Kort Reloaded – A Gamified App for Collecting OpenStreetMap Data

Bachelorarbeit

Abteilung Informatik

HSR Hochschule für Technik Rapperswil

Frühjahrssemester 2016

Autoren:	**Marino Melchiori** **Dominic Mülhaupt**
Betreuer:	**Prof. Stefan Keller**
Projektpartner:	**Jürg Hunziker** **Stefan Oderbolz** Liip AG
Experte:	**Claude Eisenhut**
Gegenleser:	**Prof. Beat Stettler**

Impressum

Autoren:	Marino Melchiori (mmelchio@hsr.ch) Dominic Mülhaupt (dmuelhau@hsr.ch)
Dokument erstellt:	10.03.2016
Letzte Aktualisierung:	

Dieses Dokument wurde mit LaTeX erstellt.

Abstract

OpenStreetMap ist ein freies Projekt das von Benutzern aus der ganzen Welt unterstützt wird. Wege, Gebäude und viele andere geografische Daten werden weltweit in einer Datenbank erfasst und gepflegt. *OpenStreetMap* kann von jedem bearbeitet werden, besteht aus einer grossen Community und setzt auf lokales Wissen der «Mapper». Aus diesem Grund ist es nicht ausgeschlossen, dass fehlerhafte oder unvollständige Daten enthalten sind. Zur Korrektur der Daten gibt es viele verschiedene Tools, die von Experten genutzt werden können. Um eine breitere Masse anzusprechen, entstand 2012 die Web-App «Kort» im Rahmen einer Bachelorarbeit. Mit Kort kann der Benutzer Aufträge lösen, die zur Verbesserung der Daten in *OpenStreetMap* beitragen. Auf einer Kartenansicht werden die Aufträge, welche sich im Umfeld des Benutzers befinden, dargestellt. Für das Eintragen einer Lösung wird man mit Punkten (sogenannten «Koins») belohnt und kann so in der Rangliste aufsteigen. Da sich HTML5 weiterentwickelt hat, muss die Web-App abgelöst werden.

In dieser Arbeit wurde Kort als natives Mobile App für *Android* und *iOS* komplett neu entwickelt. Das bestehende Backend wurde dabei praktisch unverändert weitergenutzt. Die App basiert auf dem *React Native* Framework von *Facebook*. *React Native* ist eine moderne, noch junge Technologie, welche es ermöglicht, mit *JavaScript* Apps für *Android* und *iOS* zu entwickeln. Dabei wird *JavaScript*-Code in Komponenten der jeweiligen Smartphone-Plattform übersetzt, was dem Benutzer die Erfahrung einer nativen App bietet.

Es konnten wichtige Erfahrungen mit *React Native* gesammelt werden. Dabei sind eine *Android*- und eine *iOS*-App entstanden. Die *Android*-App befindet sich noch in der Beta-Phase und wird nach Abschluss dieser Arbeit im *Google Play* Store erwartet. Die *iOS*-App ist noch in der Testphase und wird später im *Apple* App Store veröffentlicht. Weitere Informationen: www.kort.ch.

Dank

Für die Betreuung während des ganzen Projektes möchten wir uns besonders bei Herr Prof. Stefan Keller bedanken. Bei Fragen zu *OpenStreetMap* konnte er uns während der gesamten Laufzeit der Arbeit unterstützen und er gab uns bei Problemen immer Inspirationen für eine Lösung. Ein Dank geht auch an Jürg Hunziker und Stefan Oderbolz, für die Unterstützung während des ganzen Projektes. Sie gaben uns Tipps zum Vorgehen und haben für uns grössere und wichtige Backend-Änderungen vorgenommen. Herr Prof. Stefan Keller, Jürg Hunziker und Stefan Oderbolz liessen uns freien Spielraum für die Suche nach einer optimalen Lösung. Ebenfalls bedanken wir uns bei Robert Vogt und Mirco Strässle, für ihre Tipps zu React und React-Native.

Aufgabenstellung

Kort Reloaded – A Gamified App for Collecting OpenStreetMap Data

Bachelorarbeit Frühjahrssemester 2016, Studiengang Informatik

Hintergrund

Kort ist das erste App, mit dem man spielerisch OpenStreetMap-Daten verbessern kann. OpenStreetMap ist das Wikipedia der Landkarte. Auf einer Karte werden Aufträge angezeigt, die zu erledigen sind. Spieler können bereits erledigte Aufträge auf ihre Korrektheit überprüfen. Sobald genügend positive Überprüfungen eingegangen sind, kann die Lösung eines Auftrags auf OpenStreetMap eingetragen werden.

Kort nutzt Gamification, um Benutzer zu motivieren, die App zu verwenden. So können durch das Lösen von Aufträgen oder das Überpüfen von Lösungsvorschlägen Punkte (sogenannte Koins) gesammelt werden. Ziel ist es, durch das Sammeln von Koins immer höher in der Highscore aufzusteigen. Zusätzlich wird ein Spieler für seinen Einsatz auch mit Auszeichnungen belohnt.

Aufgabenstellung

Der Kort-Client soll neu als native Android App zur Verfügung stehen. Das bedingt ein kompletter Rewrite des aktuellen JavaScript-Codes (aktuell Sencha Touch) mit dem Framework React Native. Es soll ein Erfahrungsbericht zu React Native erstellt werden.

Ziele:

- Gleiche Funktionalität mit neuem Framework, damit die mobile App mit den neusten Technologien arbeitet und künftig besser wartbar ist.
- Neue Erkenntnisse zu einem aktuellen Framework zur Realisierung von Native Mobile Apps "React Native".
- Getestete Software-Entwicklungsumgebung

Deliverables

Mindestens...

- Neuer Kort-Client als native Android-App
- Ablösung des Validationsmechanismus'
- Erfahrungsbericht mit Hinweisen zu Tutorials zu React Native
- Erweiterte Software-Entwicklungsumgebung
- Die vom Studiengang geforderten Lieferobjekte: Dokumentation, Management Summary, Abstract, Poster, Präsentation mit Stellwand, Zwischenpräsentation

Erweitert...

- Kort-Client als native iOS-App
- neue Funktion: Promotions anzeigen
- Kurzvideo

Die definitive Aufgabenstellung, Lieferobjekte und das Vorgehen werden am Kickoff (erste Semesterwoche) zusammen mit dem Industriepartner festgelegt. Die gemeinsam besprochene Aufgabenstellung wird ca. zwei Wochen nach Semesterbeginn aktualisiert.

Vorgaben/Rahmenbedingungen

- Die Kort App ist clientseitig in HTML5 und JavaScript geschrieben
- Serverseitig ist u.a. PHP und PostgreSQL mit der PostGIS-Erweiterung vorhanden.
- Technologie steht React Native im Vordergrund.
- Es wird genügend Zeit für die Einarbeitung in die Themengebiete einberechnet.
- Die Software soll Open Source sein.
- Die SW-Engineering-Methode und Meilensteine werden mit dem Betreuer vereinbart.
- Sourcecode und Software-Dokumentation sind Englisch (inkl. Installation, keine Benutzerdokumentation, höchstens eine Online-Kurzhilfe).
- Die Software-Benutzerschnittstelle ist mind. Deutsch und Englisch.
- Die Projekt-Dokumentation und -Präsentation sind auf Deutsch.
- Der Source Code, die Code-Kommentare und die Versionsverwaltung sind in Englisch.
- Die Nutzungsrechte an der Arbeit bleiben bei den Autoren und gehen auch an die HSR und den Betreuer über. Die Softwarelizenz ist „MIT“.
- Ein Video gemäss den Vorgaben des Studiengangs (kann ggf. nach dem Dokumentations-Abgabetermin abgegeben werden).
- Ansonsten gelten die Rahmenbedingungen, Vorgaben und Termine des Studiengangs Informatik bzw. der HSR.

Inhalt der Dokumentation

- Die fertige Arbeit muss folgende Inhalte haben:
  1. Abstract, Management Summary, Aufgabenstellung
  2. Technischer Bericht
  3. Projektdokumentation
  4. Anhänge (Literaturverzeichnis, Glossar, CD-Inhalt)
- Die Abgabe ist so zu gliedern, dass die obigen Inhalte klar erkenntlich und auffindbar sind.
- Zitate sind zu kennzeichnen, die Quelle ist anzugeben.
- Verwendete Dokumente und Literatur sind in einem Literaturverzeichnis aufzuführen.
- Dokumentation des Projektverlaufes, Planung etc.
- Weitere Dokumente (z.B. Kurzbeschreibung, Poster) gemäss www.hsr.ch und Absprache mit dem Betreuer.

Form der Dokumentation

Bericht, Dokumente und Quellen der erstellten Software gemäss Vorgaben des Studiengangs Informatik der HSR sowie Absprache mit dem Betreuer.

Bewertungsschema

Es gelten die üblichen Regelungen zum Ablauf und zur Bewertung der Arbeit (6 Aspekte) des Studiengangs Informatik der HSR jedoch mit besonderem Gewicht auf moderne Softwareentwicklung (Tests, Continuous Integration, einfach installierbar, funktionsfähig).

Beteiligte

Diplomanden

Marino Melchiori und Dominic Mülhaupt

Industriepartner

Jürg Hunziker und Stefan Oderbolz, Liip AG Zürich

Betreuung HSR

Verantwortlicher Dozent: Prof. Stefan Keller (sfkeller@hsr.ch), Geometa Lab am IFS der HSR

Inhaltsverzeichnis

Teil I.

Technischer Bericht

1. Einführung

1.1. Problemstellung

OpenStreetMap (*OSM*) beinhaltet eine sehr grosse Menge an Geodaten, welche frei zugänglich sind. Für die Pflege dieser Daten ist es daher naheliegend, auf unterstützende Software zurückzugreifen. Zu diesem Zweck gibt es eine Reihe von Applikationen, welche sich grob in zwei Kategorien einteilen lassen: Editoren und Tools zur Qualitätssicherung.

Mit den Editoren lässt sich direkt oder indirekt die *OSM*-Karte verändern und ergänzen. Die Qualitätssicherungstools haben sich zum Ziel gesetzt, fehlende oder falsche Daten aufzuspüren. Diese werden dann entweder automatisch korrigiert oder übersichtlich dargestellt, um eine manuelle Korrektur zu ermöglichen.

Einige Tools wie *KeepRight*[1] oder *Osmose*[2] berechnen aus den Karten-Rohdaten die vorhandenen Fehler. Dazu werden einige Heuristiken verwendet oder einfache Plausibilitätsüberprüfung durchgeführt. Typische Fehler aus diesen Quellen sind POIs ohne Namen oder Strassen ohne definierte Geschwindigkeitslimiten.[25]

Zur Behebung dieser Fehler ist die Web-App KORT[3] in Form einer Bachelorarbeit von Jürg Hunziker und Stefan Oderbolz, im Herbstsemester 2012/13, entwickelt worden.[25]

KORT wurde mit dem *Sencha Touch 2* Framework entwickelt. Da sich HTML5 weiterentwickelt hat, funktioniert die Implementation auf neuen Browsern nicht mehr sinngemäss. Die Ortung und die HTTP Requests funktionieren nur noch mit dem *Firefox*. *Google Chrome* erlaubt die Geolocation nur noch mit einer HTTPS-Verbindung und diese wird vom KORT-Backend nicht unterstützt.

Aus diesem Grund entstand die Idee, die KORT mit einem native Client zu ersetzen. Dazu bot sich *React-Native* an. Diese ganz neue Technologie ermöglicht es native *iOS* und - seit Oktober 2015 - auch *Android* Apps mit *JavaScript* zu erstellen. Dadurch, dass *React-Native* noch in den Kinderschuhen steckt und die Entwickler noch keine Erfahrung mit *JavaScript* haben, stellt dies ein grosses Risiko für diese Bachelorarbeit dar.

[1] http://keepright.ipax.at/
[2] http://osmose.openstreetmap.fr/map/
[3] http://play.kort.ch/

1.2. Ziele

Ziel ist es eine *Android* App mit gleicher Funktionalität wie die derzeitige Web-App zu erstellen. Zusätzlich gibt es optionale Ziele, wie das Erstellen der *iOS*-App.

Es entstand die Idee, den Validationsmechanismus von KORT abzulösen. Bis jetzt wurden viel zu viele Missionen gelöst, die nie von anderen Benutzern validiert worden sind. Das führte dazu, dass nur sehr wenige Änderungen überhaupt in der *OpenStreetMap*-Datenbank eingefügt wurden. Die *OSM*-Community forderte aber, dass die Antworten geprüft werden müssen. Validationsaufträge werden dem Benutzer nun einfach als normalen Auftrag angezeigt. Sobald die abgeschickte Antwort mit der zu validierenden Antwort übereinstimmt, wird im Hintergrund eine positive Bewertung versendet. Ab drei positiven Bewertungen gilt die Antwort als korrekt und sie schafft es in die *OSM*-Datenbank.

- Erstellen einer *Android* App mit *React-Native* - gleiche Funktionalität mit neuem Framework, damit die mobile App mit den neusten Technologien arbeitet und künftig besser wartbar ist.
- Als Basis sollen Daten und Webdienste des *OSM*-Projekts verwendet werden.
- Der Validationsmechanismus soll abgelöst werden.
- Es sollen neue Erkenntnisse zum aktuellen Framework (*React-Native*), zur Realisierung von native mobile Apps, gesammelt werden.
- Ein Erfahrungsbericht zu *React-Native* soll erstellt werden.
- Die Internationalisierung soll einfach umgesetzt sein.

1.3. Rahmenbedingungen

Die *React-Native* baut auf das bestehende KORT-Backend auf. Folglich geht des bei der Entwicklung nur um das Frontend - also um den native Client. Anpassungen am Backend lagen nicht im Aufgabenbereich der Entwickler. Einen Überblick über die Architektur und das Gesamtsystem von KORT, welches bereits bestand, gibt das Kapitel 8 der Bachelorarbeit von Jürg Hunziker und Stefan Oderbolz.[25]

- Es gelten die Rahmenbedingungen, Vorgaben und Termine der HSR.
- Die Projektabwicklung orientiert sich an einer iterativen, agilen Vorgehensweise. Als Vorgabe dient dabei Scrum.

1.4. Vorgehen

- Einarbeiten in *JavaScript* und *React Native* sowie damit verbundenen Technologien

- Einarbeiten in den Code und die Infrastruktur von Kort
- Iteratives Entwickeln des Prototyps
- Dokumentation abschliessen

Ziele und Resultate werden im technischen Bericht erläutert. Die Risikoanalyse, die Meilensteine und das Team befinden sich im Kapitel Projektmanagement. Der Source Code ist auf *GitHub* frei zugänglich.

2. Stand der Technik

Es gibt eine grosse Anzahl von Projekten, mit dem Ziel *OpenStreetMap* zu verbessern. Sowohl Editoren für erfahrene Benutzer, als auch Tools, die auf das finden und Beheben von Fehlern spezialisiert sind, werden angeboten. Zum Beispiel gibt es JOSM[1] (*Java* OSM *Editor*) als Desktop Client für *Windows*, *Mac OS X* und *Linux*. Sogar Geometrieobjekte sind damit editierbar. Daneben gibt es auch Dienste, welche Fehlerdaten sammeln und öffentlich anbieten. Ein bekanntes Beispiel ist *KeepRight*[2]. Es werden über 50 Fehlertypen angeboten. Mit diesen Werkzeugen können nun Daten korrigiert werden. Weitere Editoren sind auf dem *OSM*-Wiki[3] zu finden.

Die Zielgruppe dieser Werkzeuge sind Benutzer mit dem Interesse, die *OSM*-Daten zu verbessern. Um diesen Prozess effizienter zu gestalten, würde sich ein Crowdsourcing-Ansatz anbieten. Dafür muss die Zielgruppe aber erweitert werden. Ein sehr gutes Beispiel dazu ist *MapRoulette*[4]. Dem Benutzer werden Challenges präsentiert, die zum Beispiel ein Satellitenbild von einem Objekt anzeigen, das nach der Inspektion vom System einem Fussballfeld ähnlich sieht. Nun muss der Benutzer entscheiden, ob es sich tatsächlich um ein Fussballfeld handelt, oder nicht. Gamification durch Challenges bietet sich also sehr gut an. Die Spiel-Elemente halten den Spieler motiviert und binden ihn an das Spiel. Einige Projekte[5], die durch Gamification an *OSM*-Verbesserungen beitragen, sind bereits entstanden.[25]

2.1. Bestehende Lösungsansätze und Normen

Da wir die KORT-Web-App neu schreiben und es sich um eine Fortsetzungsarbeit handelt, übernehmen wir die bereits dort evaluierten Konzepte.

2.2. React Native

Aktuell befindet sich *React Native* in der Version 0.28. Dieses Projekt startete mit *React Native* 0.19 (veröffentlicht am 29. Januar 2016). *React Native* ist ganz neu und noch in der Entwicklungsphase. Momentan erscheint immer noch alle zwei Wochen ein neues Release, mit Änderungen, die teilweise sehr nützlich und wichtig sind. Es ist aber auch schon vorge-

[1] https://josm.openstreetmap.de/
[2] http://www.keepright.at/
[3] http://wiki.openstreetmap.org/wiki/Editors#Choice_of_editors/
[4] http://wiki.openstreetmap.org/wiki/MapRoulette/
[5] http://wiki.openstreetmap.org/wiki/Games#Gamification_of_map_contributions

kommen, dass Updates ausgelassen werden mussten, da die App nicht mehr lauffähig war. Build-Probleme sind des öfteren aufgetreten.

Die offizielle Dokumentation[6] ist immer noch spärlich und Best Practices gibt es in vielen Bereichen gar keine.

2.3. OpenStreetMap OAuth

Im Gegensatz zu den anderen Social-Login-Varianten (*Google* und *Facebook*), die OAuth 2.0 anbieten, unterstützt *OSM* noch OAuth 1.0a[7].

2.4. Kort Schnittstelle

Die KORT-Web-App nutzte eine Session-basierte (Cookie-based) Authentifizierung.[25] Eine Session-ID wird auf der Seite des Clients in einem Cookie gespeichert. Bei jedem Request sendet der Client-Browser das Cookie an den Server und wird so wiedererkannt. Der Server geht dann davon aus, dass es sich beim Client um den Inhaber der Session-ID handelt. KORT als native App kann aber keine Session zu einem Server aufbauen. Das KORT-Backend wurde von Stefan Oderbolz und Jürg Hunziker dann so angepasst, dass eine Token-basierte Authentifizierung unterstützt wird. Vorerst wurde nur eine Unterstützung für eine Login-Variante über *Google* umgesetzt.

[6] https://facebook.github.io/react-native/docs/getting-started.html
[7] http://wiki.openstreetmap.org/wiki/OAuth

3. Evaluation

An einem *React Native* Meetup, am 8. März 2016 — an der HSR, konnten erste Entscheidungen zur Software-Entwicklungsumgebung und welche Lösungskonzepte es für die Darstellung der Karte gibt, geklärt werden.

Unter den vielen *JavaScript*-Editoren haben wir uns für *Atom* entschieden. *Facebook* hat speziell für *React*, *React Native* das *Atom*-Package *Nuclide*[1] veröffentlicht. *Atom* ist Open Source und bietet viele weitere Community-Packages.

Weitere Hinweise zur Entwicklungsumgebung und den verwendeten Werkzeugen sind im Kapitel Projektmanagement beschrieben.

3.1. Architektur

Bei der Evaluation, welches Architektur-Pattern wir anwenden würden, konnten wir uns auf Frontend-Architekturen beschränken. Dabei kamen für uns zwei Patterns in Frage: Traditionelles MVC und *Flux*[2].

[1] https://nuclide.io/
[2] https://facebook.github.io/flux/

Variante A: MVC	
Vorteile	Nachteile
Unidirektionaler Datenfluss	Grösserer Aufwand für erstmalige Einrichtung
Kaum Kopplung – Anpassungen in späteren Arbeiten relativ problemlos möglich	Viele Callbacks nötig
Abstraktion des Applikationszustands passend für unseren Anwendungsfall	
Nutzt wichtige Konzepte von *React*	
Variante B: *Flux*	
Vorteile	Nachteile
Pattern ist bekannt, keine Einarbeitung nötig	

Tabelle 3.1.: Bewertung Navigations-Komponente

3.1.1. Fazit

Die Architektur-Entscheidung fiel zu Beginn sehr schwer da vorgängig noch nicht genügend Erfahrungen im Umgang mit *React* gesammelt werden konnten. Entsprechend unserem Projektplan und der Grundidee, erst Erfahrungen mit für uns neuen Technologien (*JavaScript*, *React* und *React Native*) zu sammeln, haben wir uns in einem ersten Ansatz dazu entschlossen, die ersten Schritte – das Darstellen der Missionen auf der Karte – mit dem MVC-Pattern umzusetzen. Welche Auswirkungen der Einsatz von *Flux* hätte, war zu Beginn zu wenig voraussehbar.
Als die Wahl der Architektur im Zusammenhang mit Meilenstein 5 neu evaluiert wurde, konnten wir leichte Vorteile in der *Flux*- gegenüber einer MVC-Architektur sehen. Einen guten Einblick in die Thematik verschafft der Artikel *Why we are doing MVC and FLUX wrong*[1]. Wir haben uns letztlich aufgrund der oben aufgelisteten Vorteile auf *Flux* festgelegt.

3.2. App Navigation

Um die Navigation von KORT zu implementieren gab es zwei Möglichkeiten. Die erste Variante war die Umsetzung mit der von *React Native* zur Verfügung gestellte Navigator-Komponente. Für eine Tab-Ansicht müsste aber eine weitere Library evaluiert werden.

Als weitere Variante für die Navigation gab es die React-Native-Router-Flux Library. Diese Komponente basiert auf dem Navigator von *React Native* und unterstützt durch die bereits integrierte Tab-Library eine Tab-Ansicht.

Variante A: *React Native* Navigator-Komponente	
Vorteile	Nachteile
Es gibt eine Unterstützung für *Android* und *iOS*.	Die Dokumentation zur korrekten Verwendung ist nicht vorhanden.
Flexibel und für einfache Use Cases gedacht	Die Tab-Ansicht wird nicht unterstützt.
Variante B: React-Native-Router-Flux[3]	
Vorteile	Nachteile
Alle Views (Scenes) sind an einem Ort deklariert. Es müssen keine Navigator-Objekte herumgereicht werden.	Stellt eine weitere Abhängigkeit an ein externes Projekt dar
Leicht erweiterbar und wartbar. Integrierte Tab-Navigation.	Der aktuelle Navigationszustand ist nicht genau definiert.
Ein Wechsel der View ist mit einem Funktionsaufruf von überall aus möglich. Daten lassen sich dabei einfach als Parameter weitergeben.	Die Hintergrundabläufe und der Lebenszyklus sind nicht erkennbar.

Tabelle 3.2.: Bewertung Navigations-Komponente

3.2.1. Fazit

Da *React Native* standardmässig keine Tab-Navigation anbietet, wurde React-Native-Router-Flux als Navigations-Variante evaluiert. Der erste Prototyp mit dieser Library erfüllte die Anforderungen. Alle Views sind an einem Ort im Code festgelegt und es lassen sich bequem weitere hinzufügen.

Während dem Verlauf vom Projekt sind dann aber vermehrt Fehler aufgetreten. Im Nachhinein wäre es sinnvoller gewesen, die Navigator-Komponente zu verwenden. Das Verhalten vom React-Native-Router-Flux war nicht voraussehbar.

3.3. Karten Darstellung

Für die Darstellung der Karte mit *React Native* sind die Varianten A bis E ausfindig gemacht worden. Diese Punkte beinhalten Libraries oder Ideen zur Umsetzung einer Kartenansicht für die App.

Variante A: React Native Map Komponente

Diese Variante wird standardmässig von *React Native* als MapView-Komponente zur Verfügung gestellt. Die Möglichkeiten zur Verwendung sind eingeschränkt, denn es lassen sich auf

Android nur normale Pins als Marker einsetzen./citereact-native-mapview-pin

Variante B: Extended React Native Map Komponente

Die Extended React Native Map Komponente[4] wird von *Facebook* anstelle der Standard-MapView-Komponente, empfohlen.

Variante C: Mapbox GL Library

Mapbox bietet mit dieser experimentellen *React Native*-Komponente (Mapbox GL Library[5]) eine weitere Lösung für *iOS* und *Android*.

Variante D: Portierung von Leaflet nach React

Für *React* gibt es eine Map-Komponente namens React-Leaflet[6]. Das liesse sich für *React Native* portieren. Schon in der KORT-Web-App wurde die *Leaflet*[7]-Library verwendet.

Variante E: Raster-Kacheln selbst darstellen

Die letzte mögliche Variante war, dass wir die benötigten Raster-Kacheln, die der Benutzer braucht, mit einer eigenen Implementation entsprechend laden und anzeigen.

[4] https://github.com/lelandrichardson/react-native-maps
[5] https://libraries.io/npm/react-native-mapbox-gl
[6] https://github.com/PaulLeCam/react-leaflet
[7] http://leafletjs.com/t

Variante A: *React Native* Map Komponente	
Vorteile	Nachteile
Komponente von *Facebook* - *React Native*	Pattern Fill ist nicht implementiert und auch nicht in Planung.[3] Dadurch lassen sich keine eigene Marker auf der Kartenansicht darstellen.
	Es lassen sich keine Map-Kacheln von einem beliebigen Service darstellen.
Variante B: Extended React Native Map Komponente	
Vorteile	Nachteile
Bietet alle benötigten Funktionen (eigene klickbare Marker platzieren und vieles mehr).	Nutzt die native Map API von *Apple iOS* und *Android* SDK. Ist fest mit *Apple* und *Google Maps* verbunden.
Wird von *Facebook* empfohlen.	Native Map APIs sind für ein *OSM*-Projekt aus moralischen Gründen unpassend.
Variante C: Mapbox GL Library	
Vorteile	Nachteile
Lässt sich mit Offline-Kacheln von *OSM2VectorTiles*[8] füttern (Vektor Kacheln).	Diese Library ist eine Experimentelle Komponente.[24]
Es können eigene Marker-Bilder eingesetzt werden.	
Variante D: Portierung von Leaflet nach React	
Vorteile	Nachteile
Dies wäre eine der einzigen Möglichkeiten, falls keine andere Variante in Frage kommt.	Die Darstellung ist nur im Browser möglich.
Variante E: Raster-Tiles selbst darstellen	
Vorteile	Nachteile
Dies wäre eine der einzigen Möglichkeiten, falls keine andere Variante in Frage kommt.	Bringt einen zu hohen Aufwand mit sich.

Tabelle 3.3.: Bewertung Map-Komponente

3.3.1. Fazit

Varianen, die Native Map APIs von *Google* und *Apple* verwenden, kamen für uns nicht in Frage. Wir möchten mit unserer App *OSM* Daten verbessern und möchten somit aus moralischen Aspekten auch auf diese Karte setzen.

Bei der *React Native* MapView-Komponente gab es keine Möglichkeit Bilder auf der Karte darzustellen und die Raster-Tiles "von Handänzuzeigen wäre schlicht zu aufwendig. Es liesse sich auch nur sehr umständlich eine schöne Map designen.

Somit sprang uns als erstes die Portierung von Leaflet für *React* ins Auge. Nach genauerem Betrachten fiel uns aber auf, dass diese Variante nur möglich ist, wenn die Karte in einer WebView-Komponente von *React Native* dargestellt wird. Das heisst, dass die App müsste zur Browser-Ansicht wechseln müsste.

Als letzte Möglichkeit blieb die Mapbox GL Library. Diese hat beim Testen auf Anhieb funktioniert und uns überzeugt. Die Kosten bei, einer Anzahl von 50 000 Nutzern pro Monat, sind für dieses Projekt nicht problematisch.[22]

Der Gedanken zur Offline-Unterstützung wurde auch besprochen. Nur unterstützt dies das Backend nicht und es gäbe Probleme mit dem Speicherplatzbedarf der App.

3.4. OAuth Implementation

Gebraucht wird ein Login-Dienst für *Google*-, *Facebook*- und *OSM*-Konten. Für einen *OSM*-Login muss eine eigene Lösung entwickelt werden. Für die Implementation der Authentifizierung wurden diese zwei Möglichkeiten evaluiert:

Variante A: Auth0

Auth0[9][10] bietet eine Implementation für beliebige OAuth 2-Dienste.

Variante B: Open-Source-Projekte

Ein geeignetes Open Source Projekt für die *Google*-Authentifizierung wäre react-native-google-signin[11]. Für *Facebook* bot sich react-native-facebook-login[12] an.

[9] https://github.com/auth0/react-native-lock
[10] https://auth0.com/
[11] https://github.com/devfd/react-native-google-signin
[12] https://github.com/magus/react-native-facebook-login

Variante A: *Auth0*	
Vorteile	Nachteile
Sehr einfach Einbindung	Nur OAuth 2 Unterstützung[2]
	Backend Anpassung nötig
Kostenlos für Open-Source-Projekte	Schlecht erweiterbar mit eigenem Login für *OSM*

Tabelle 3.4.: Bewertung OAuth-Komponente

3.4.1. Fazit

Auth0 kam definitiv nicht in Frage, da es in nächster Zukunft nicht mit einer OAuth 1.0a-Authentifizierung, wie sie von *OSM* unterstützt wird, erweiterbar ist. Entschieden haben wir uns für das react-native-google-signin-Projekt. Es funktioniert auf beiden Plattformen und liefert ein Token, das vom KORT-Backend überprüft werden kann. Der Nachteil ist, dass es kein Open Source Projekt gab, welches alle gewünschten Social-Logins für *iOS* und *Android* anbietet.

Facebook wird vom KORT-Backend derzeitig nicht unterstützt und das react-native-facebook-login-Projekt auf Github liefert auch kein Token, wie es beim *Google*-Login der Fall ist. Um dies zu behandeln wären weitere Anpassungen am Backend nötig gewesen. Wir hatten uns dazu entschieden, keine Backend-Anpassungen durchzuführen.

3.5. Internationalisierung

Die Übersetzungen, die für die Internationalisierung (I18n) der App zur Verfügung stehen, wurden in der KORT-Bachelorarbeit von 2012 mit *Transifex*[13] bereits erhoben.[25] Es stehen bereits 13 vollständig und weitere 13 teilweise übersetzte Sprachen zur Verfügung. *Transifex* ist eine Plattform, bei der Projekte zur freiwilligen Übersetzung von dessen Benutzern, aufgeschaltet werden können. Das heisst, dass sich ein Benutzer registriert und bei der Übersetzung von anderen Projekten beitragen kann. Die Übersetzungen sind allerdings im Java Properties Format, was eine Schwierigkeit darstellen würde.

Bei der Umsetzung der I18n müssen folgende Punkte umgesetzt werden: Auslesen des Locales aus den Geräteinformationen, entsprechendes Einsetzen der Übersetzungen und Einbinden der Übersetzungen, welche auf *Transifex* zur Verfügung stehen. Eine Library, die eine *React Native* Bridge zum Auslesen des Locales, die I18n-Funktionalität zur Verfügung stellt und `.properties` Dateien entgegennimmt, gibt es leider nicht.

Es boten sich folgende Alternativen an:

[13] https://www.transifex.com/

1. Auslesen des Locales mit einer Library für *React Native*[14] und Übersetzen anhand von Java Properties Files mit einer anderen Library[15]

2. Auslesen des Locales und Übersetzen mit einer Library für *React Native*[16] und manuelles Umwandeln der Java Properties Files in JSON

Die 1. Variante kam für uns nicht in Frage, da JQuery nicht ins Projekt eingebunden werden soll. So haben wir uns für die 2. Variante entschieden, vorerst aber nur Deutsch und Englisch ins richtige Format umgewandelt. Die weiteren Sprachen werden noch nach der Abgabe übernommen.

[14] https://github.com/fixd/react-native-locale
[15] https://github.com/jquery-i18n-properties/jquery-i18n-properties
[16] https://github.com/AlexanderZaytsev/react-native-i18n

4. Resultate

Wir konnten die wichtigsten Hauptziele erreichen. KORT erfüllt nun die Grundlage aller wichtigen Anforderungen an eine moderne App. Nach dem Login wird der Benutzer wie schon bei der KORT-Web-App zur Kartenansicht weitergeleitet. Er kann auf einen Marker klicken und gelangt direkt zur Ansicht, um die gewählte Mission zu lösen. Der Zwischenschritt, dass der Benutzer zuerst gefragt wird, ob er die Lösung kennt, wurde weggelassen. Somit entfällt ein weiterer Klick, was die Spielmechanik vereinfacht. Nur noch das Marker-Icon auf der Karte verrät etwas über den Missionstyp und weckt dabei immer noch die Neugier beim Benutzer.

4.1. Zielerreichung

Erreichte Ziele:

- *Android*-App mit gleicher Grundfunktionalität, wie die Web-App
- *iOS*-App mit gleicher Grundfunktionalität, wie die Web-App
  - Nicht getestet
- neuer Validationsmechanismus
- Erfahrungsbericht zu *React-Native*
- Internationalisierung umgesetzt

Das GUI-Design ist noch nicht ansprechend und erfüllt nur die minimalen Ansprüche. Für eine finale Version, die veröffentlicht werden kann, muss das Design aufgebessert werden.

Social-Login wurde nur für *Google* realisiert.

Die Badges, welche vom Backend empfangen werden, konnten durch den neuen Validationsmechanismus nicht mehr verwendet werden. Das Backend macht nämlich immer noch die Unterscheidung zwischen Missionen und Validationen.

Leider hat es zeitlich auch nicht mehr gereicht, um die Kartenansicht beim Lösen einer Mission anzuzeigen.

Die Veröffentlichung im *Google-Play* Store ist im Meilenstein 9 aufgeführt.

Wie diese zusätzlichen Arbeiten umgesetzt werden könnten wurde im Kapitel Vorgehen dokumentiert.

4.2. Ausblick

Offene Punkte und nächste geplante Arbeiten mit höherer Priorität:

- *OSM*-Login
- Finale *iOS*- und *Android*-App
- Veröffentlichung im *Apple App Store* und *Google Play Store*
- Promotions-Funktion

Das genaue Vorgehen, wie die offenen Punkte umgesetzt werden, wird im Kapitel Vorgehen beschrieben. Für die Zukunft gibt es bereits viele weitere Ideen. Eine Liste wurde im Kapitel Weiterentwicklung erstellt.

4.3. Persönliche Berichte

Dominic Mülhaupt

Marino Melchiori

Beim Start von diesem Projekt hatte ich keine *JavaScript*-Vorkenntnisse. Im Verlauf der Arbeit gelang es mir aber, dank der Zusammenarbeit in unserem Team, mich gut einzuarbeiten. Das Highlight dieser Arbeit war für mich die Implementation der App mit *React Native* und die Gestaltung der Benutzeroberfläche mit *JSX*. Trotzt all den neuen und teilweise unreifen Technologien, ist es uns gelungen, eine gute App-Idee neu zu entwickeln. Ich bin froh, dass ich *JavaScript*-Erfahrungen sammeln konnte.

Teil II.

Projektdokumentation

5. Anforderungsspezifikation

5.1. Anforderungen an die Arbeit

Die Autoren hatten im Vorfeld der Arbeit wenige Kenntnisse über Webtechnologien und insbesondere gar keine Erfahrung mit *JavaScript*. Insofern war der machbare Umfang des Projekts schwer absehbar. In Abstimmung mit dem Betreuer und dem Projektpartner wurde deshalb festgelegt, dass der Fokus auf dem Frontend liegt, so dass man sich nicht auch noch in die Technologien des Backends einarbeiten muss.

In der Anforderungsanalyse sind viele Aufgaben erkannt worden, die in diesem Projekt bearbeitet werden könnten. Im Rahmen dieser Arbeit ist nur ein Bruchteil davon umsetzbar. Zum Einen aus zeitlichen Gründen und zum Anderen, weil Anpassungen am Backend nötig wären. All diese Anforderungen sind in *Muss*, *Soll*, *Kann*, *zukünftige Arbeiten* und *abgewiesene Arbeiten* unterteilt.

5.1.1. Muss

- neuer Kort-Client als native *Android*-App
- Ablösung des Validationsmechanismus
- Erfahrungsbericht mit Hinweisen zu Tutorials zu React Native
- die vom Studiengang geforderten Lieferobjekte: Dokumentation, Management Summary, Abstract, Poster, Präsentation mit Stellwand, Zwischenpräsentation, Schlusspräsentation

5.1.2. Soll

- Kort-Client als native *iOS*-App

5.1.3. Kann

- neue Funktion: Promotions anzeigen
- Kurzvideo (zur Instruktion und Promotion)

5.2. Use Cases

Alle Use Cases für die KORT-App sind im Kapitel 4.1.1. User Szenarien der Bachelorarbeit von Jürg Hunziker und Stefan Oderbolz beschrieben und entsprechen weiterhin den Anforderungen. Es wurden folgende vier Szenarien beschrieben[25]:

- Szenario 1: Zeitvertrieb an der Bushaltestelle
- Szenario 2: Validieren
- Szenario 3: Erster Kontakt zur App
- Szenario 4: Highscore-Anwärter

6. Technologien

In diesem Kapitel sind Informationen zur Funktionsweise der Technologien *React* und *React Native* dokumentiert. Das Unterkapitel zu *React Native* beschreibt ebenfalls unsere Erfahrungen und Schlüsse.

6.1. React

React[1] wurde im März 2013 veröffentlicht[35], ist eine Open Source *JavaScript* Library und dient für die Implementation der View vom MVC-Pattern. Die View besteht aus wiederverwendbaren Komponenten, die wiederum Komponenten beinhalten. *React* wird von *Facebook*, *Instagram* und von der Community entwickelt und gewartet.[36]

Für *React* wird JSX, welches eine HTML ähnliche Syntax nutzt, zur Erstellung der Komponenten empfohlen. So lassen sich Komponenten-Bäume direkt mit *JavaScript* erstellen. Anders formuliert können *JavaScript*-Objekte mit einer HTML-Syntax erzeugt werden. Eine Hauptkomponente gibt seine Daten per Props an die Kind-Komponenten weiter (one-way-dataflow).[10] JSX wird nicht zwingend benötigt.[7]

Anstatt der DOM nutzt *React* die sogenannte Virtual DOM. Wie der Begriff schon sagt, wird mit einer Abstraktion der echten DOM – also mit einer virtuellen DOM – kommuniziert. Die komplette DOM, also die Repräsentation der View vom HTML-Code, ist im lokalen Speicher abgelegt.[21] In der `render()`-Methode jeder *React*-Klasse wird eine Beschreibung der DOM zurückgeliefert, die *React* mit der lokalen Kopie der DOM vergleicht. Mit einem sehr effizienten Diffing-Algorithmus berechnet *React* den Unterschied zwischen diesen Versionen der DOM und errechnet den schnellsten Weg um den Browser zu aktualisieren.[16]

6.2. React Native

Der Ansatz von *React Native*[2] ist *learn once – write anywhere*, das heisst, lerne eine Technologie und nutze sie für alle unterstützten Plattformen.[15]

Am 26. März 2015 wurde *React Native* erstmals für *iOS* veröffentlicht. Im Oktober 2015 kam *Android* dazu.[14] Seit dem Release gibt es alle zwei Wochen eine neue Version. Durch diese häufigen Änderungen konnten sich noch keine Best Practices etablieren. Auch die meisten

[1] https://facebook.github.io/react/, https://github.com/facebook/react
[2] https://facebook.github.io/react-native/, https://github.com/facebook/react-native

Open Source Projekte verfolgen eigene Implementationsansätze.

Eine Desktop Unterstützung für *OSX* ist ebenfalls in Entwicklung[3]. Und am 13. April 2016, an der *Facebook* Developer Konferenz, kündigten *Microsoft* und *Facebook* den Support für die *Universal Windows Platform* (*UWP*) an.[23]

6.2.1. Layout

Alle GUI-Komponenten befinden sich in sogenannten Containern. Ein Container wird durch eine `View`-Komponente definiert. Das Layout und die Gestaltung der Container und Komponenten wird mit *Flexbox* geregelt.

6.2.2. Technische Details

React Native nutzt einen *JavaScript*-Layer, beziehungsweise *JavaScriptCore* als Engine, um den Code auszuführen.[8] Die native Funktionen werden auf die *JavaScript*-Objekte oder Funktionen gemappt. Das Endprodukt ist also keine Web-App für den Browser und wird auch nicht in native Code kompiliert. Ausserdem wird der *JavaScript*-Code auf einem separaten Thread ausgeführt und nicht auf dem UI-Thread. Dadurch wirken zum Beispiel die Animationen sehr flüssig.[13]

Native Module

Damit ein *iOS* Native-Modul in *React Native* verwendet werden kann, muss das konkrete Modul das
`RCTBridgeModule`-Protokoll[4] implementieren und das Makro
`RCT_EXPORT_MODULE()` enthalten. Das Protokoll dient nur dazu, das Modul in einem Array zu speichern, damit es später von der Bridge gefunden werden kann. Wenn die *JavaScript*-Seite der Bridge initialisiert ist, kann sie auf diese Daten zugreifen. Dem Makro kann auf der *JavaScript*-Seite ein optionaler Namen als Parameter mitgegeben werden. Falls dieser Parameter fehlt, wird die Komponente auf *JavaScript*-Seite nach dem *Objective-C*-Klassennamen benannt. Ein Ähnliches Vorgehen gilt für *Swift*- und *Android*-Module. Genauere Hinweise sind in der *React Native*-Dokumentation unter *Native Modules* beschrieben.[12][11] Mit diesem Feature lässt sich bereits vorhandenen native Code wiederverwenden.

6.2.3. Setup

Die Entwicklungsumgebung lässt sich am schnellsten und einfachsten auf *OS X* einrichten. *Windows* und *Linux* sind mittlerweile ebenfalls geeignet, was am Anfang dieser Bachelorarbeit nicht der Fall war.

Mit dem *React Native*-CLI (Command Line Interface) können neue Projekte initialisiert

[3] https://github.com/ptmt/react-native-desktop
[4] RCT ist eine Abkürzung für ReaCT

werden und Projekte auf einem Emulator oder einem Gerät getestet werden. Nur die *iOS*-App muss über *Xcode* erzeugt werden.

Die Projektstruktur vom *JavaScript* Code hat vor allem die verwendete Architektur vorgegeben. Komponenten der Architektur sind in die entsprechenden Hauptbestandteile eingeordnet. Konstanten, IDs und Access-Tokens befinden sich im `Constants`-Ordner. Alle GUI-Komponenten wurden im `Components`-Ordner erstellt.

6.2.4. Community

Die Community wirkt sehr zerstreut, denn viele Informationen sind in den Issues vom *React Native* GitHub-Repository[5] versteckt. Ausserdem sind aktuell, am 15.06.2016, 733 offene Issues vorhanden.

- Vorhanden ist eine öffentliche, aktive und hilfsbereite *Facebook*-Gruppe[6], mit derzeit ca. 3 500 Mitgliedern.
- Es gibt eine *Stack-Overflow*-Kategorie[7] - leider nur mit wenigen Antworten und Lösungen.
- JS.coach[8] - listet viele Open Source Projekte auf.
- Übersicht über aktuelle Artikel und Blogposts:
  - reactnative.com[9]
  - *React Native* Newsletter[10]
- Es ist ebenfalls ein aktiver Subreddit[11] vorhanden.

6.2.5. Zusammenfassung

Eine Stärke von *React Native* ist die Plattformunabhängigkeit. Wenn keine spezifische *Android-* oder *iOS*-Komponenten verwendet werden, kann der Code für beide Plattformen genutzt werden.

Die grössten Hürden von *React Native* sind das Erlernen von*React* und *Flexbox*. Denn *Flexbox* für *React Native* unterscheidet sich in vielen Details vom *Flexbox* für Webseiten. Einerseits ist das Grundkonzept für das Layout bei vielen verschachtelten Komponenten schwer zu verstehen. Selbst Die Umsetzung von ganz simplen Views ist am Anfang schwer und frustrie-

[5] https://github.com/facebook/react-native/issues
[6] https://www.facebook.com/groups/react.native.community/
[7] http://stackoverflow.com/questions/tagged/react-native
[8] https://js.coach/
[9] http://www.reactnative.com/
[10] http://reactnative.cc/
[11] https://www.reddit.com/r/reactnative

rend. Andererseits wirkt es, nachdem viele Stunden in das Lernen investiert wurden, doch konsistent und praktisch. Dazu kommt, dass durch *Flexbox* die definierten Styles gut wiederverwendbar sind. Das Gleiche gilt für jegliche Komponenten, die auch gut durch Inspiration aus anderen Open Source Projekten erstellt werden können. Durch die wachsende Community gibt es immer mehr solcher Projekte, Pull-Requests und Beiträge an *React Native* selber. Das macht *React Native*, langfristig gesehen, zu einer immer besseren und wichtigeren Technologie.

Die Umstellung von der native Entwicklung zu *React Native* ist Anfangs auch schwer. Vor Allem wenn Fehler auf *React Native*-Seite existieren, die native elegant und knapp lösbar sind. Mit der Zeit zeigen sich aber die grossen Vorteile von *React-JavaScript*. Jede einzelne Komponente wird allein durch ihren State kontrolliert. Je nach State kann die Komponente durch die eigene `render()`-Methode kontrolliert und verändert werden.

Ein ausschlaggebender Vorteil ist das Live- oder Hot-Reload Feature. Nur beim ersten Ausführen der app muss ein Build erstellt werden. Dank dem gleichzeitigen Starten vom Packager kann die Ansicht auf dem Emulator oder dem Smartphone, direkt während dem Programmieren, aktualisiert werden. Die Hot-Reload Funktion erlaubt ein manuelles Aktualisieren der App.

Der App-Showcase, der offiziellen *React Native*-Dokumentation, wächst stetig.[12].[6] Nach uns werden Apps in Zukunft nur noch mit *React Native* entwickelt.

[12] https://facebook.github.io/react-native/showcase.html

7. Design

7.1. Datenmodell

Die Model Logik ist weitestgehend im Backend implementiert, welches die Daten über eine REST-Schnittstelle zur Verfügung stellt. Die über die Schnittstelle zur Verfügung gestellten Informationen sind also Ausgangspunkt für die von uns verwendbaren Domain Klassen. Es waren kleinere Anpassungen nötig, etwa um den neuen Validationsmechanismus zu unterstützen. Da die empfangenen Daten fast ausschliesslich als Repräsentation von Informationen dienen und keine eigene Logik implementieren, wurden sie als Datentransferobjekte umgesetzt.
Das folgende Datenmodell repräsentiert die Datentransferobjekte und ist nicht als semantisches Modell zu verstehen.

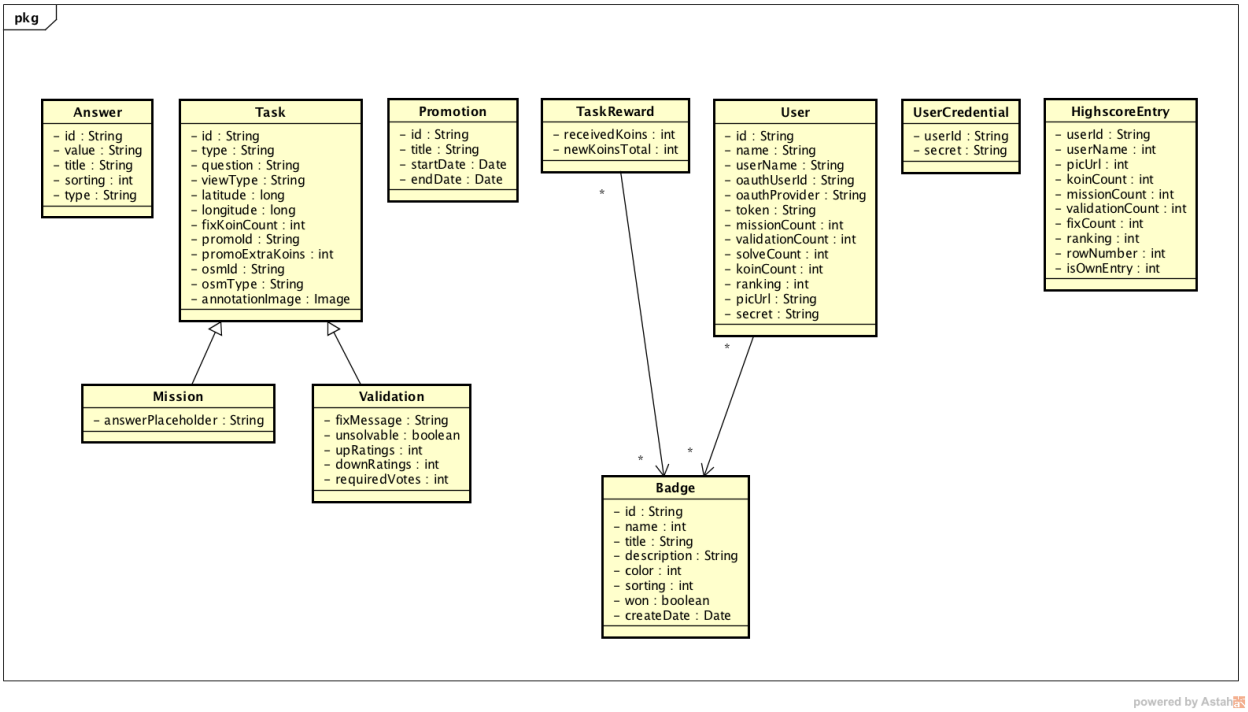

Abbildung 7.1.: Modellierung der DTOs

7.2. Architektur

Für unsere Applikationsarchitektur haben wir das *Flux*[1] Architekturpattern von *Facebook* eingesetzt.
Die folgenden Erklärungen wurden zu grossen Teilen aus der *Flux* Dokumentation[5] abgeleitet. Zu den nachfolgend beschriebenen Konzepten der *Flux*-Architektur mussten für unsere Realisierung keine Anpassungen gemacht werden. Die Beschreibungen entsprechen also unserer Implementation.
Flux ist eine Architektur, die in Frontend-Applikationen eingesetzt wird. Ein unidirektionaler Datenfluss ist das Grundprinzip, auf welchem *Flux* aufbaut, und in welchem es sich von MVC-Architekturen unterscheidet. Das Pattern beschreibt folgende vier Hauptbestandteile: die Stores, die Views (oder Controller-Views), die Actions und der Dispatcher.

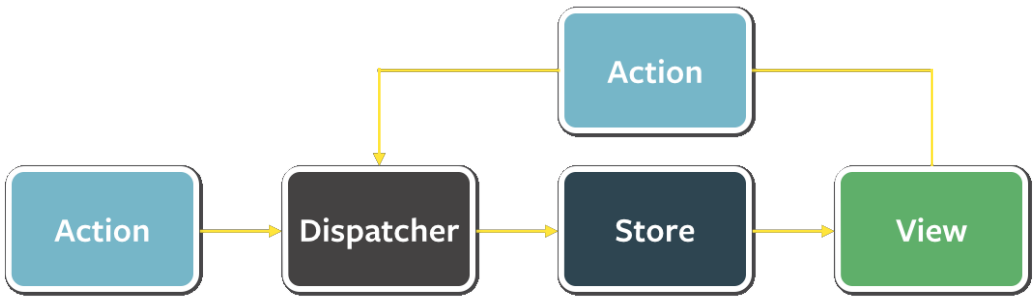

Abbildung 7.2.: Idee der *Flux*-Architektur

7.2.1. Stores

Stores enthalten den Applikationszustand und die Applikationslogik. Verglichen mit dem MVC-Pattern entsprechen sie am ehesten dem Model. Sie unterscheiden sich aber insofern, dass sie nicht unbedingt einzelne Modellklassen repräsentieren sollen, sondern domänenspezifische Aufgaben übernehmen. Da KORT keine besonders komplexe Domäne enthält, ist diese Unterscheidung allerdings nicht von grosser Bedeutung. Ein Beispiel für diese Einteilung nach Domäne und nicht nach Modell findet man in der Aufteilung von `UserStore` und `AuthenticationStore`
Stores sind als Singletons umgesetzt. Sie registrieren sich beim Dispatcher um Updates zu erhalten und ihren Zustand entsprechend anzupassen. Views wiederum können sich bei den Stores registrieren um neue Informationen zu erhalten, können die Stores aber nicht direkt anweisen, sich zu aktualisieren.

[1] https://facebook.github.io/flux/

7.2.2. Views

Der Applikationsanwender kommuniziert seine Absichten über die View. Deshalb gilt die View in *Flux* als Auslöser für neue Aktionen. In *React* kann zwischen *Views* und *Controller-Views* unterschieden werden.
Controller-Views finden sich an der Spitze der View-Hierarchie. Sie warten auf Updates der Stores und reichen die Daten entlang der Kette ihrer untergeordneten *Views* weiter.
Diese untergeordneten *Views* wiederum reagieren auf Zustandsänderungen indem ihre `render()` Methode neu aufgerufen wird. Somit bleiben *View* Komponenten modular austauschbar, da sie unabhängig von ihrem Kontext eingesetzt werden können.

7.2.3. Actions

Actions sind Helfermethoden, welche im Dispatcher ein Ereignis und somit in den Stores ein Update auslösen. Sie werden ausschliesslich durch Views ausgelöst, da diese für den Kontrollmechanismus der Applikation zuständig sind. Ausnahmen wären hier denkbar, waren aber nicht nötig. Beispielsweise könnte der `LocationStore` eine Action auslösen wenn er eine neue Position erkannt hat oder der Server wenn ein Update an die Applikation gesendet werden soll.
Ausserdem sollten Daten, wenn diese für ein Update nötig sind, bereits durch die Actions an den Dispatcher mitgeliefert werden. Aufrufe an die REST-Schnittstelle des Backends werden also über die Actions ausgelöst. Somit kann sichergestellt werden, dass verschiedene Stores, welche auf dieselbe Action reagieren, denselben API-Aufruf mehrmals ausführen. In Abbildung 7.3 ist dies ersichtlich.

7.2.4. Dispatcher

Der Dispatcher ist der zentrale Knotenpunkt, durch den der gesamte Datenfluss der Applikation koordiniert wird. Seine einzige Aufgabe ist die Verteilung der Actions an die Stores. Er enthält also keine intelligente Logik, sondern ist grundsätzlich ein Register von Callbacks.

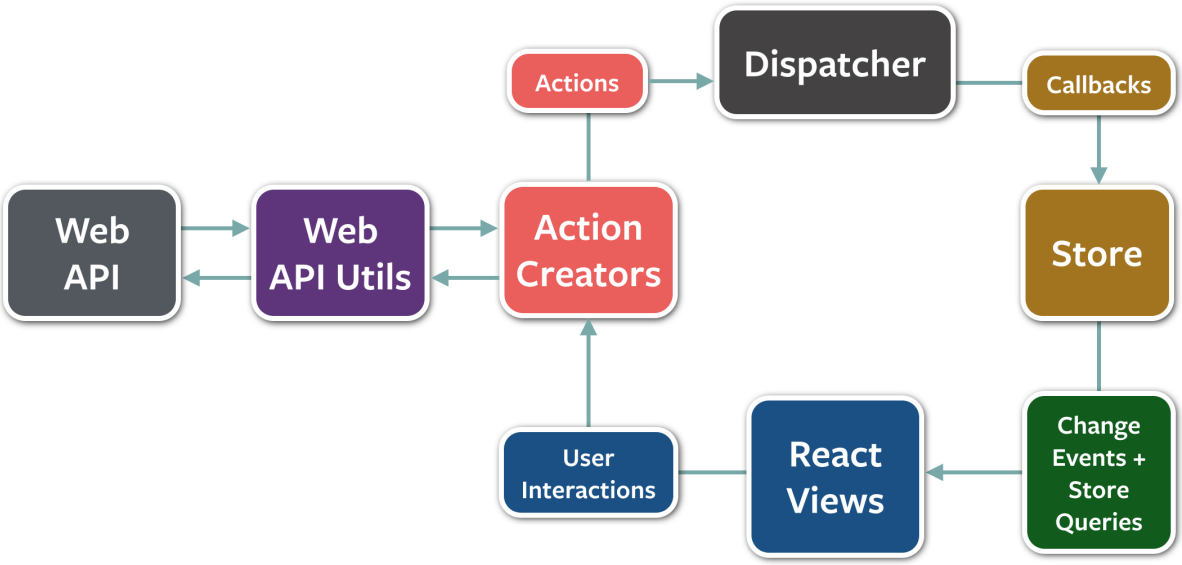

Abbildung 7.3.: Vollständiges *Flux*-Diagramm

7.3. Klassenkonzepte

Die Klassen können folgendermassen strukturiert werden: Actions, Components, Data, Dispatcher, DTO, Stores.

Actions werden von den Components benötigt, um Aktionen in den Stores auszulösen, Daten vom Backend zu laden oder Daten an das Backend zu senden.

Components entsprechen den Views der *Flux*-Architektur. Component Klassen erweitern `React.Component`[2] und implementieren mindestens die `render()` Methode[3]. In der `render()` Methode wird in JSX Syntax definiert, wie die Component dargestellt werden soll.
Eine Component kann andere Components wiederverwenden, Dadurch wird sowohl ein hierarchischer Aufbau der View, als auch ein modularer Einsatz von Components ermöglicht.
Ein weiteres Konzept, das von Components realisiert wird, ist die Unterscheidung von Property und State. Eine Property repräsentiert eine sich nicht ändernde Eigenschaft einer Component, während der State den aktuellen Zustand ausdrückt. Properties werden vom Owner[4] gesetzt und können durch den Ownee als `propTypes` deklariert werden. Wird eine State Variable neu gesetzt, führt dies zu einem erneuten Aufruf der `render()` Methode.
Components dürfen den Zustand – der in den Stores gehalten wird – nie direkt manipulieren, sondern müssen dies über Actions auslösen. In den Callbacks, welche sie bei den Stores

[2] https://facebook.github.io/react/docs/component-api.html
[3] https://facebook.github.io/react/docs/component-specs.html
[4] Die Owner-Ownee-Beziehung wird unter *Ownership* beschrieben: https://facebook.github.io/react/docs/multiple-components.html

registriert haben, können sie dann die Getter aufrufen um den neuen Zustand zu erhalten. Components sollen nicht mehr über den Applikationszustand wissen als für ihre Ansicht nötig ist. Die Logik sollte also so weit möglich in den Stores umgesetzt sein.

7.4. Sequenzdiagramme

7.4.1. Starten der App

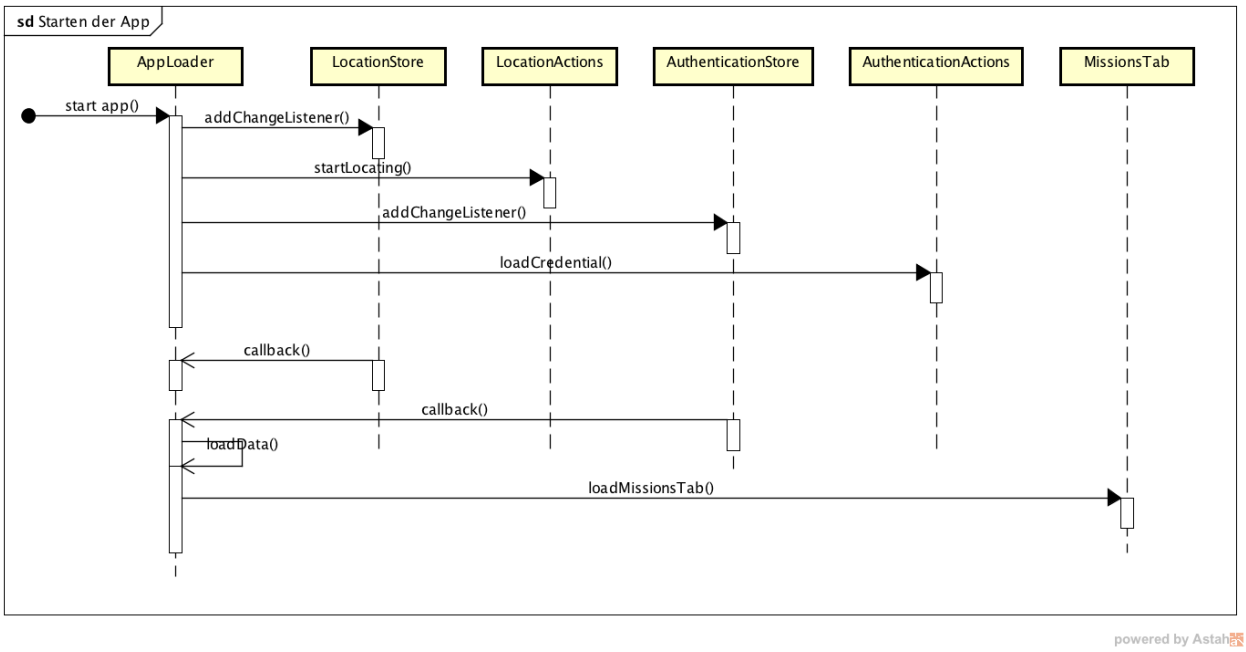

Abbildung 7.4.: Bevor Daten geladen und Informationen dargestellt werden können, muss der Benutzer autorisiert und lokalisiert sein.

7.4.2. Mission lösen

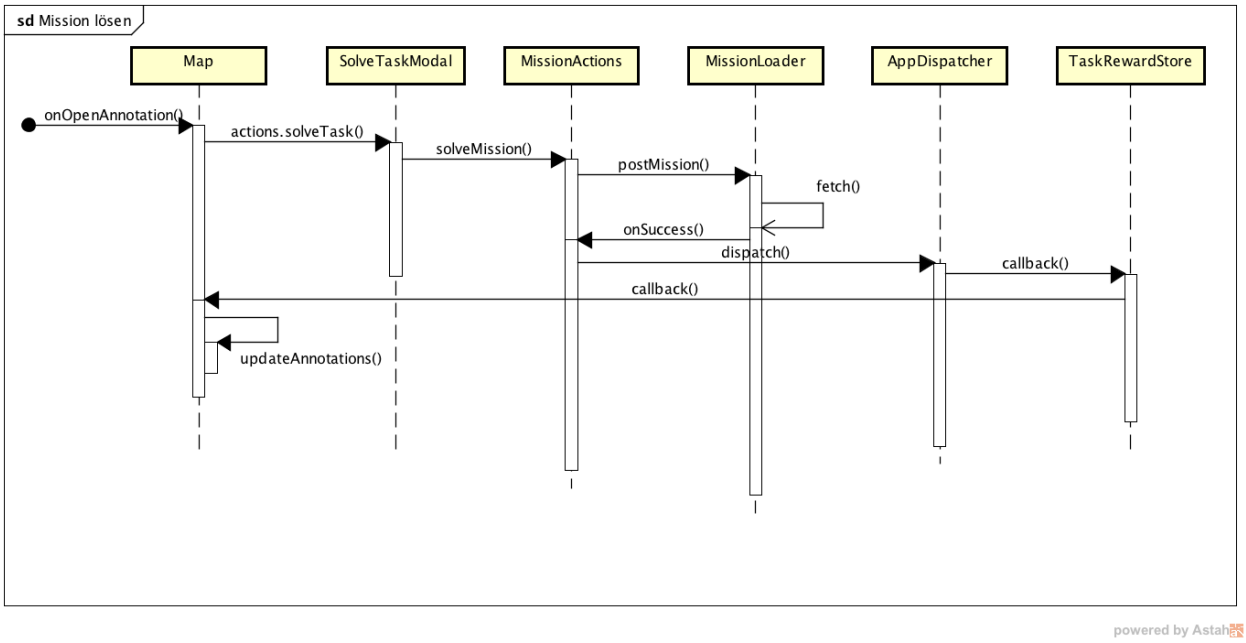

Abbildung 7.5.: Lösen einer Mission und Anzeigen der Belohnung

8. Entwicklungsumgebung

8.1. IDE

Die Funktionalitäten und Features der App wurden alle mit *Atom* implementiert. Für das Debugging waren die native IDEs, *Android Studio* und *Xcode*, aber besser geeignet. Das Debuggen in er *Atom*-IDE oder im *Google Chrome* Browser war oft fehlerhaft. Ansonsten wurden die native IDEs nur für das Einfügen von statischen Bildern, mit passenden Auflösungen für entsprechende Displays, genutzt.

8.2. Continuous Integration

Als Versionsverwaltungssystem wurde *git*[1] zusammen mit dem Online-Dienst *GitHub*[2] verwendet: https://github.com/kort/kort-reloaded. Dabei haben wir folgendes Branching Model eingesetzt: Der master Branch wird nur für Releases verwendet. Für die Entwicklung wurde der develop Branch genutzt, wobei jedes Feature und jeder Fix eines Bugs einen eigenen Branch erhielt, welcher nach Fertigstellung wieder in den develop Branch gemerged wurde.

Für die Continuous Integration (Continuous Integration) nutzen wir den freien Dienst von *Travis-CI*[3]. Dieses Setup lässt sich bequem in Verbindung mit *GitHub* nutzen. Dabei wird bei jeder Neuerung auf dem master und develop Branch über *Travis-CI* ein neuer Build erstellt. Bei jedem Build werden die Tests durchlaufen und der Code auf die Einhaltung der Code-Richtlinien überprüft.
Die Konfigurationsdatei für *Travis-CI* (`.travis.yml`) befindet sich im *GitHub*-Repository von Kort.

8.3. Projektmanagement-Tool

Als Projektmanagement-Tool wurde *Redmine* verwendet. Weiterführende Links zum Redmine-Projekt, das für die Ticketerfassung verwendet wurde, sind im Kapitel Projektmanagement dokumentiert.

[1] https://git-scm.com/
[2] https://github.com
[3] https://travis-ci.org/

8.4. Testing

Fürs Testing wurden Unit Tests, Integration Tests und Funktionalen Tests evaluiert. Letztlich konnten – zum Zeitpunkt der Abgabe – lediglich die Unit Tests umgesetzt werden. Integration Tests wären für die `data` Klassen wünschenswert gewesen, konnten aber aus Zeitgründen nicht umgesetzt werden. Automated UI Tests sind mit *React Native* möglich[4], sind aber sehr aufwendig einzurichten. Für die Unit Tests wurde *Jest*[5] eingesetzt. *Jest* ist ein *JavaScript* Testing-Framework und wird zum Beispiel von *Facebook* zum Testen von *React*-Applikationen verwendet.
Eine besondere Eigenschaft von *Jest* ist, dass standardmässig für alle Module automatisch Mocks bereitgestellt werden. Somit wird verhindert, dass aus Versehen das Verhalten anderer Module getestet wird.

8.5. Code-Richtlinien

Um Code-Richtlinien festzulegen und deren Einhaltung zu prüfen, wurde *ESLint*[6] eingesetzt. Dadurch wird die die Lesbarkeit und Wartbarkeit vom Code erhöht. Die Konfiguration von *ESLint* stammt von *Airbnb*[7]. Darüber hinaus wurden Plugins für *React*[8] und *React Native*[9] eingesetzt.
Die Konfiguration findet sich in der Datei `.eslintrc.json`.

[4] Einen guten Überblick bietet diese Seite: http://testdroid.com/tech/testing-react-native-apps-on-android-and-ios
[5] https://facebook.github.io/jest/
[6] http://eslint.org/
[7] https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb
[8] https://github.com/yannickcr/eslint-plugin-react
[9] https://github.com/Intellicode/eslint-plugin-react-native

9. Implementation

Dieses Kapitel beschreibt, wie die folgenden Punkte eingerichtet und implementiert wurden. Beim Entwickeln der *Android*-App wurde darauf geachtet, dass möglichst keine Plattform spezifische Komponenten zu nutzen. Das konnte in diesem Projekt erfolgreich umgesetzt werden. Die einzige spezifische Komponente ist der Ladebildschirm, der jeweils auch für die *iOS*-Version erstellt wurde. Aus Zeitgründen konnte die *iOS*-Version aber nicht getestet werden.

9.1. Kort Backend

Das KORT-Backend existierte bereits und wurde von den Entwicklern nicht mehr abgeändert. Es ist im Kapitel 10.2. REST-Schnittstellen der Bachelorarbeit von Jürg Hunziker und Stefan Oderbolz dokumentiert.[25]

9.2. Libraries

Damit die entsprechenden Libraries genutzt werden können, mussten sie mit dem Node-Package-Manager (npm) heruntergeladen werden. Mit dem `npm install`-Befehl wurden alle aufgelisteten Abhängigkeiten in der `package.json`-Datei installiert und im Ordner `node-modules` gespeichert. Das Linken der Libraries fand dann im *Android*- und *iOS*-Projekt statt. Dann konnten die Libraries in einer *JavaScript*-Komponente importiert und verwendet werden.

9.3. Navigation

Die Implementation der Navigation wurde in der Einstiegs-Komponente der App umgesetzt. Alle Scenes werden mit einem Key deklariert und falls nötig mit Optionen für eine Tab-Ansicht erweitert. Die definierten Scenes werden als Kind-Komponenten einem Router übergeben. Jede Scene kann mit einem Funktionsaufruf ...

- ... die Properties aktualisieren
- ... sich selber schliessen
- ... eine andere Scene mit dem Key und mit optionalen Properties, als Parameter, aufrufen

Die API der Navigations-Komponente ist auf der entsprechenden *Github*-Seite[1] dokumentiert.

9.4. Karte

Um die Map-Komponente von *Mapbox* zu nutzen und ein Access-Token zu erhalten, muss auf der *Mapbox*-Webseite[2] ein Benutzerkonto angelegt werden.

Damit die Marker an richtiger Position angezeigt werden, wurden der *Mapbox*-Komponente ein Array mit Annotations als Parameter übergeben. Das Annotation-Array wird fortlaufend mit allen Missionen vom Backend, in der Umgebung des Benutzers, aktualisiert. Eine Annotation enthält die Koordinaten und die ID einer Mission. Es war von der *Mapbox*-API her nicht möglich, einer Annotation direkt die Mission als Objekt mitzugeben. Bei einem Klick auf einen Marker wird die Funktion `onOpenAnnotation` aufgerufen. Diese Funktion öffnet wiederum eine Scene, die es dem Benutzer erlaubt, die gewählte Mission zu lösen. Die *Mapbox*-API ist auf der entsprechenden *Github*-Seite[3] dokumentiert.

9.5. OAuth

Für die Authentifizierung über *Google* wurde eine Open Source Library evaluiert (Kapitel Evaluation **??**). Damit die App sich mit der *Google*-API verbinden kann, muss eine `google-services.json`-Datei auf der *Google*-Webseite[4] generiert werden. Diese Datei enthält die Konfiguration vom *Google*-Developer Konto.

Über diese Library enthält die App, nach dem Login des Benutzers, das Benutzer-Token von *Google*. Dieses Token wird dann von der App dem KORT-Backend mitgeteilt und von dort aus bei *Google* überprüft. Wie die Authentifizierung auf der Backend-Seite weiter verläuft ist im Kapitel 10.4.1. der Bachelorarbeit von Jürg Hunziker und Stefan Oderbolz beschrieben.[25]

9.6. Internationalisierung

[1] https://github.com/aksonov/react-native-router-flux
[2] https://www.mapbox.com/
[3] https://github.com/mapbox/react-native-mapbox-gl
[4] https://developers.google.com/cloud-messaging/android/client

10. Weiterentwicklung

KORT hat ein grosses Potenzial um weiterentwickelt zu werden. Dieses umfasst vor allem zwei Bereiche:

Wie kann KORT besser dazu beitragen, *OSM* Daten zu verbessern?

Und wie kann der Benutzer mithilfe von Konzepten der Gamification weiter motiviert werden, zur Datenpflegung beizutragen?

Wir haben uns Gedanken dazu gemacht und hier zusammengefasst, wie KORT weiter optimiert werden könnte. Das Unterkapitel Vorgehen enthält Arbeiten, die beim derzeitigen Stand noch verbessert werden müssen. Weiterhin sind Ideen aufgelistet, die keine grösseren Änderungen am Backend mit sich bringen.

10.1. Vorgehen

Ein sehr wichtiges Feature, das bisher noch nicht umsetzbar war, ist der *OSM*-Login. Neben den Änderungen am Backend, damit dieses Feature überhaupt brauchbar ist, gibt es folgende Schritte zu erledigen:

1. App bei *OSM* mit Callback-URL registrieren
   - Als Callback-URL könnte zum Beispiel 'http://www.kort.ch/' verwendet werden.
2. *OSM*-Request-Token-URL aufrufen, um das OAuth-Token und das Token-Secret zu erhalten
3. *OSM*-Authorize-URL mit dem OAuth-Token aufrufen, damit sich der Benutzer auf der *OSM*-Seite einloggen kann
4. WebView-Komponente öffnen, um auf die Callback-URL zu warten
5. Wenn sich der Benutzer eingeloggt hat, wird er zur Callback-Seite, mit dem OAuth-Token und dem OAuth-Verifier in der URL, umgeleitet.
6. WebView schliessen
7. Request *OSM*-Access-Token-URL
8. dem Backend das Token zur Überprüfung senden

Dieses Vorgehen wurde noch nicht getestet und dient nur als Idee zur möglichen Umsetzung. Die korrekten Request-URLs können aus der Dokumentation im *OSM*-Wiki[1] entnommen werden.

Als nächstes müsste für den Gamification-Ansatz das Design weiter verbessert werden. Dafür wäre zuerst eine Änderung am Backend geplant, um die Validationen endgültig abzuschaffen, indem sie als normale Missionen gezählt werden. Dann wäre es auch wieder möglich korrekte Badges für den Missionen-Counter anzuzeigen.

Durch den Einsatz von Farben oder einem Hintergrundbild beim Login-Screen würde das Design einen Spieler besser ansprechen. Passend dazu könnten die verwendeten Bilder, Icons und Marker einheitlich gestaltet und erneuert werden.

Meldungen, die dem Benutzer die Anzahl gewonnener Koins anzeigen, erscheinen momentan im Vollbildmodus. Das liegt an einem Fehler der eingesetzten Navigations-Komponente, die keine transparenten Hintergründe zulässt. In Zukunft könnte das aber noch behoben werden. Dann wäre es eleganter, Meldungen in einem Fenster zu zeigen.

GUI-Arbeiten

- Map-Marker ersetzen
- Textfeld, um den Benutzernamen anzupassen, anbieten
-

Um die Funktionalität der Web-App komplett umgesetzt zu haben fehlt nur noch die zweite Highscore-Ansicht und die Karte, die beim Lösen einer Mission angezeigt wird. Die zweite Highscore-Ansicht zeigt den eingeloggten Benutzer, mit seinen direkten Konkurrenten vor und nach ihm. Wenn sich der Benutzer in der Ansicht zum Lösen einer Mission befindet, gäbe es in einem weiteren Tab eine Karte, die das Objekt der Mission markiert hervorgehoben anzeigt.

Gerne hätten wir noch Tests der Komponenten durchgeführt, doch die Priorität dafür war zu tief und die Zeit zu knapp. Dazu haben wir für die Zukunft die Enzyme[2]-Testing-API (für Komponenten-Tests) und das Mocha[3]-Framework (um die Tests laufen zu lassen) evaluiert.

10.2. Realistische Arbeiten

Punkte, die Kort attraktiver machen würden:

- Benutzerlogin mit weiteren OAuth-Diensten erweitern (z.B. *Twitter*, *GitHub*)
- Gamification

[1] http://wiki.openstreetmap.org/wiki/OAuth
[2] http://airbnb.io/enzyme/
[3] https://github.com/mochajs/mocha

  - von gesammelten *Koins* abhängige Levels einführen (z.B. bestimmte Fehlertypen erst ab fortgeschrittenem Level anzeigen, Avatars, Levelbezeichnungen)
  - verschiedene Schwierigkeitsstufen
  - Einbindung in *Game Center* der jeweiligen Plattform
  - weitere Badges einführen (viele Ideen finden sich hier: https://wiki.openstreetmap.org/wiki/Badges, zum Beispiel auch Badges für Spielertypen)
  - verschiedene Highscores anzeigen (zum Beispiel zeitlich oder Regional begrenzt, nach Fehlertypen kategorisiert, schnellste Aufsteiger)
  - zusätzliche Berechtigungen für erfahrene Benutzer (für den langfristigen Erfolg)
- neue realistische Fehlertypen:[4]
  - Hausnummern einfügen
  - Stockwerk-Anzahl einfügen
  - Einbahnstrassen erfassen
  - Öffnungszeiten von öffentlichen Gebäuden festhalten
- weniger realistische Fehlertypen:
  - Kreisel erfassen
  - Bushaltestellen von *DIDOK*[5] erfassen
- Erkennen von Benutzern, die nicht sorgfältig validieren[6]
- ausführliche Statistiken für individuelle Benutzer[7]
- Aufträge aus *wheelmap*[8] einfügen
- Offline-Fähigkeit (offline Maps für *React Native* wären erforderlich)
- wenn Aufträge zum Beispiel drei Mal nicht gelöst werden können, soll eine *OSM*-Notiz generiert werden

Unrealistische Arbeiten

Punkte, die für KORT als weniger geeignet empfunden wurden:

[4] https://github.com/kort/kort/issues/81
[5] https://didok.osm.ch/
[6] https://github.com/kort/kort/issues/7
[7] https://github.com/kort/kort/issues/71
[8] http://wheelmap.org

- Erweitern der Verifizierung mit der Möglichkeit, ein Foto als Beweis hochzuladen
  - *Begründung:* Aspekte des Datenschutzes bergen ein gewisses Risiko. Benutzer müssten für das Hochladen von Bildern zusätzliche Bedingungen akzeptieren.
- standortunabhängige Aufgaben lösen (Gefahr von Couch Mapping)
  - *Begründung:* Es ist ein Anliegen der *OSM* Community, dass die Mapper vor Ort sein sollen um Aufträge zu lösen.

11. Installation

In diesem Kapitel werden zwei Varianten beschrieben, wie die App getestet werden kann.

11.1. Installation mit Source Code

Um die App anhand des Source Codes zu installieren, muss die *React Native* Entwicklungsumgebung aufgesetzt sein. Wie diese eingerichtet wird ist in der Getting-Started-Anleitung, der online Dokumentation von *React Native*[1], erklärt. Diese Anleitung erklärt Schritt für Schritt den Ablauf von der Einrichtung der Entwicklungsumgebung auf einem *Mac*-, *Linux*- oder *Windows*-Rechner (für *iOS* und *Android*) bis zum Starten der App.

Hier muss noch beachtet werden, dass die SecretConfig.js-Datei
`(/kort-reloaded/js/constants/SecretConfig.js)`
mit folgenden Werten ergänzt werden sollte:

- Mapbox Access Token
  - Dieses Token kann auf der Mapbox-Webseite[2] erstellt werden.
- Google Client ID
  - Wie die Google Client ID erhalten wird, ist in der Anleitung vom Github-Projekt der Google-Signin-Library erklärt.[3]
  - Wichtig ist, dass anhand dieser Anleitung auch die google-services.json-Datei neu generiert und in das Projekt eingefügt werden muss.

11.2. Installation mit APK-Datei

Damit die *Android*-App, von der bereitgestellten APK-Datei auf der CD, installiert werden kann, muss das Smartphone mit dem Computer per USB-Kabel verbunden sein und erkannt werden.

1. APK-Datei in einen Ordner auf dem Gerätespeicher kopieren

[1] https://facebook.github.io/react-native/docs/getting-started.html
[2] https://www.mapbox.com/help/create-api-access-token/
[3] https://www.mapbox.com/help/create-api-access-token/

2. Smartphone Verbindung mit dem Computer trennen
3. APK-Datei auf dem Smartphone auffinden und anklicken
4. App-Herausgeber vertrauen und Installation abschliessen

Teil III.

Projektmanagement

12. Projektmanagement

Hier finden Sie die Angaben zum Projekt, zu den eingesetzten Entwicklungswerkzeugen und zur eingesetzten Software.

- Website: http://www.kort.ch/ (http://play.kort.ch/)
- Source Code: https://github.com/kort/kort-reloaded/
- Projektmanagement: Redmine (sinv-56059.edu.hsr.ch/redmine/projects/ba-kort/)
- Issues: (http://sinv-56059.edu.hsr.ch/redmine/projects/ba-kort/issues (Backlog: im Wiki auf Redmine (mit SK für eigene Issues gekennzeichnet))
- Dokumentation: https://github.com/kort/kort-reloaded-docu

12.1. Team

- *Betreuer*: Prof. Stefan F. Keller
- *Projektpartner*: Liip AG, Limmatstrasse 183, CH-8005 Zürich
  - Hr. Jürg Hunziker
  - Hr. Stefan Oderbolz
- *Experte*: Hr. Claude Eisenhut
- *Gegenleser*: Prof. Beat Stettler

Autoren

Diese Arbeit wird als Bachelorarbeit an der Abteilung Informatik durchgeführt von

- Hr. Marino Melchiori
- Hr. Dominic Mülhaupt

12.2. Risikomanagement

Daher, dass KORT bereits in einer vorhergehenden Bachelorarbeit erfolgreich umgesetzt werden konnte und Anklang gefunden hat, wurden bereits viele Risiken abgedeckt.
Dennoch wurde das Risikomanagement nicht vernachlässigt. Alle bekannten Risiken sind gesammelt aufgelistet und wurden nach jedem Meilenstein neu evaluiert.

12.2.1. Risikoanalyse

Am Anfang der Bachelorarbeit wurden folgende Risiken identifiziert:

R01: Mangelnde Erfahrung mit *JavaScript*, *React* und *React Native*	
Beschreibung	Wirkt sich negativ auf Design und Programmcode aus.
Schadenspotential	60h
Eintrittswahrsch.	65%
Auswirkung	Da die Entwickler sich mit den zugrundeliegenden Programmierkonzepten vertraut machen während sie bereits planen und Entscheidungen treffen, teilweise sogar schon programmieren müssen, kann und wird es vorkommen, dass gewisse Entscheidungen schlecht getroffen und gewisse Konzepte nicht sauber umgesetzt werden. Dies kann zu Verspätungen oder unsauberem Programmcode führen.
Vorbeugung	Die Entwickler haben mit Herr Keller besprochen, dass das Minimum Viable Product der Bachelorarbeit eine Android App mit der gleichen Funktionalität, wie sie bereits im ursprünglichen Kort Game umgesetzt wurde, sein wird. Da im Rahmen dieser Arbeit schlicht keine Zeit bleibt um sich umfassend in die Technologien einzuarbeiten bevor die restliche Arbeit angepackt ist, wird vor allem zu Beginn vermehrt auf Pair Programming gesetzt. Die Entwickler haben sich auch in regelmässigen Abständen die Zeit genommen, ein Code Review durchzuführen. Ausserdem ist es wichtig, dass bei Schwierigkeiten nicht zu lange gezögert wird, den Kontakt zu Experten im jeweiligen Bereich zu suchen.
Massnahmen beim Eintreffen	Komplexe Features vereinfachen und so gestalten, dass sie leicht erweiterbar sind.

Tabelle 12.1.: Risiko R01

R02: Es existiert keine passende Map Library für *React Native*	
Schadenspotential	70h
Eintrittswahrsch.	30%
Auswirkung	Es müsste statt *React Native* ein alternatives mobiles Framework gefunden werden, welches die *OSM*-Daten anzeigen kann.
Vorbeugung	Zu Beginn des Projektes muss eine *React Native* Prototyp-Applikation implementiert werden, welche *OSM*-Daten auf der Karte darstellt.
Massnahmen beim Eintreffen	Auf eine aufwendigere Variante der Kartendarstellung, die im Kapitel Evaluation evaluiert wurde, zurückgreifen.

Tabelle 12.2.: Risiko R02

R03: *KeepRight* stellt den Dienst ein	
Schadenspotential	70h
Eintrittswahrsch.	10%
Auswirkung	Die Anzahl der zur Verfügung stehenden Missionen würde stark eingeschränkt werden. Ausserdem wären diese auf die Schweiz beschränkt. Ein anderer Dienst (z.B. *Osmose*) müsste eingesetzt werden, was vor allem Änderungen im Backend erfordern würde.
Vorbeugung	Bereits früh im Projekt wird das Thema mit dem Projektpartner besprochen um festzustellen, ob dieser bereit wäre, sich dieser Problematik anzunehmen.
Massnahmen beim Eintreffen	Ausarbeitung einer neuen Aufgabenstellung mit Herrn Prof. S. Keller, Jürg Hunziker und Stefan Oderbolz, die Änderungen am Backend beinhaltet.

Tabelle 12.3.: Risiko R03

R04: *React Native* ist noch nicht ausgereift genug für *Android*	
Schadenspotential	20h
Eintrittswahrsch.	30%
Auswirkung	*Android* wird erst seit Oktober 2015 durch *React Native* unterstützt. Es stehen noch nicht alle Funktionalitäten wie für *iOS* zur Verfügung.
Vorbeugung	Die mangelnden Funktionalitäten (wie zum Beispiel im Testing) werden bereits bevor mit der Implementation begonnen wird, so weit möglich, analysiert. Grundsätzlich sollte es möglich sein, die App für *Android* umzusetzen da bereits komplexere Apps damit erstellt wurden. Wahrscheinlich wird es vorkommen, dass Workarounds nötig sein werden. Im schlimmsten Fall müsste während der Entwicklung auf *iOS* umgestellt werden, was grundsätzlich möglich ist.
Massnahmen beim Eintreffen	Ausarbeitung einer neuen Aufgabenstellung mit Herrn Prof. S. Keller, die eine *iOS*-Version bevorzugt.

Tabelle 12.4.: Risiko R04

R05: Mangelnde Erfahrung mit der flux-Architektur	
Beschreibung	Wirkt sich negativ auf Design und Programmcode aus.
Schadenspotential	30h
Eintrittswahrsch.	40%
Auswirkung	Da die Konzepte der flux-Architektur neu erarbeitet werden müssen und flux in der Community als eher komplex eingestuft ist, kann die Entwicklung mehr Zeit beanspruchen.
Vorbeugung	Die korrekte Umsetzung wird bereits vor dem Start der Implementation so weit wie möglich analysiert. Dabei wurden die Themengebiete zum Einlesen aufgeteilt und am schlussendlich besprochen und erklärt.
Massnahmen beim Eintreffen	Rücksprache mit dem IFS-Team, das an einem *React*-Projekt arbeitet.

Tabelle 12.5.: Risiko R05

R06: Mangelnde Erfahrung mit OAuth	
Beschreibung	Keiner der Entwickler ist mit OAuth vertraut. Dadurch, dass sich die Implementation der Authentifizierung bei einem native Client zu einer Implementation einer Web-App unterscheidet (Token- vs. Session-Basiert), wird mehr Zeit beansprucht.
Schadenspotential	40h
Eintrittswahrsch.	60%
Auswirkung	Die *OSM*-Community würde nur sehr ungern auf einen entsprechenden Login verzichten.
Vorbeugung	Mehr Aufwand in die Evaluation stecken.
Massnahmen beim Eintreffen	Rücksprache mit Herrn Prof. S. Keller

Tabelle 12.6.: Risiko R06

Risikoanalyse MS2: Ende Elaboration

Am 29. März 2016 (Fälligkeitsdatum des Meilensteins) wurden die Risiken erneut besprochen.

R01: Mangelnde Erfahrung mit *JavaScript*, *React* und *React Native*: Die Eintrittswahrscheinlichkeit konnte auf 50% gesenkt werden.

R03: *KeepRight* stellt den Dienst ein: Wurde nach Absprache mit Jürg Hunziker und Stefan Oderbolz ganz eliminiert.

Risikoanalyse MS3: 1. Prototyp

R02: Es existiert keine passende Map Library für *React Native*: Die Wahrscheinlichkeit des Eintretens konnte nach erfolgreicher Implementation auf 20% gesenkt werden. Dadurch, dass noch keine Missionen gelöst wurden, galt das Risiko noch nicht als behoben.

Risikoanalyse MS4: Zwischenpräsentation

Das Eintreten von **R02** konnte weiterhin auf 10% gesenkt werden. Die Marker waren klickbar und sie enthielten alle nötigen Informationen für das Lösen einer Mission.

Risikoanalyse MS5: 1. Release

R02 wurde eliminiert, **R01** auf 20% gesenkt und **R05** als tief eingeschätzt (10%).

Risikoanalyse MS6: 2. Release

R05 wurde ebenfalls eliminiert - die flux-Architektur war stabil und eine Mission konnte erfolgreich gelöst werden.

Risikoanalyse MS7: Schlussabgabe

R04: *React Native* ist noch nicht ausgereift genug für *Android*: Wurde dank einer Lauffähigen Version auf 10% gesenkt.

R06: Mangelnde Erfahrung mit OAuth: Dies ist ein weiterhin bestehendes Risiko - die *OSM*-Authentifizierung konnte leider nicht umgesetzt werden.

12.3. Projektplan

Die Planung wurde nach dem ersten Kickoff-Meeting festgehalten. Der erste Prototyp, für die Demo an der Zwischenpräsentation, war am 11.04.2016 geplant. Am 16.05. war dann das erste Release, mit dem Missionen gelöst werden können, vorgesehen. Um bei der Abgabe vom Projekt die App zu veröffentlichen, entstand der folgende Projektplan.

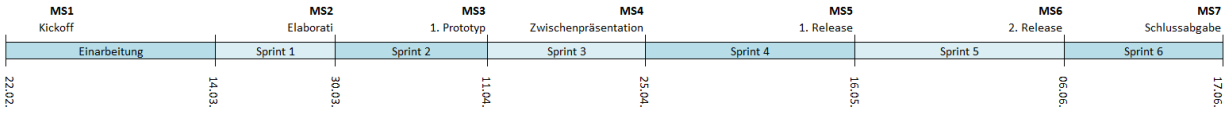

Abbildung 12.1.: 1. Projektplan als Zeitstrahl dargestellt

Beim dritten Sprint wurde festgestellt, dass die Planung zu optimistisch war. Die Planung wurde neu besprochen und in einem neuen Zeitstrahl festgehalten. Es gab Schwierigkeiten bei der Evaluation der Architektur und von den Libraries. Die Libraries und Komponenten wurden in einzelnen Projekten getestet. Das hat uns zu neuen Erkenntnissen für die Architekturentscheidung verholfen.

Erst beim Meilenstein 5 konnte eine Version mit den definitiv evaluierten Komponenten entstehen. Beim achten Meeting, am 29.04.2016, wurde mit Herrn Stefan Keller besprochen, dass wir das Projekt nach der Abgabe gerne weiterführen würden. Dann wäre es nämlich auch möglich, die App zu veröffentlichen.

Bis zum Ende vom Meilenstein 6 entstand ein erstes Release, von einem Prototypen mit der geplanten Grundfunktionalität. Bei der Abgabe ist das Projekt in einem Zustand, bei dem sich neue Features zügig implementieren lassen. Dieser aktuelle Stand wäre am Anfang vom Projekt für den Meilenstein 5 geplant gewesen.

Somit ist dieser neue Projektplan entstanden.

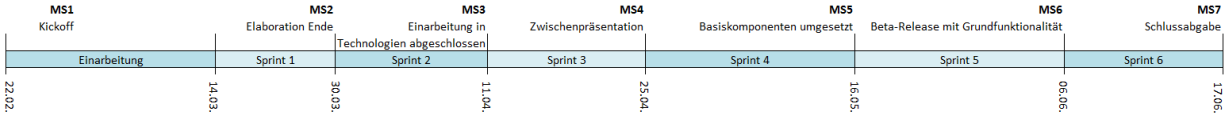

Abbildung 12.2.: 2. Projektplan als Zeitstrahl dargestellt

12.4. Sprints

Durch die Anwendung von Scurm-Ansätzen, wurden Sprints geplant. Eine Sprint-Periode dauerte immer bis zum Ende von einem Meilenstein. Am Anfang von einem Sprint wurde das konkrete Vorgehen geplant und am jeweiligen Ende sind die Schwerpunkte dokumentiert worden.

12.4.1. Sprint 1

14.03.2016 bis 30.03.2016

Zum Schwerpunkt in diesem Sprint gehörte die Einarbeitung in die verwendeten Technologien und die Einarbeitung in das bestehende KORT-Projekt.

12.4.2. Sprint 2

30.03.2016 bis 11.04.2016

Im zweiten Sprint wurden die benötigten Libraries evaluiert und getestet. Die Details und die Begründungen zu den Entscheidungen wurden im Kapitel Evaluation festgehalten.

12.4.3. Sprint 3

11.04.2016 bis 25.04.2016

Neben der Vorbereitung der Zwischenpräsentation wurden die Anforderungen aktualisiert. Das ersetzen der Validationen im Frontend kam neu dazu. Zukünftige Arbeiten und Ideen wurden ebenfalls gesammelt und im Kapitel Realistische Arbeiten dokumentiert.

12.4.4. Sprint 4

25.04.2016 bis 16.05.2016

Am Anfang vom Sprint vier wurde abgeklärt ob ein Webprototyp mit *React* umsetzbar ist. Dies stellte sich dann aber als zu Aufwendig heraus — Die Idee (für eine folgende Studienarbeit) einer *React*-Web-App wurde Abgewiesen. Am Ende des Sprints wurde festgelegt, was

getestet werden muss.

12.4.5. Sprint 5

16.05.2016 bis 06.06.2016

Während diesem Sprint fand das Meeting mit Jürg Hunziker und Stefan Oderbolz, um die Authentifizierung abzuklären. Es ging darum, neu zu evaluieren, wie das Login in der App umgesetzt wird und was dies für Änderungen am Backend mit sich bringen würde.

12.4.6. Sprint 6

06.06.2016 bis 17.06.2016

Die Validationen konnten im Frontend, anhand von Anpassung der Businesslogik, abgeschafft werden. Ausserdem konnte die wichtige Internationalisierung für die Sprachen Englisch und Deutsch umgesetzt werden. Am Ende von Sprint 6 wurde ein Code Freeze bis zum Abgabetermin eingeleitet.

12.5. Meilensteine

In diesem Projekt wurde die Planung auf neun Meilensteine (MS) aufgeteilt. Die Meilensteine MS 8 und MS 9 finden nach der offiziellen Schlussabgabe statt und sind deswegen noch ohne Resultate dokumentiert.

12.5.1. MS1: Kickoff

Fällig am 25.02.2016

Resultate

- Kickoff Meeting bei Liip mit Jürg Hunziker, Stefan Oderbolz und Stefan Keller

12.5.2. MS2: Ende Elaboration

Fällig am 29.03.2016

Resultate

- Infrastruktur aufgesetzt
  - Datenbank
  - Continuous Integration

  - uservoice
  - Redmine
  - Installationsskripte
  - Dokumentation aufgesetzt
- Ausgangslage definiert und dokumentiert
- Anforderungsspezifikation erarbeitet
- Risikomanagement analysiert und dokumentiert
- Projektplan erarbeitet
- Map Komponente ausgewählt und eingesetzt
- Aufgabenstellung erarbeitet
- Testspezifikation erarbeitet
- Einarbeitung in die KORT-Web-App
- GUI-Mockups

Erledigte Arbeiten

Eine erste Version der Aufgabenstellung konnte erarbeitet werden. Die Dokumentation wurde eingeleitet. Es gelang uns einen automatisierten Build der App, aus dem *GitHub*-Sourcecode, mit *Travis CI* aufzusetzen. Für die Map Komponente wurde die MapBox GL Library[1] mit den Vektor Daten von osm2vectortiles[2] evaluiert und getestet. Nebenbei konnten wichtige Erfahrungen in den verwendeten Technologien (*Java Script*, *React* und *React Native*) gemacht werden. Zusätzlich wurde ein für *Android* angepasstes Design entworfen und Grundkonzepte der Architektur erarbeitet. Die Architektur ist noch nicht final, sie erleichtert uns aber den Einstig beim Programmieren. Die Infrastruktur ist soweit aufgesetzt.

Probleme

Für die detaillierte Erarbeitung der Testspezifikation fehlte noch die nötige Erfahrung in *React Native*.

12.5.3. MS3: Evaluation der Komponenten

Fällig am 08.04.2016

[1] https://libraries.io/npm/react-native-mapbox-gl
[2] http://osm2vectortiles.org/

Resultate

- Einarbeitung in *JavaScript*, *React* und *React Native*
- *Android* Prototyp
  - Tab-Navigation
  - Darstellung der Karte
  - Evaluation der Architektur und Implementation des Grundgerüstes
  - Missionen auf Map anzeigen
  - Authentifizierung mit OAuth evaluiert
- *Travis-CI* Konfiguration aktualisieren

Erledigte Arbeiten

Die Entwicklungsumgebung wurde optimiert. Das *Travis*-Konfigurationsfile prüft den Code nun mit *ESLint*[3] (*JavaScript* linter, checkt Styleguidelines) und *flow*[4] (static type checker).

Für die Tab-Navigation konnte eine Demo mit react-native-router-flux[5] umgesetzt werden. Diese muss noch in der KORT App implementiert werden.

OAuth (nur Client-Seitig, ohne Backend Kommunikation) wurde evaluiert und konnte dann anhand einer Demo getestet werden — allerdings nur mit *Facebook* und *Google*.

Die Darstellung der Karte aus MS2: Ende Elaboration wurde leicht ausgebaut. Neu wird nun der Standort des Benutzers ermittelt.

Die Missionen konnten im Umkreis von fünf Kilometern geladen werden.

Probleme

Schwierigkeiten traten vor allem im Zusammenhang mit *React Native* auf. Oft gab es Build-Fehler bei gleicher Code-Basis. Die Fehlerbehandlung hat uns enorm viel Zeit gekostet und es war schwer im Internet Hilfestellungen zu erhalten. Da *React Native* jede zwei Wochen ein Update erhält, sind die Dokumentationen oder die Diskussionen im Internet teilweise schon wieder veraltet.
Im Austausch mit anderen *React Native* Entwicklern haben wir dasselbe Feedback erhalten. Der Prototyp konnte nicht wie geplant fertiggestellt werden und einige Arbeiten mussten auf den nächsten Meilenstein ausgelagert werden.

Die Authentifizierung mit dem Backend konnte noch nicht umgesetzt werden.

[3] http://eslint.org/
[4] http://flowtype.org/
[5] https://github.com/aksonov/react-native-router-flux

12.5.4. MS4: Zwischenpräsentation

Fällig am 22.04.2016

Resultate

- *Mapbox* Prototyp fertig
  - Missionen mit Marker auf Karte dargestellt
  - Tab-Navigation implementiert
- Zwischenpräsentation (Dauer ca. 30 Minuten)
  - Aufgabenstellung, Problembesprechung
  - IST-Situation
  - geplantes Resultat
  - Beschlussprotokoll für den Betreuer

Erledigte Arbeiten

Der Prototyp enthielt nun die Karte in einer Tab-Ansicht. An den jeweiligen Positionen der geladenen Missionen konnten Marker eingefügt werden. Durch einen Klick auf einen Marker konnte der Titel der Mission erfolgreich in die Konsole geloggt werden.

Leider konnte noch immer keine Lösung für die Implementation eines Logins mit OAuth 1.0a (*OSM*) gefunden werden. Dieses Feature noch zur Abgabe zu liefern wäre unrealistisch. Deswegen wurde es auf den Meilenstein 9: Release für App Store verschoben.

Probleme

Weiterhin traten Schwierigkeiten mit willkürlichen Fehlern der *React Native* App auf. Aus diesen Gründen konnte die Navigation und die Struktur der App nicht abschliessend umgesetzt werden. Dass der Benutzer nach dem Login automatisch zur Tab-Ansicht weitergeleitet wurde, hatte wegen einem Fehler in der Navigations-Library nicht geklappt. Zwischenzeitlich wurde aus diesem Grund das Erhalten des Login-Tokens, nach der Authentifizierung über *Google* und *Facebook*, in einer separaten App getestet.

12.5.5. MS5: Basis-Komponenten umgesetzt

Fällig am 20.05.2016

Resultate

- native location tracking implementiert
- View-Komponenten fertig entworfen
- Testing-Framework aufgesetzt
- Architektur-Entscheid und -Implementation
- Token basierte *Google*-Authentifizierung im Frontend umgesetzt, nachdem das Backend von Stefan Oderbolz dafür angepasst wurde.
- KORT-Datenbank für Tests lokal aufgesetzt
- KORT ist *iOS*-fähig
- *React*-Web-App evaluiert
- Validationen vom Backend als Missionen laden
- Kurzvideo-Konzept besprochen

Erledigte Arbeiten

Wir haben die Architekturvarianten evaluiert und uns für die *Flux*-Architektur entschieden. Daraufhin wurde das bestehende – noch sehr schlanke – Grundgerüst, welches mit dem MVC-Pattern umgesetzt wurde, durch *Flux* ersetzt.

Unterdessen konnte das Grundgerüst der GUI grösstenteils fertiggestellt werden. Es fehlte noch die dynamische Behandlung von gewissen View-Komponenten. Zum Beispiel die Unterscheidung ob beim Lösen einer Mission ein Picker, zum Auswählen der Antwort, gerendert wird, oder ob ein Text-Input-Feld angeboten wird.

KORT konnte erfolgreich auf *iOS* getestet werden.

Probleme

Weiterhin war es nicht möglich mit der Router-Flux-Navigationskomponente eine Weiterleitung, nach Erfolgreichem Login, zur Kartenansicht umzusetzen. Hierbei handelte es sich um den bekannten Fehler dieser Komponente. Das Verzögerte leider das geplante Ziel: eine Mission zu lösen.

Die KORT-Datenbank konnte nur bei einem Entwickler richtig aufgesetzt werden. Es gab Probleme mit der Installation der Scripts auf *OS X*. Aus diesem Grund wurde für Tests, nach Absprache mit Jürg Hunziker, das Development-KORT-Backend genutzt.

12.5.6. MS6: Beta-Release mit Grundfunktionalität

Fällig am 06.06.2016

Resultate

- dynamische View-Komponenten fertiggestellt
- Login Weiterleitung
- Login-Logik mit Local Storage
- Testing abgeschlossen
- Missionen und Validationen sind lösbar
- Internationalisierung umgesetzt

Erledigte Arbeiten

Alle View-Komponenten sind nun dynamisch und zeigen je nach Zustand der Komponente die entsprechende Oberfläche.

Durch eine zusätzliche App-Loader-View konnte das Problem der Weiterleitung nach dem Login behoben werden. Diese Komponente lädt nun alles im Voraus und erst danach wird je nach Zustand, ob der Benutzer eingeloggt ist, oder nicht, die entsprechende Ansicht angezeigt. Nach und nach wurden die Platzhalter der View-Komponenten mit dynamischen Daten der Businesslogik ersetzt.

Schlussendlich konnten wir erfolgreich Missionen und Validationen lösen.

Probleme

12.5.7. MS7: Schlussabgabe

Fällig am 17.06.2016

Resultate

- gebundene, vollständige Dokumentation eingereicht
- CD mit Abgabe eingereicht
- Abstract eingereicht
- Poster eingereicht

12.5.8. MS8: Schlusspräsentation

Fällig am 29.06.2016

Ziele

- Badges anzeigen
- aktualisierte Dokumentation
- Design Verbesserungen
- Präsentation fertiggestellt
- Handouts ausgedruckt

12.5.9. MS9: Release für App Store

Fällig am 03.07.2016

Ziele

- *Android* App im *Google Play* Store veröffentlicht

13. Projektmonitoring

13.1. Zeitanalyse

Für eine Bachelorarbeit werden 12 ECTS-Punkte vergeben, wobei ein Punkt einem Aufwand von 30 Stunden entspricht. In einem Team von zwei Entwicklern, entspricht dies 360 Stunden Aufwand pro Person. Leider haben wir diese Vorgabe (siehe Tabelle 13.1) überschritten.

Person	**Aufwand**
Marino Melchiori	461.5 h
Dominic Mülhaupt	467 h

Tabelle 13.1.: Arbeitsaufwand pro Person

Wenn die gleich folgenden Tabellen 13.1 und 13.1 verglichen werden, ist die Arbeitsaufteilung sehr gut erkennbar. Dies ist in den Unterschieden des Aufwandes, in den Aktivitäten Analyse, Implementation, Dokumentation und Testing, erkennbar. Bei neuen Erfahrungen haben sich die Entwickler ausgetauscht und waren so immer auf dem aktuellen Stand.

Marino Melchiori	
Aktivität	**Aufwand**
Analyse & Design	106.00 h
Implementation	125.00 h
Dokumentation	141.00 h
Requirements	30.75 h
Deployment	16.25 h
Allgemein	16.75 h
Projektmanagement	25.75 h

Tabelle 13.2.: Aufwand pro Aktivität — Marino Melchiori

Dominic Mülhaupt	
Aktivität	**Aufwand**
Analyse & Design	49.50 h
Implementation	191.75 h
Dokumentation	66.25 h
Requirements	26.00 h
Deployment	7.00 h
Allgemein	52.50 h
Projektmanagement	52.00 h
Testing	22.00 h

Tabelle 13.3.: Aufwand pro Aktivität — Dominic Mülhaupt

13.1.1. Soll-Ist-Zeitvergleich

Die Tabelle zum 13.1.1 zeigt, wie das Projekt kategorisiert wurde. In der *Kort Projekt*-Kategorie ging es vor allem um die Implementation der Architektur und die Verwendung und Einrichtung von *React Native*. Die Kategorien *Login*, *Missionen*, *Profil* und *Highscore* enthalten die Implementation der GUI. *Map* und *Internationalisierung* sind Kategorien, bei denen es um die Installation der entsprechend verwendeten Libraries ging. In der Kategorie *Technologien* wurde die Einarbeitungszeit gebucht. Die *Qualitätssicherung* beinhaltet das Testing und die Kategorie *Allgemein* die Evaluation von verwendeten Konzepten und Libraries.

Die Schätzung fand jeweils bei der Erstellung eines Tickets in einer Kategorie statt.

Die Differenz der *Kort Projekt*-Kategorie lässt sich durch die häufigen Build-Fehler erklären. Das Suchen nach Lösungen im Internet war sehr Zeitaufwendig.

Bei der Kategorie *Missionen* wurden 15 Stunden zu viel geschätzt. In diesem Fall setzten wir vermehrt auf Pair Programming, da es sich dort um eine der ersten umgesetzten Funktionen handelte. Somit konnte kostbare Zeit, die für das Refactoring geplant war, eingespart werden. Später folgende Features, der Kategorien *Login*, *Profil* und *Highscore*, wurden viel besser geschätzt.

Schlussendlich wurden 21 Stunden zu viel geschätzt, was bei einem Arbeitstag von 8 Stunden etwa 2.5 Arbeitstagen entspricht.

Soll-Ist-Vergleich vom Gesamtaufwand der Kategorien			
Kategorie	**Soll-Aufwand**	**Ist-Aufwand**	**Differenz**
Kort Projekt	71.75 h	88.25 h	+16.50 h
Login	91.50 h	92.50 h	+1.00 h
Map	47.75 h	45.50 h	-2.25 h
Missionen	59.75 h	44.75 h	-15.00 h
Highscore	11.50 h	12.70 h	+1,20 h
Profil	18.00 h	16.25 h	-1.75 h
Technologien	182.50 h	187.55 h	+0.05 h
Gamification und Design	7.25 h	4.75 h	-2.50 h
Qualitätssicherung	36.50 h	30.00 h	-6.50 h
Internationalisierung	4.50 h	4.25 h	-0.25 h
Dokumentation	204.00 h	198.25 h	-5.75 h
Allgemein	152.75 h	144.50 h	-8.25 h
Meeting	61.75 h	59.25 h	-2.50 h
Total	**975.00 h**	**926.50 h**	**-21.00 h**

Tabelle 13.4.: Soll-Ist-Vergleich — Gesamtaufwand pro Kategorie

13.2. Code-Statistik

Zur groben Einschätzung, der Grösse vom Projekt wurde in der Tabelle 13.5 die Gesamtanzahl der *JavaScript* der Code-Zeilen (ohne Code-Kommentare) gezählt. Beim Linken von Libraries gab es auch Anpassungen im *Android*- und *iOS*-Projekt. Der Umfang dieser Änderungen wurde nicht dokumentiert, da es sich vor allem um Anpassungen an vorhandenen Dateien handelte.

Sprache	**Zeilen**
JavaScript	3913

Tabelle 13.5.: Dateien und Codezeilen

Die Tabelle 13.6 zeigt die Anzahl an Code-Zeilen in einem Package. Die Packages `Actions`, `Data`, `Dispatcher`, `DTO` und `Stores` sind von der verwendeten Architektur gegeben. `Constants` beinhaltet Konstanten, IDs und sonstige fixe Parameter. Im Package `Components` befinden sich die GUI-Komponenten. Die `Shared Components` enthalten Buttons und weitere Komponenten, die oft wiederverwendet wurden.

Package	**Zeilen**
Components	1042
Shared Components	463
Total Components	**1511**
Actions	267
Data	513
Dispatcher ohne Tests	2
Dispatcher Tests	43
DTOs ohne Tests	199
DTO Tests	69
Stores ohne Tests	408
Store Tests	466
Total Logik ohne Tests	**1389**
Constants	441

Tabelle 13.6.: Codezeilen pro Package

Teil IV.

Anhänge

Glossar

API

Application Programming Interface, Schnittstelle für die Programmierung[25]. 26, 33

Backend

Als Backend wird das auf dem Server laufenden Programm bezeichnet. ii, 24, 26, 27, 33

Continuous Integration

Unter dem Begriff *Continuous Integration*[18] beschreibt die Idee, dass Änderungen an einer Software schnell eingebracht werden sollen. Dazu zählt, dass diese in einem Versionsverwaltungs-Tool eingetragen und durch automatisierte Tests geprüft werden[25]. 30, 48

Crowdsourcing

Unter Crowdsourcing versteht man die Auslagerung von Auslagerung von traditionell internen Teilaufgaben an eine Menge von freiwilligen Usern im Internet[26]. 5

DOM

DOM bedeutet Document Object Model und ist eine Spezifikation für den Zugriff auf HTML-Dokumente. Anhand der DOM kann ein Computerprogramm den Inhalt der HTML-Elemente dynamisch verändern. Für einen Webentwickler wäre das der HTML-Code.[28]. 20

DTO

Ein Datentransferobjekt ist ein Objekt, das nur zur Übertragung von Daten oder Informationen dient und keine logischen Komponenten enthält. 24

Framework

Ein Framework ist noch kein fertiges Programm, sondern ein Rahmen, ein Gerüst, das der Programmierer verwenden kann um sein eigenes, persönliches Programm zu erstellen[29]. 2, 31, 35, 43, 52

Frontend

Als Frontend wird das beim Client laufende Programm bezeichnet. 7, 25

Funktionaler Test

Funktionale Tests überprüfen das Gesamtverhalten des Systems. Mit einem automatisierten UI Testing Tool werden verschiedene Szenarien durchlaufen, welche aus den Use Cases abgeleitet werden können. Funktionale Tests werden im Vergleich zu Unit Tests und Integration Tests eher spärlich eingesetzt. 31

Gamification

Unter Gamification versteht man das Hinzufügen von Spiel-Elementen in einem nicht spieltypischen Kontext.[20]. 5, 35

GUI

GUI ist eine Abkürzung für graphical user interface und bedeutet grafische Benutzeroberfläche[30]. 21, 22, 49, 52, 56, 57

IDE

IDE bedeutet Integrierte Entwicklungsumgebung – auf englisch integrated development environment[31]. 30

Integration Test

In Integration Tests wird die Zusammenarbeit von Systemteilen getestet. 31, 61

JSX

JSX ist eine *JavaScript*-Syntax-Erweiterung, die ähnlich wie XML aussieht[9]. 20, 27

Library

Eine Library (Programmbibliothek) bezeichnet in der Programmierung eine Sammlung von Unterprogrammen, die Lösungswege anbieten. Bibliotheken laufen im Unterschied zu Programmen nicht eigenständig, sondern sie enthalten Hilfsmodule, die angefordert werden können[34]. 20, 32, 33, 49, 51, 56, 57

Mapper

Personen welche auf OpenStreetMap die Karten ergänzen und pflegen, nennen sich selbst *Mapper*[25]. ii

Minimum Viable Product

Das Minimum Viable Product ist ein Produkt, welches gerade die Kernfunktionalität aufweist, die nötig ist, um es zu veröffentlichen[32]. 42

Mock

Mock-Objekte werden in diversen Tests verwendet um das Verhalten eines anderen Objekts für spezifische Fälle zu simulieren ohne es tatsächlich zu implementieren. 31, 63

MVC

Der englischsprachige Begriff *model view controller* (MVC) ist ein Muster zur Strukturierung von Software-Entwicklung in die drei Einheiten Datenmodell (model), Präsentation (view) und Programmsteuerung (controller)[4]. 7, 8, 20, 25

OAuth

OAuth ist ein offenes Protokoll, das eine standardisierte, sichere API-Autorisierung erlaubt[19]. 12, 13, 35, 45, 46, 50, 51

Packager

Der Packager kompiliert und bundelt den JavaScript Code für das Smartphone während der Entwicklungsphase und dem Testen auf dem Gerät.. 23

Pair Programming

Bei Pair Programming handelt es sich um eine Arbeitsweise, bei der zwei Programmierer an einem Rechner arbeiten. Dabei ist ein Programmierer damit beschäftigt, den Code zu schreiben, während der andere über die Problemstellungen nachdenkt, den geschriebenen Code kontrolliert sowie Probleme, die ihm dabei auffallen, sofort anspricht[33]. 42, 56

POI

Abkürzung für Point of Interest. Dies ist ein allgemeiner Begriff für einen Ort mit irgendeiner Bedeutung, sei es eine Schule, Kirche, Bushaltestelle oder sonst etwas von besonderem Interesse[25]. 2

REST

Representational State Transfer[17] ist ein Programmierparadigma, welches besagt, dass sich der Zustand einer Webapplikation als Ressource in Form einer URL beschreiben lässt. Auf eine solche Ressourcen können folgende Befehle angewendet werden: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `HEAD` und `OPTIONS`. HTTP ist ein Protokoll welches REST implementiert[25]. 24, 26, 32

Singleton

Das Singleton beschreibt ein Entwurfsmuster, wonach sichergestellt wird, dass von einer Klasse immer höchstens ein Objekt existiert, welches üblicherweise global verfügbar ist[37]. 25

Unit Test

Unit Tests werden geschrieben um kleine Teile des Codes – üblicherweise Module – zu testen. Da wirklich nur das Verhalten des entsprechenden Moduls getestet werden soll, müssen Abhängigkeiten dieses Moduls durch Mocks ersetzt werden. 31, 61

Virtual DOM

Die virtual DOM ist eine Abstraktion der DOM. Es ist eine lokale Kopie der DOM[21]. 20

Web-App

Der Begriff Web-App (von der englischen Kurzform für web application), bezeichnet im allgemeinen Sprachgebrauch Apps für mobile Endgeräte wie Smartphones und Tablet-Computer, die über einen in das Betriebssystem integrierten Browser aus dem Internet geladen und so ohne Installation auf dem mobilen Endgerät genutzt werden können[27]. ii, 2, 3, 5, 6, 10, 15, 21, 35, 45, 47, 49, 52

Literaturverzeichnis

[1] Christian Alfoni. Why we are doing MVC and FLUX wrong, 2015. [Online; Stand 15. Juni 2016]. URL: http://www.christianalfoni.com/articles/2015_08_02_Why-we-are-doing-MVC-and-FLUX-wrong.

[2] Auth0. Authenticate React Native – iOS with Generic OAuth2 Provider, 2016. [Online; Stand 11. Juni 2016]. URL: https://auth0.com/authenticate/react-native-android/oauth2.

[3] Jason Brown. Create a map with React-Art. React Native, 2015. [Online; Stand 11. Juni 2016]. URL: http://browniefed.com/blog/create-a-map-with-react-art/.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the Gang Of Four). *Design Patterns: Elements of Reusable Object-Oriented Software*. AddisonWesley Professional, 1994.

[5] Facebook. Flux | Application Architecture for Building User Interfaces, 2015. [Online; Stand 14. Juni 2016]. URL: https://facebook.github.io/flux/docs/overview.html.

[6] Facebook. Apps using React Native. Featured Apps, 2016. [Online; Stand 16. Juni 2016]. URL: https://facebook.github.io/react-native/showcase.html.

[7] Facebook. Displaying Data. JSX Syntax, 2016. [Online; Stand 15. Juni 2016]. URL: https://facebook.github.io/react/docs/displaying-data.html#jsx-syntax.

[8] Facebook. JavaScript Environment. JavaScript Runtime, 2016. [Online; Stand 15. Juni 2016]. URL: http://facebook.github.io/react-native/releases/0.27/docs/javascript-environment.html#javascript-runtime.

[9] Facebook. JSX in Depth, 2016. [Online; Stand 13. Juni 2016]. URL: https://facebook.github.io/react/docs/jsx-in-depth.html.

[10] Facebook. Multiple Components. Data Flow, 2016. [Online; Stand 15. Juni 2016]. URL: https://facebook.github.io/react/docs/multiple-components.html#data-flow.

[11] Facebook. Native Modules. Android, 2016. [Online; Stand 15. Juni 2016]. URL: http://facebook.github.io/react-native/releases/0.27/docs/native-modules-android.html.

[12] Facebook. Native Modules. iOS, 2016. [Online; Stand 15. Juni 2016]. URL: http://facebook.github.io/react-native/releases/0.27/docs/native-modules-ios.html#native-modules.

[13] Facebook. Performance. JavaScript frame rate, 2016. [Online; Stand 15. Juni 2016]. URL: https://facebook.github.io/react-native/docs/performance.html#javascript-frame-rate.

[14] Facebook. React Native, 2016. [Online; Stand 15. Juni 2016]. URL: https://code.facebook.com/projects/450791118411445/react-native/.

[15] Facebook. React Native. A framework for building native apps using React, 2016. [Online; Stand 13. Juni 2016]. URL: https://facebook.github.io/react-native/.

[16] Facebook. The Virtual DOM, 2016. [Online; Stand 12. Juni 2016]. URL: https://facebook.github.io/react/docs/working-with-the-browser.html#the-virtual-dom.

[17] Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[18] Martin Fowler. Continuous Integration, 2006. [Online; Stand 9. Dezember 2012]. URL: http://www.martinfowler.com/articles/continuousIntegration.html.

[19] Ed D. Hardt. The OAuth 2.0 Authorization Framework, 2012. [Online; Stand 6. Dezember 2012]. URL: http://tools.ietf.org/html/rfc6749.

[20] Kevin Werbach, Dan Hunter. *For the Win: How Game Thinking Can Revolutionize Your Business*. Wharton Digital Press, 2012.

[21] Bartosz Krajka. The difference between Virtual DOM and DOM. Virtual DOM, 2015. [Online; Stand 12. Juni 2016]. URL: http://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/.

[22] Mapbox. Plans & Pricing, 2016. [Online; Stand 11. Juni 2016]. URL: https://www.mapbox.com/pricing/.

[23] Microsoft. React Native on the Universal Windows Platform, 2016. [Online; Stand 15. Juni 2016]. URL: https://blogs.windows.com/buildingapps/2016/04/13/react-native-on-the-universal-windows-platform/.

[24] Bobby Sudekum. React Native Mapbox GL. An experimental React Native component for building maps with the Mapbox iOS SDK and Mapbox Android SDK, 2015. [Online; Stand 11. Juni 2016]. URL: https://github.com/mapbox/react-native-mapbox-gl#react-native-mapbox-gl.

[25] Jürg Hunziker und Stefan Oderbolz. Gamified Mobile App für die Verbesserung von OpenStreetMap. HSR Hochschule für Technik Rapperswil, 2012. [Online; Stand 15. Juni 2016]. URL: https://eprints.hsr.ch/272/.

[26] Wikipedia. Crowdsourcing – Wikipedia, Die freie Enzyklopädie, 2012. [Online; Stand 19. Dezember 2012]. URL: http://de.wikipedia.org/w/index.php?title=Crowdsourcing&oldid=111553592.

[27] Wikipedia. Webapp – Wikipedia, Die freie Enzyklopädie, 2012. [Online; Stand 6. Dezember 2012]. URL: http://de.wikipedia.org/w/index.php?title=Webapp&oldid=102886692.

[28] Wikipedia. Document Object Model – Wikipedia, Die freie Enzyklopädie, 2016. [Online; Stand 12. Juni 2016]. URL: https://de.wikipedia.org/w/index.php?title=Document_Object_Model&oldid=151802596.

[29] Wikipedia. Framework – Wikipedia, Die freie Enzyklopädie, 2016. [Online; Stand 10. Juni 2016]. URL: https://de.wikipedia.org/w/index.php?title=Framework&oldid=153658347.

[30] Wikipedia. Grafische Benutzeroberfläche – Wikipedia, Die freie Enzyklopädie, 2016. [Online; Stand 12. Juni 2016]. URL: https://de.wikipedia.org/w/index.php?title=Grafische_Benutzeroberfl%C3%A4che&oldid=154152657.

[31] Wikipedia. Integrierte Entwicklungsumgebung – Wikipedia, Die freie Enzyklopädie, 2016. [Online; Stand 14. Juni 2016]. URL: https://de.wikipedia.org/w/index.php?title=Integrierte_Entwicklungsumgebung&oldid=153872017.

[32] Wikipedia. Minimum viable product - Wikipedia, the free encyclopedia, 2016. [Online; Stand 17. März 2016]. URL: https://en.wikipedia.org/w/index.php?title=Minimum_viable_product&oldid=706716563.

[33] Wikipedia. Paarprogrammierung – Wikipedia, Die freie Enzyklopädie, 2016. [Online; Stand 17. März 2016]. URL: https://de.wikipedia.org/w/index.php?title=Paarprogrammierung&oldid=150084864.

[34] Wikipedia. Programmbibliothek – Wikipedia, Die freie Enzyklopädie, 2016. [Online; Stand 12. Juni 2016]. URL: https://de.wikipedia.org/w/index.php?title=Programmbibliothek&oldid=152936522.

[35] Wikipedia. React (JavaScript library) – Wikipedia, Die freie Enzyklopädie, 2016. [Online; Stand 15. Juni 2016]. URL: https://en.wikipedia.org/w/index.php?title=React_(JavaScript_library)&oldid=725306667.

[36] Wikipedia. React (JavaScript library) – Wikipedia, Die freie Enzyklopädie, 2016. [Online; Stand 12. Juni 2016]. URL: https://en.wikipedia.org/w/index.php?title=React_(JavaScript_library)&oldid=723818681.

[37] Wikipedia. Singleton (Entwurfsmuster) – Wikipedia, Die freie Enzyklopädie, 2016. [Online; Stand 14. Juni 2016]. URL: https://de.wikipedia.org/w/index.php?title=Singleton_(Entwurfsmuster)&oldid=154168600.

Abbildungsverzeichnis

Tabellenverzeichnis