenabschnitten erreichte. Dies wird dargestellt, indem eine farbige Polyline über die besuchten Wegpunkte gelegt wird. Anhand dieser Einfärbung kann der Benutzer eine Abschätzung über sein Geschwindigkeitsprofil treffen und zwar werden langsame Abschnitte Rot, mittelschnelle Grün und schnelle Cyan unterlegt.
Die *ShowRouteActivity* welche für diese Darstellung verantwortlich ist unterscheidet zwischen einem bereits aufgezeichneten Profil und einem gerade entstehenden. Für eine bereits aufgezeichnete Strecke übergibt die *ActiveEventActivity* die Indizes der besuchten Routenpunkte und die dazugehörigen Zeitstempel. Diese Daten werden benötigt um das Geschwindigkeitsprofil mithilfe der *createRouteHightlight(int[], long[])* Methode zu rekonstruieren. Wird das Profil noch aufgezeichnet, so werden stattdessen die Extras *EventID* und *SpeedProfile (boolean)* übergeben. In diesem Fall registriert sich die *ShowRouteActivity* als BroadcastReveiver für die Positionsänderungen des *LocationTransmitterService* . Wenn eine Positionsänderung stattfindet werden die besuchten Routenpunkte und Zeitstempel an die *ShowRouteActivity* weitergeleitet.

Abbildung 19: Anzeige der lokalen Auswertung

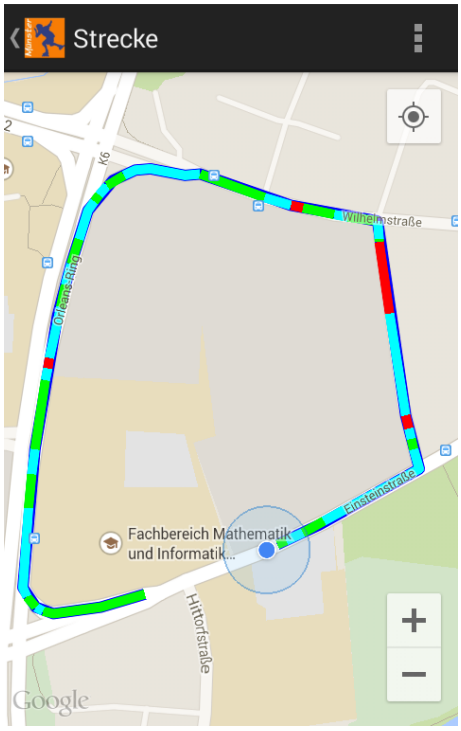


Abbildung 20: Anzeige des Geschwindigkeitsprofils

Die *createRouteHightlight(int[], long[])* Methode zeichnet den zurückgelegten Weg auf der Karte ein. Sie ermittelt mithilfe der Zeitstempel und der Distanz zwischen den Routenpunkten die Durchschnittsgeschwindigkeit des Skater. Da Polylines nur eine einzelne Farbe unterstützen, werden bei Änderung der Geschwindigkeit neue Polylines an die bereits gezeichneten angehängt. Die Methode *colorForSpeed(float)* ermittelt die benötigte Farbe abhängig von der übergebenen

Geschwindigkeit in Km/h.

6.3.5 Einstellungen

von Tristan

Die *SettingsActivity* zeigt dem Nutzer das Einstellungsmenü an. Hier kann er auswählen, ob seine Position innerhalb der App an den Server gesendet werden darf oder nicht. Zudem nutzt die *SettingsActivity* ein *PreferenceFragment* . Dies ist für Android 3.0 (API Level 11) und höher bestimmt. Für Versionen vor Android 3.0 hätte die *SettingsActivity* von der *PreferenceActivity* erben müssen, was hier jedoch nicht der Fall ist, da Fragmente eine flexiblere Architektur für die Anwendung bereitstellen im Vergleich zu der Acitivity alleine.
Das hier genutzte Fragment ist das *SettingsFragment* , welches das Menü über eine XML Datei *preferences.xml* unter *res/xml/preferences.xml* generiert. Für die Einstellung zum Senden der eigenen Position wurde eine Checkbox verwendet, deren Standardwert auf true gesetzt ist. D.h. beim erstmaligen Starten der App ist das Senden der eigenen Position aktiviert. Die Einstellungen werden in den *SharedPreferences* als boolean gespeichert unter dem Namen „prefSendLocation“.

7 Testing

7.1 Einleitung

von Tristan

In dem Projekt gibt es zwei Arten von Tests, die „Task Tests“ und die „Anwendungsfalltests“ (Use Case Tests). Die Tasks Tests dienen hierbei zum Testen einer bestimmten Funktionalität, wie zu Beispiel dem Hinzufügen oder Löschen einer Route (*AddRouteTaskTest* und *DeleteRouteTaskTest*). Die Anwendungsfalltests sollen im Gegensatz zu den Tasks Tests nicht nur eine Funktionalität sondern ganze Anwendungsabläufe Testen, was die UI mit einschließt. Beide Tests, der Task Test und Anwendungsfalltest bestehen aus einem Konstruktor zum Initialisieren oder Aufsetzen der Testdaten, welcher einmalig im Test aufgerufen wird und einer *setUp* -Methode, die vor jedem einzelnen Test aufgerufen wird. Sie dient dazu Variablen zu initialisieren und Daten von vorherigen Tests „aufzuräumen“. Weiterhin gibt es bei dem Task Test eine einzige Methode, die ausschließlich die Funktion testet, für die der Test bestimmt ist (*testTask()* -Methode).

7.2 Funktionsweise der Anwendungsfalltests

von Tristan

Da die TaskTests an anderer Stelle genauer beschrieben werden und nicht die Komplexität der Anwendungsfalltests aufweisen, wird hier der allgemeine Ablaufe sowie die Funktionsweise dieser Tests genauer beschrieben.
Einfachheitshalber werden die Anwendungsfalltests ab jetzt nur noch als Tests bezeichnet und beziehen sich nur auf diesen Abschnitt. Die Testklassen verwenden als „Base Test Case Class“ die *ActivityInstrumentationTestCase2* Klasse, welche Methoden zur Interaktion mit der UI unter Testbedingungen zur Verfügung stellt und in dem Paket „android.test“ enthalten ist. Der Test erbt also von dieser Klasse. Zusätzlich wird hierbei die Activity angegeben, in der der Test starten soll. Zum Beispiel startet der Test bei *ActivityInstrumentationTestCase2<HoldTabsActivity>* in der *HoldtabsActivity* . Zudem ist die Reihenfolge, in der die Tests ausgeführt werden nicht vorgegeben und kann variieren.

Konstruktor

Weiterhin wird im Konstruktor lediglich der superclass Konstruktor auf gerufen und die Klasse der Start Activity übergeben (HoldtabsActivity.class).

setUp Methode

Die *setUp* -Methode wird vor jedem Test aufgerufen und dient wie bereits erwähnt der Initialisierung und dem Löschen der Testdaten vorheriger Tests, da ein Anwendungsfall mehrere Test Methoden enthalten kann. Zuerst wird die superclass-Methode aufgerufen (super.setUp) und anschließen folgt der Initialisierungscode. Um die UI ohne Fehler von außen Testen zu können wird der Touch Mode mit *setActivityInitialTouchMode(false)* ausgeschaltet. In der Regel wird hier auch die Activity gestartet, falls dies noch nicht der Fall sein sollte (*getActivity()*) und einem Klassenattribut *mActivity* zugewiesen.

testPreConditions Test

Stellt sicher, dass die Vorbedingungen für die Anwendung fehlerfrei initialisiert wurden. Dazu gehören unter anderem die UI Elemente und das Starten der Activity.

testUseCase Test

Alle Tests, die auch die UI Testen, laufen auf einem extra dafür vorgesehen Thread, dem Ui-Thread. Auf diesem werden alle Interaktionen wie Buttonklicks oder Swipen simuliert. In den meisten Fällen werden Key Events simuliert mit *sendKeys* in dem UIThread und anschließen folgt die Überprüfung mit Hilfe von „Asserts“ wie in Junit. Diese prüfen, ob die Werte zum Beispiel die über den UI Thread eingegebenen Daten in Editfelder mit den erwarteten Werten übereinstimmen. Ist dies der Fall fährt der Test fort und endet, wenn alle Asserts fehlerfrei waren. Sollte jedoch ein Assert fehlschlagen, wird die Methode *testUseCase* abgebrochen und gilt somit als nicht bestanden. Außerdem können innerhalb des Tests auch andere Activities gestartet werden. Diese laufen dann über einen *Instrumentation.ActivityMonitor* über den die gestartete Activity überwacht wird, anschließen wird der Test Code ausgeführt und die Activity wieder geschlossen mit dem Befehl *finish()* . Sollte diese Activiy an dieser Stelle nicht über *finish()* beendet werden und der *testUseCase()* erfolgreich abgeschlossen worden sein, läuft diese Activity im Hintergrund über den Monitor weiter und kann zu fehlerhaften Ergebnissen in anderen Tests führen.

State Management Tests

Diese Art von Tests verifizieren die Activity in den Status „Pausieren“ oder „Beendet“. Folglich das Verhalten der Activity nach dem Fortsetzen oder Neustarten. Diese Art von Tests sind nicht in allen Anwendungsfalltests enthalten. Der Test *testStateDestroy* verwendet die von der Klasse *InstrumentationTestCase2* bereit gestellte Methode *fininsh* (*mActivity.finish()*) zum Beenden der Activity und der Test *testStatePause* die Methoden *callActivityOnPause* und *callActivityOnResume* . Diese halten die App an, was beispielsweise ein Klick des Nutzers auf den „Home Screen Button“ sein könnte.
Innerhalb des Tests wurde häufiger der Befehl *Thead.sleep(Zeitangabe)* verwendet, was den Hintergrund hat bei zeitkritischen Abschnitten eine bestimmte Zeit lang zu warten, damit zum Beispiel Daten vom Server über Netzwerkverbindungen, UI Elemente rechtzeitig initialisiert werden (die Zeit kann Testgeräte abhängig sein) oder die GPS Position ermittelt werden kann.
Auch kommen in den Testklassen die Begriffe „Small“, „Medium“ und „Large“ – Test vor. Deren Zweck ist es über den „Scope“, Abhängigkeiten und Performance Faktoren der Tests zu informieren, die Code-Qualität sowie System Instandhaltung zu verbessern.

Small (Unit): Verifiziert kleine „low-level“ Logik meist im Bereich einer Methode oder Klasse und bezieht sich auf keine externen Ressourcen. Die Ausführungszeit sollte im Sekunden (besser Millisekunden) Bereich liegen.
Medium (Integration): Kann mehrere Interaktionen zwischen Komponenten enthalten und auch auf externe Datei Systeme zugreifen oder mehrere Prozesse laufen lassen. Die Ausführungszeit kann Minuten betragen (vorzugsweise Sekunden).
Large (System): Hier kann es sein das der Test auch auf externe Ressourcen, wie Server über das Internet zu greifen muss. Die Ausführungszeit kann hier also etwas länger betragen, von Sekunden, Minuten oder Stunden.

7.3 Simulator

von Richard

Zu besseren Testbarkeit der App ist während der Entwicklung auch ein Simulator für eine Skatenight entstanden. Der Simulator ist in JavaScript geschrieben und benutzt JQuery, sowie die Google Maps API v3 als Bibliotheken.
Im Quelltext des Simulators kann in Zeile 50 die Adresse des Backends eingegeben werden, dass zur Simulation genutzt werden soll. So hat man die Möglichkeit, nicht nur auf dem Debug-Server sondern auch auf dem Jenkins- und Release-Server Felder zu simulieren. Nach dem Starten des Simulators werden dann die Events des entsprechenden Servers automatisch heruntergeladen und in dem Dropdown-Menü angezeigt. Wenn ein Event ausgewählt wird, wird die Strecke des Events als roter Polygonenzug auf der Karte dargestellt. Es kann unterhalb der Dropdownliste angegeben werden, wieviele Skater simuliert werden sollen, wie schnell sie fahren, wie groß das Feld ist, also auf welcher Fläche sie sich verteilen sollen, und in welchen Abständen die Position der Skater auf den Server übertragen werden soll. Mit den Schaltfläche „Start“, „Pause“ und „Stop“ kann die Simulation gesteuert werden.
Zur Simulation der Skater wird auf dem Server die Methode *simulateMemberLocations* aufgerufen. Diese nimmt die Positionen für viele Skater gleichzeitig entgegen und ruft anschließend jeweils die eigentliche Methode zur Aktualisierung der Position *updateMemberLocation* auf. Wir haben uns für diese Lösung entschieden, da Probleme auftraten, wenn der Simulator direkt die *updateMemberLocation* -Methode nutzte. Die zu große Anzahl Anfragen hat den kostenlosen Rahmen der Google App Engine innerhalb einer sehr kurzen Zeit aufgebraucht. Über die Methode zur Simulation der Positionen wird unabhängig von der Anzahl der Skater immer nur eine Anfrage gestellt.

7.4 Task-Tests

von Bernd

Fast alle TaskTests befinden sich in der Veranstalterapp angefangen mit dem *AddRouteTaskTest* . Dieser Test erstellt eine neue Route und löscht als Vorbereitung alle Routen und Veranstaltungen vom Server. Im eigentlichen Test wird dann die erstellte Route mit dem *AddRouteTask* dem Server hinzugefügt und es wird geprüft, ob diese dann auf dem Server gespeichert ist, indem die *getRoutes(...)* Methode auf dem Server aufgerufen wird.
Der *CreateEventTaskTest* ist ähnlich wie der AddRouteTaskTest. Er erstellt eine Route, eine Veranstaltung und fügt der Veranstaltung die Route hinzu, da Veranstaltungen nicht ohne Routen gespeichert werden können. Als Vorbereitung werden auch hier alle Routen und Veranstaltungen gelöscht und es wird auf dem Server die *addRoute(...)* Methode und die *createEvent(...)* Methode aufgerufen. Im eigentlichen Test wird dann auf dem Server der *QueryEventTask* aufgerufen, um die erstellte Veranstaltung zu überprüfen.

Der *DeleteEventTaskTest* erstellt eine Route und zwei Veranstaltungen. Als Vorbereitung werden wieder alle Veranstaltungen und Routen vom Server gelöscht und dann die beiden Veranstaltungen mit der Route auf dem Server gespeichert. Im eigentlichen Test wird dann eine Veranstaltung vom Server durch Aufrufen des *DeleteEventsTasks* gelöscht und es wird überprüft, ob die übrig gebliebene Veranstaltung die richtige ist.
Der *DeleteRouteTaskTest* erstellt drei Routen und als Vorbereitung werden alle Routen und Veranstaltungen vom Server gelöscht und die drei Routen dem Server hinzugefügt. Im eigentlichen Test wird dann eine Route vom Server durch Aufrufen des *DeleteRouteTasks* gelöscht und es wird geprüft, ob die gelöschte Route noch auf dem Server vorhanden ist. Dies geschieht, indem auf dem Server die *getRoutes(...)* Methode aufgerufen wird und die erhaltene Liste von Routen auf die gelöschte Route untersucht wird.
Der *EditEventTaskTest* erstellt eine Veranstaltung und zwei Routen. Als Vorbereitung werden alle Veranstaltungen vom Server gelöscht und die neue Veranstaltung dem Server hinzugefügt. Im eigentlichen Test werden die Daten der angelegten Veranstaltung geändert und es wird der *EditEventTask* aufgerufen. Abschließend werden die veränderten Daten der Veranstaltung überprüft.
Der *GetEventTaskTest* erstellt eine Veranstaltung und eine Route. Als Vorbereitung werden alle Veranstaltungen vom Server gelöscht und die neue Veranstaltung dem Server hinzugefügt. Im eigentlichen Test wird dann die *getEvent(...)* Methode auf dem Server aufgerufen und die Daten der erhaltenen Verantstaltung mit der erstellten verglichen. Abschließend werden alle Routen und Veranstaltungen vom Server gelöscht.
Der *QueryEventTaskTest* erstellt zwei Veranstaltungen und zwei Routen. Als Vorbereitung werden alle Veranstaltungen und Routen vom Server gelöscht und die beiden Veranstaltungen mit den beiden Routen auf dem Server gespeichert. Im eigentlichen Test werden dann alle Veranstaltungen vom Server mit dem *QueryEventTask* abgefragt und es wird überprüft, ob die erhaltene Liste von Veranstaltungen die beiden erstellten Veranstaltungen beinhaltet. Abschließend werden alle Veranstaltungen und Routen vom Server gelöscht.
Der *QueryRouteTaskTest* erstellt drei Routen. Als Vorbereitung werden alle Routen und Veranstaltungen vom Server gelöscht und die drei erstellten Routen dem Server hinzugefügt. Im eigentlichen Test werden die Routen vom Server abgefragt indem der *QueryRouteTask* aufgerufen wird und die erhaltene Liste von Routen auf die drei erstellten Routen untersucht wird. Abschließend werden alle Routen vom Server gelöscht.

7.5 Use-Case-Tests: Veranstalter-App

7.5.1 AddAndDeleteHostTest

von Bernd

7.5.2 AuthenticationOrganizerTest

von Bernd

7.5.3 CreateAndDeleteRouteTest

von Richard

Der *CreateAndDeleteRouteTest* legt, wie der Name schon sagt, eine Route an und löscht sie anschließend wieder. Da es sich um einen Use-Case-Tests handelt, werden die Abläufe vollständig

über die GUI gesteuert. Lediglich beim Löschen der Route wird nicht über die GUI-Elemente das Löschen angestoßen, sondern direkt die entsprechende Methode auf dem Fragment zur Verwaltung der Routen aufgerufen. Dies ist notwendig, da wir in den Tests das Auswählen von Optionen in einem Kontextmenü nicht an allen Stellen simulieren konnten. Der weitere Ablauf des Löschens wird aber exakt wie in der Veranstalter-App abgearbeitet.
Im Test wird nach einer kurzen Wartezeit in der initialisierten View auf den Plus-Button im *ManageRoutesFragment* gedrückt. Es öffnet sich dadurch der Dialog, der den Namen der neuen Route entgegennimmt. Nach Eintragen eines Teststrings, wird mit der OK-Taste bestätigt und es öffnet sich der Routeneditor. Nach einer erneuten Wartezeit ist die Route vollständig geladen und es werden zwei Wegpunkte über den Plus-Button in der ActionBar erstellt. Nach Erstellung des ersten Wegpunktes wird die Karte ein Stück verschoben, damit sich der zweite Wegpunkt an einer anderen Position befindet. Durch die Simulation eines Zurück-Tastenklicks wird der Sicherungsdialog angezeigt, der bestätigt und damit geschlossen wird. Der Test wartet dann, bis die neue Route angelegt wurde und löscht diese über den bereits erwähnten Aufruf beim *ManageRoutesFragment* .

7.5.4 PublishNewInformationTest

von Tristan

Anwendungsfallbeschreibung: Veröffentlichen neuer Informationen 2.0
Beteiligte Akteure: Veranstalter
Anfangsbedingungen: Der Veranstalter hat die Veranstalter-App geöffnet, ist bereits eingeloggt und befindet sich in der *HoldTabsActivity* auf dem Veranstaltungen Tab.
Ereignisfluss:

1. Es wird auf das Tab zum Event erstellen geswipet.
2. Es werden alle Felder für das Event ausgefüllt bzw. neue Eigenschaften hinzugefügt.
3. Über „Erstellen“ wird das Event erstellt.
4. Die Erstellung wird über einen Dialog nochmal bestätigt.
5. Ab dem ersten Schritt wiederholen, um mehrere Events anzulegen.
6. Es wird zu dem Veranstaltungen Tab geswipet.
7. Von den angezeigten Events wird eines ausgewählt.
8. Es wird ein Menü zur Bearbeitung des ausgewählten Events angezeigt.
9. Bereits existierende Informationen können bearbeitet oder über den Button „Editiere Event Eigenschaften“ neue Eigenschaften hinzugefügt werden.
10. Das Event wird über den Speichern Button editiert.

Abschlussbedingungen: Neue Veranstaltung anlegen und diese anschließend editieren.
Ausnahmen: Keine
Spezielle Anforderungen: Internetverbindung

Testablauf: Vorbedingung für den Test ist eine bestehende Internetverbindung, damit das anzulegende Event auf dem Server gespeichert werden kann. Die Test Daten für die Klasse sind als private, statische sowie konstante Member der Klasse definiert (TEST_TITLE etc.). Dabei sind die Daten mit dem Präfix TEST_ für den normalen useCase Test und StatePaueResume Test vorgesehen und die mit dem Präfix TEST_STATE_DESTROY für den StateDestroy Test. Zudem sind die UI Elemente als private Member deklariert. Der Test läuft über die HoldTabsActivity der „VeranstalterApp". Das hier angelegte Test Event wird an anderer Stelle in einem Test der User-App *ShowServeralEventsTest* verifiziert und wird somit nach Beendigung des Tests nicht gelöscht.
setUp(): Hier wird für den Nutzer das Google Konto ausgewählt zu Authentifizierung, zum „Veranstaltungen Erstellen" Tab geswipt sowie die App gestartet.
Swipe(Direction): Da die InstrumentationTestCase2 Klasse keine Methoden zum swipen für das wechseln der Tabs bereitstellt, wird diese Funktionalität hier definiert. Es kann entweder right oder left (privates enum) der Klasse übergeben werden und führt dann die entsprechende Tätigkeit aus.
testViews(): Activity, Testet, ob alle UI Elemente, Edits, Buttons, TimePicker etc. initialisiert wurden in der setUp Methode mit Hilfe von assertNotNull(UIElement).
testViewsVisible(): Verifiziert lediglich ob die Elemente auch auf dem Smartphone Screen sichtbar sind (assertOnScreen bereitsgellte Methode der InstrumentationTestCase2).
testPreConditions(): Prüft, ob alle Tabas in der ActionBar (mActionBar) enthalten sind.
testUseCase(): Kann eine Exception werfen auf Grund des UI Threads. Hier wird der Anwendungsfall verifiziert. Hierzu wird auf das Tab zum erstellen geswipt, die Test Daten eingegeben und abgebrochen um auch diese Funktionalität zu testen. Danach werden die Test Daten erneut eingegeben und auf den „Erstellen" Button gedrückt und mit einem auftauchenden Dialog nochmal bestätigt wird. Es wird geprüft, ob das Event auf dem Server gespeichert wurde. Danach wird zu den Veranstaltungen Tab geswipt und das eben angelegt Event mit einem Long Touch zum Editieren ausgewählt. Die Testen zum Editieren werden eingegeben und wie vorhin wird Bestätigt und zu dem Veranstaltung Tab geswipt sowie geprüft, ob das editierte Event auch auf dem Server gespeichert wurde.
State Tests: Hier wird getestet, ob die Daten beim Beenden der App aus den Eingabefelder gelöscht werden, falls welche beim Erstellen schon eingegeben worden sind oder bestehen bleiben, beim Pausieren der App.
Anmerkung: Die weiteren Tests *SendCurrentPositionTest* , *SendPositionSettingsTest* und *ShowSeveralEventsTest* werden nicht so genau geschrieben, sondern auf JavaDoc und Inline Kommentare verwiesen.

7.6 Use-Case-Tests: User-App

7.6.1 CreateJoinLeaveDeleteUserGroupTest

von Bernd

Dieser Test ist ein Use-Case in dem eine Nutzergruppe erstellt wird, einer anderen Nutzergruppe, bei der man noch nicht Mitglied ist, beigetreten wird, diese dann wieder verlässt und die erstellte Nutzergruppe zum Schluss wieder gelöscht wird. Damit dieser Test funktioniert muss vor Ausführung schon eine Nutzergruppe existieren, bei der der auf dem Testhandy angemeldete Benutzer nicht schon ein Mitglied ist. Des weiteren darf noch keine Nutzergruppe mit dem Namen „TestGroup" auf dem Server existieren.
Zu Beginn des Tests wird der Benutzer mit dem Server verbunden, das heißt es wird seine E-Mail an den Server gesendet und es wird die *UsergroupActivity* gestartet. Im ersten Teil des Tests werden die Views getestet, das heißt es wird geprüft, ob sowohl die ActionBar als auch der sich darauf befindende Plus Button richtig referenziert wurden. Im zweiten Teil beginnnt

der Use-Case, in dem als erstet ein Klick auf den Plus Button simuliert wird. Es wird dann so lange gewartet, bis sich die *AddUserGroupActivity* gestartet hat und dort wird dann in das Feld für die Nutzergruppe der Name „TestGroup“ eingetragen und es wird ein Klick auf den „Ok“-Button simmuliert. Es wird dann wieder auf die *AddUserGroupActivity* gewartet und in der ListView vom *AllUsergroupsFragment* nach der Nutzergruppe mit dem Namen „TestGroup“ gesucht. Ist die Suche erfolglos so schlägt der Test fehl, bei erfolgreichem Suchen läuft der Test weiter. Als nächstes wird die ListView nach einer Nutzergruppe durchsucht, bei der der Benutzer noch nicht als Mitglied eingetragen ist. Ist keine vorhanden, so schlägt der Test mit einer Fehlermeldung fehl. Ist eine Nutzergruppe vorhanden, bei der der Benutzer noch kein Mitglied ist so wird ein Klick auf die Stelle in der ListView simuliert, bei der die Nutzergruppe gefunden wurde. Dies startet dann einen AlertDialog zum beitreten einer Nutzergruppe. Es wird so lange gewartet, bist der Dialog angezeigt wird. Es wird ein Klick auf den Ja Button simuliert um der Nutzergruppe beizutreten. Darauf hin wird überprüft, ob der Bediener nun in der Liste von Mitgliedern der soeben beigetretenen Nutzergruppe enthalten ist. Ist dies nicht der Fall, so schlägt der Test fehl ansonsten läuft der Test weiter. Nun wird ein Klick auf die Stelle in der ListView simuliert, bei der sich die Nutzergruppe befindet der man soeben beigetreten ist. Auch hier wird ein AlertDialog zum Verlassen der Nutzergruppe angezeigt. Es wird wieder so lange gewartet, bis der AlertDialog erscheint. Hier wird wieder ein Klick auf den Ja Button simuliert um die Nutzergruppe zu verlassen. Nun wird überprüft, ob der Benutzer in der Liste von Mitgliedern der Nutzergruppe enthalten ist. Ist dies der Fall so schlägt der Test fehl ansonsten läuft der Test weiter. Zum Schluss sollte an der Stelle in der ListView an der sich die erstellte Nutzergruppe befindet ein „Longclick“ simuliert werden, damit der AlertDialog zum Löschen einer Nutzergruppe erscheint. Dies konnte von uns in dem automatischen Test jedoch nicht umgesetzt werden. Als Lösung wurde dann an dieser Stelle der *DeleteUserGroupTask* aufgerufen. Zum Schluss wird noch geprüft ob die Nutzergruppe noch auf dem Server existiert. Ist dies der Fall so schlägt der Test fehl ansonsten wird der Test erfolgreich beendet.

7.6.2 SendCurrentPositionTest

von Tristan

Anwendungsfallbeschreibung: Übertragung der aktuellen Position an den Server
Beteiligte Akteure: Teilnehmer
Anfangsbedingungen: Der Teilnehmer hat die User-App geöffnet, ist bereits eingeloggt und befindet sich in der *ShowEventsActivity* .
Ereignisfluss:

1. Erfassen der Position in der App mittels GPS bzw. WLAN.
2. Regelmäßige Übertragung der Position an den Server.

Abschlussbedingungen: Regelmäßige Übertragung des Teilnehmerstandortes an den Server, wenn der Teilnehmer dies aktiviert hat.
Ausnahmen: Keine
Spezielle Anforderungen: Internetverbindung, GPS

Testablauf: In diesem Test steht das Senden der Position mit Hilfe des *LocationTransmitterService* im Vordergrund und nicht der Ablauf der Anwendung des Nutzers, d.h. es wird auf eine korrekte Reihenfolge (Starten, Einstellungsmenü) verzichtet. Es wird in der *SettingsAcitivty* gestartet und die Checkbox für die Übertragung aktiviert, worauf hin der *LocationTransmitterService* gestartet wird. Anschließend wird die Checkbox wieder deaktiviert und geprüft, ob der *LocationTransmitterService* auch wirklich gestoppt wird. Weiterhin wird getestet ob die Position an den Server gesendet wird, selbst wenn die App pausiert.

Anmerkung: Da das Menü über eine xml. Datei generiert wird, lassen sich die einzelnen UI Elemente nicht genau ansprechen.

7.6.3 SendPositionSettingsTest

von Tristan

Anwendungsfallbeschreibung: Handhabung der aktuellen Position
Beteiligte Akteure: Teilnehmer
Anfangsbedingungen: Der Teilnehmer hat die User-App geöffnet, ist bereits eingeloggt und befindet sich in der *ShowEventsActivity* .
Ereignisfluss:

1. Es wird mit einem Klick in dem Einstellungsmenü „Einstellungen“ ausgewählt.
2. Es wird die Checkbox zum Übertragen der Informationen an ausgewählt oder auch nicht.

Abschlussbedingungen: Die Teilnehmer-Position wird je nach Einstellung an den Server gesendet oder nicht.
Ausnahmen: Keine
Spezielle Anforderungen: Internetverbindung, GPS

Testablauf: Diese Klasse besteht aus verschiedenen Tests. Den UI Tests und einem Anwendungsfalltest. Die UI Tests dienen lediglich dem Überprüfen der Funktionalität des Einstellungsmenüs. Hierzu wird in der *SettingsActivity* die Checkbox aktiviert, geprüft ob die Position an den Server gesendet wird. Danach wird die Checkbox wieder deaktiviert, wodurch die Position sich auf dem Server nicht verändert sollte. Vor jedem Test werden die Positionsdaten auf dem Server zurückgesetzt damit es nicht zu Fehlern kommen kann. Die unveränderte Position befindet sich in China. Der Test schlägt nur fehl, wenn sich das Testgerät genau an dieser Position aufhält, da der *LocationsTransmitterService* die Position nicht auf dem Server verändert.
Der eigentliche UseCaseTest läuft wie in dem Ereignisfluss in der Anwendungsfallbeschreibung ab. Zusätzlich gibt es noch einen Teilnehmen *testAttend()* Test. Hier wird in der *ShowEventsActivity* gestartet, anschließend ein Event ausgewählt, auf die *ShowInformationActivity* gewechselt und der Teilnehmen-Button bestätigt. Vor dem Test wird sichergestellt, dass das Senden in den Einstellungen aktiviert ist. Anschließend wird geprüft, ob an den Server gesendet wird. Dies hat den Hintergrund, dass der *LocationTransmitterService* zum Senden gestartet wird, wenn man an an einem Event teilnimmt, das bereitsgestartet ist.

7.6.4 ShowSeveralEventsTest

von Tristan

Anwendungsfallbeschreibung: Anzeigen mehrerer Veranstaltungen
Beteiligte Akteure: Teilnehmer
Anfangsbedingungen: Der Teilnehmer hat die User-App geöffnet, ist bereits eingeloggt und befindet sich in der *ShowEventsActivity* .
Ereignisfluss:

1. Es wird eine Liste aller Veranstaltungen angezeigt.
2. Auswahl einer Veranstaltung.
3. Die Veranstaltung wird detailliert mit allen Informationen angezeigt.

Abschlussbedingungen: Anzeigen aller Veranstaltungen. Auswahl einer Veranstaltung und Anzeigen von Informationen zu dieser.
Ausnahmen: Keine
Spezielle Anforderungen: Internetverbindung

Testablauf: Vorbedingung für den Test ist eine bestehende Internetverbindung, damit die Events von dem Server abgerufen werden können. Für diesen Test muss zuerst der *PublishNewInformationTest* in der Veranstalter-App ausgeführt worden sein. Dieser Test wird in der User-App ausgeführt und soll überprüfen, ob das Event, das durch den *PublishNewInformationTest* angelegt wurde, auch in dieser App abrufbar und fehlerfrei ist. Es wird zuerst die App gestartet und die Liste aller verfügbaren Events angezeigt. Es wird das erste Event ausgewählt (welches das aus dem anderen Test sein sollte). Anschließend werden die Details zu dem Event genauer angezeigt in der *ShowInformationActivity* und die Daten aus der Veranstalter-App verglichen.