HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Digital Tourist Map

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

FS 2018

Autor(en):	Luca Schneider
	Leo Zurbriggen
Betreuer:	Stefan Keller
Projektpartner:	-

Abstract

Wenn ein Gast in einem Tourismusbüro oder an einer Rezeption nach Points-Of-Interest fragt, erhält er vom Tourismus-Vertreter einen gedruckten Ortsplan mit Notizen. Die Digital Tourist Map ist eine Webapplikation, die das manuelle Einzeichnen von Notizen auf einem Plan durch einen digitalen Prozess ablösen soll.

Bei Arbeitsbeginn meldet sich der Tourismus-Vertreter mit seinem OpenStreetMap-Login an. Die Webapplikation ermöglicht es ihm nun, dem Gast Auskunft zu geben und einen individuellen Plan oder Karte zu erstellen. Mittels Vorlagen können häufige Fragestellungen rasch abgerufen werden, was Zeit spart. Der Plan kann mit einem Link oder einem QR-Code an den Gast mitgegeben werden. Alternativ gibt es eine Druckfunktion, die es dem Tourismus-Vertreter erlaubt, den Plan mit hoher Auflösung via PDF auszudrucken.

Die Digital Tourist Map nutzt POI-Daten von OpenStreetMap. Die POIs bleiben damit aktuell und können bei Bedarf selber aktualisiert werden. Als Hintergrundkarte werden skalierbare Vektordaten verwendet (Mapbox Vector Tiles), dies macht Übergänge beim Bewegen und Zoomen der Karten sanft und reibungslos. Auch ist es damit einfach möglich, in Zukunft nochmals verbesserte Kartengrafiken einzubinden. Die Digital Tourist Map umfasst zudem eine Gruppenverwaltung, mit der die Organisation die Rechte ihrer Vertreter/Mitarbeiter innerhalb der Applikation verwalten kann. Die Benutzerschnittstelle unterstützt Englisch und Deutsch. Weitere Sprachen können mit geringem technischem Aufwand hinzugefügt werden. Die Applikation wurde mit React, Python und modernen Entwicklungstechniken (Git für Versionierung, Docker, Continuous-Integration und -Deployment) realisiert.

Aufgabenstellung ^

Bei dieser Arbeit geht es um die Realisierung eines digitalen personalisierten Orts- und Informationsplans (mit OpenStreetMap) für Tourist Offices oder Hotel-Front Desks. Der Plan wird mit Points-of-Interests (POIs) ergänzt und kann als Karte mitgenommen werden (gedruckt/online). Der Orstplan ist aktuell kann individuell mit POIs und Logo für die Destination (bzw. das Hotel) konfiguriert werden.

Heute bietet sich folgende Situation: Ein Hotelgast erkundet sich am Tourist Office oder am Front Desk nach einem Restaurant, einer Aktivität oder einem Geschäft (Point-of-Interest, POI). Der Tourismus-Mitarbeiter nimmt einen mehr oder weniger passenden Ortsplan hervor und zeichnet mit dem Stift Kreuze ein – von denen der Gast nachher kaum mehr weiss, was damit gemeint war.

Das geht 'digitaler', d.h. aktueller und personalisierter. Mit dieser Webapplikation (responsive, d.h. nutzbar auf PC und Tablet) kann eine personalisierte, Webkarte angeboten werden, die auf den Gast und die Destination zugeschnitten ist. Dabei wird anstelle eines „Kreuzes“ ein POI (bzw. Marker) aus einer Liste ausgewählt und auf die Webkarte gesetzt. Mit zusätzlichem Kommentar zu den POIs weiss der Gast, was gemeint war. Dadurch ergeben sich folgende Vorteile:

1. Der Ortsplan und die POIs sind lesbar.
2. Der Ortsplan enthält mehr Informationen/POIs,
3. Der Ortsplan ist aktuell (dank OpenStreetMap)

Zudem kann das Hotel die POIs ergänzen (Differenzierung) sowie den Ortsplan auf seine Lage zentrieren und mit eigener Kopfzeile versehen (Kundenbindung).

Vorgehen ^

Grobe Arbeitspakete und SW-Komponenten:

- Einarbeiten, Umgebung aufsetzen
- Use Cases, GUI Mocks, Architektur, API Design (Swagger)
- Software-Komponente Front Desk mit Workflow (Front+Backend):
  - Eingabe/Auswahl POIs und Eingabe Kommentar
  - Konfigurieren (u.a. Kopfzeile), Verwalten von Vorlagen, Eingabe individueller POIs
- Software-Komponente Schnittstellen: Übernahme/Import POIs der Destination
- Spezifikation der Import-Schnittstellen zu anderen Plattformen/Hubs ist Teil der Arbeit
- Spezifikation vom Export, Übergabe der Webkarte an den Gast ist Teil der Arbeit
- Webdienst (Software-as-a-Service) erstellen/testen/deployen

Abrenzung:

- Es wird ein vorhandener Basisplan-Style verwendet - kein angepasster Ortsplan/Situationsplan/Basiskarte.
- Keine Offline-Funktionalität.

Rahmenbedingungen ^

- Frontend: React, Javascript/Typescript, HTML5
- Backend: Python (Flask)
- Persistenz: PostgreSQL
- APIs:
  - Web API mit Flask/Swagger
  - Basiskarte Vector Tiles von *OpenMapTiles*.com
- Daten:
  - Fahrplan-API (search.ch/HAFAS oder genormteres...)
  - OSM APIs
- Sonstiges: Docker, Linux (Server)

Inhaltsverzeichnis

1 Management Summary

1.1 Ausgangslage

In der Tourismusbranche ist die Frage „Was kann man hier unternehmen?" allgegenwärtig. Normalerweise nimmt daraufhin das Hotelfachpersonal oder die Tourismusverantwortlichen eine Karte hervor und zeichnen die wichtigsten Punkte für den Gast ein. Das Ganze funktioniert reibungslos, ist schnell und effizient. Allerdings weiss der Gast nach einer gewissen Zeit meistens nicht mehr genau was welche Punkte oder Kreuze auf der Karte bedeuten. Es fehlt eine entsprechende Notiz, die mit den herkömmlichen Mitteln aufwändig aufzuschreiben ist, um den Gast daran zu erinnern was an diesem Punkt speziell ist. Ausserdem besitzen viele Kunden ein Smartphone, mit welchen sie die Informationen bekommen könnten, wenn sie einen Internetzugang besitzen, was leider meistens in den Ferien nicht gemacht wird um Geld zu sparen.

An diesem Punkt soll die Webapplikation Digital Tourist Map ansetzen. Die Applikation soll den Prozess digitalisieren. Die Einfachheit und Schnelligkeit der Erstellung dieser benutzerdefinierten Karten soll beibehalten und zusätzliche Informationen wie z.B. Rollstuhlgängigkeit oder Essensinformationen (Vegetarisch, Vegan, Koscher) angezeigt werden.

1.2 Vorgehen, Technologien

Die initiale Projektplanung wurde minimal gehalten. In einem RUP-artigen Prozess wurden Zeitbereiche für die Implementation der Hauptfeatures festgelegt. Arbeitspakete sind bestimmt und auf die verschiedenen Meilensteine geplant worden. Zusätzlich wurden alle Arbeitspakete in GitLab integriert und dort in einem Kanbanboard verwaltet. Im wöchentlichen Meeting wurden Fortschritte präsentiert, Fragen geklärt und einen Ausblick auf die Planung der nächsten Woche gegeben.

Die Applikation wird mittels Docker und GitLab CI/CD automatisch getestet und auf den HSR-Server deployt. Im Frontend wird auf React mit Redux und Flow gesetzt. Beim Backend kommen Python mit Flask und Node.js mit Express zum Einsatz. Als Datenbank wird PostgreSQL eingesetzt.

1.3 Ergebnisse

Das Resultat dieser Arbeit ist eine lauffähige Webapplikation mit einer modernen und übersichtlichen Benutzeroberfläche. Durch die Anbindung an OpenStreetMap können die benötigten Informationen auf einer Karte angezeigt werden. Die Erstellung von eigenen Kartenvorlagen ist einfach und intuitiv. Durch diese Vorlagen beschleunigte sich der Erstellungsprozess für benutzerdefinierte Karten nach Wunsch stark. Es können zusätzliche Punkte eingezeichnet oder Anmerkungen beigefügt werden. Die erstellten Karten können via Link oder QR-Code auf einem Smartphone o.Ä. angezeigt werden. Damit man Kunden ohne Smartphonezugang auch Zugang zu den Karten ermöglichen wollte, können diese auch hochauflösend ausgedruckt werden.

Die Arbeit ist nicht in produktivem Einsatz, es wurde aber alles dafür vorbereitet und dokumentiert.

1.4 Ausblick

Das Drucken der vektorbasierten Karten gestaltete sich als schwierig und könnte eventuell auch durch andere Libraries oder ein eigenes GeoJSON-Layer gelöst werden. Der ursprünglich angedachte Datenimport wurde nicht umgesetzt und die Anbindung an eine Tourismus-API könnte später interessant werden. Grundsätzlich implementiert die Anwendung die Grundfunktionalitäten und kann nach den Bedürfnissen der Kunden angepasst und erweitert werden.

2 Einleitung

2.1 Vision

Unsere Vision in einem Satz:

> Die Tourismusbranche soll digitaler werden um individuelle Beratungen für Kunden so schnell, effizient und hochwertig wie möglich durchführen zu können.

Mit dem Projekt Digital Tourist Map soll ein wichtiger Schritt in Richtung digitalem Zeitalter für die Tourismusbranche gemacht werden.

Die Kundendienstmitarbeiter am Frontdesk können damit massgeschneiderte Karten mit Empfehlungen und wichtigen Bemerkungen in wenigen Sekunden dem hilfesuchenden Gast übergeben. Sei das über die herkömmliche Methode auf ausgedrucktem Papier oder online über das eigene Smartphone. Beide Funktionalitäten können offline mitgenommen werden und helfen dem Gast bei der Entscheidung und Zielfindung von seinen Wünschen.

Die herkömmlichen Arten diese Informationen zu teilen, verursachen grosse Kosten (eigene ausgedruckte, personalisierte Karten) oder sind sehr ungenau und nicht selbsterklärend (mit Stift eingezeichnete Orte auf allgemeiner Karte). Genau dort soll unsere Software ansetzen und den Mitarbeiter unterstützen, sowie eine qualitativ hochwertige Informationsquelle für den Kunden sein.

Bei späterer Integration in Umsysteme können auch weiter Funktionalitäten hinzugefügt werden wie z.B. eine Eventintegration und somit die aktuellsten Karten für ein bestimmtes Gebiet liefern.

Die Software soll überall verfügbar sein und kostengünstig auf die Bedürfnisse der jeweiligen Nutzer zugeschnitten werden können.

2.2 Rahmenbedingungen

Besonders technisch wurden vor Beginn der Arbeit Rahmenbedingungen festgelegt.

Da es sich um eine moderne Webapplikation handelt, werden unabhängig ein Frontend sowie ein Backend implementiert, welche nur über eine gemeinsame Schnittstelle kommunizieren. Das Frontend wird mit React umgesetzt. Für das Backend kommt Python mit dem Flask-Framework zum Einsatz.

Des weiteren sollen Daten von OpenStreetMap (über eine API) bezogen und Applikationsdaten in einer PostgreSQL-Datenbank abgelegt werden.
Zum Thema Infrastruktur: Um die Applikation einfach auf beliebigen Servern installieren zu können, werden die verschiedenen Teile in eigenen Docker-Containern implementiert. Dies ermöglicht es später auch Container unabhängig voneinander zu ersetzen oder auf anderen physischen Systemen zu deployen.

3 Anforderungen

3.1 Funktionale Anforderungen

3.1.1 Karten Einzeichnen und Beschreiben

Das Einzeichnen von POIs auf Karten stellt die Kernfunktion der Applikation dar. POIs werden über die OSM API geladen und können dann vom Benutzer auf der Karte platziert werden. Die platzierten POIs können mit zusätzlichen Beschreibungen/Kommentaren versehen werden.
POIs in der Umgebung können mittels eines Filters gefunden und dann schnell auf die Karte aufgenommen werden. Der Filter dient dazu, Queries zusammenzustellen und an die OSM API zu senden. Oft getätigte Abfragen (z.B. für Hotels, Restaurants, Sehenswürdigkeiten, etc.) sollen möglichst schnell getätigt werden können.

3.1.2 Ausliefern von eingezeichneten Karten

Die Karten mit platzierten POIs und allfälligen Kommentaren müssen an Personen weitergegeben werden können. Für die Karten müssen eindeutige Links generiert werden. Diese Links führen den Anwender auf eine Kartenansicht mit POIs, die POIs können jedoch in dieser Ansicht nicht mehr verändert werden. Die Applikation wird mehrere Wege bieten den Link Personen mitzuteilen, beispielsweise mittels QR Code oder via Email.
Eine optionale Anforderung ist das Drucken der erwähnten Ansicht. Von der Applikation soll ein druckfähiges Dokument erzeugt werden können (z.B. ein PDF oder direkt in der Browser Druckansicht).

3.1.3 Personalisierung

Da die Anwendung von Hotels oder ähnlichen Institutionen eingesetzt werden soll, muss der Benutzer die resultierenden Karten personalisieren können. Es soll möglich sein ein Header-Logo einzupflegen, welches dann auf der Karte erscheint, wenn diese einem Kunden angezeigt wird.

3.2 Nicht-funktionale Anforderungen

3.2.1 Benutzbarkeit

Die Benutzbarkeit der Applikation ist sehr wichtig. Die Applikation ersetzt einen einfachen physischen Vorgang in welchem mit einem Stift Punkte auf Papierkarten eingezeichnet werden. Dieser Prozess ist schnell und einfach verständlich, weshalb unsere Applikation diese Anforderung ebenfalls erfüllen muss um sinnvoll eingesetzt zu werden.
Mittels Vorlagen (eine Art gespeicherte Auswahl von POIs) soll das Erstellen von oft gebrauchten Karten automatisiert/verschnellert werden.
Der POI Filter mit dessen Hilfe POIs gefunden und geladen werden muss effizient und intuitiv anwendbar sein. Um POIs von einfachen Kategorien einzufügen, sollten nicht mehr als drei Klicks nötig sein.

3.2.2 Portierbarkeit

Die Portierbarkeit ist insbesondere im Zusammenhang mit erzeugten Karten wichtig, welche Kunden auf möglichst vielen verschiedenen Geräten betrachten können sollen. Dies wird durch den Einsatz von Webtechnologien sowie ggf. einer druckbaren Version realisiert.

3.3 Szenarien

3.3.1 Restaurantauskunft

Ein Hotelgast erkundigt sich an der Rezeption seines Hotels über gute Restaurants in der Umgebung. Der Rezeptionist öffnet die Digital Tourist Map-Applikation über den Webbrowser. Er ist bereits mit seinem OSM-Account eingeloggt und sieht eine lokale Karte, zentriert auf das Hotel. Im Filter sucht er nach lokaler Gastronomie und erhält eine Liste. Er fragt den Hotelgast genauer nach seinen Vorlieben und klickt auf die Listeneinträge von italienischen Restaurants. Die ausgewählten Restaurants erscheinen als Punkte auf der Karte. **(UC5)**
Der Rezeptionist klickt auf Teilen. Es wird ein QR-Code angezeigt. **(UC8)** Der Hotelgast scannt den Code mit seinem Smartphone. Die Digital Tourist Map öffnet sich in seinem Browser, am oberen Bildschirmrand ist das Logo des Hotels sichtbar, es wird jedoch nur die Karte und Restaurants angezeigt, Filter sind nicht verfügbar. **(UC9)**
Der Hotelgast bedankt sich und verlässt das Hotel. Ausser Reichweite des Hotels verliert er seine Internetverbindung. Da die Karte offline verfügbar ist findet er jedoch ohne Probleme den Weg zu einer guten Pizzeria.

3.3.2 Sehenswürdigkeiten

Im Tourismusbüro erkundigt sich eine Frau nach Sehenswürdigkeiten. Ein Berater öffnet die Digital Tourist Map im Browser. Da öfter nach Sehenswürdigkeiten gefragt wird, wurde bereits eine Vorlage als Favorit gespeichert. Der Berater klickt auf die Sehenswürdigkeit-Vorlage und es erscheinen Punkte auf der Karte. **(UC7)** Die Touristin hat die Sehenswürdigkeiten rund um das Schloss bereits besichtigt, der Berater entfernt diese Punkte mit Klicks von der Karte. Er erfährt, dass die Touristin gerne Naturszenen fotografiert und fügt einen Punkt auf einem Hügel ein. Er fügt einen Kommentar hinzu in welchem er erwähnt, dass der Sonnenuntergang um 18:40 einen atemberaubenden Anblick bietet. **(UC6)**
Die Touristin hat keinen QR-Code-Scanner, tippt den Link zur Karte also manuell in ihren Browser ein. **(UC9)**
Sie schnappt sich ihre Kamera und macht sich auf den Weg.

3.3.3 Datenimport

Der Berater im Tourismusbüro hat sich ein wenig mit der Digital Tourist Map auseinander gesetzt, musste jedoch feststellen, dass einige Punkte im Filter nicht gefunden werden können. Bisher hat er diese Punkte in einer Excel-Datei gepflegt. In der Applikation findet er die Import-Funktion. Es öffnet sich ein Dateibrowser und er wählt seine Excel-Datei, bei der Bestätigung wird die Datei hochgeladen. Es werden Punkte aus der Datei gelesen und auf der Karte in der Applikation angezeigt. **(UC10)**
Später erkundigt sich ein Tourist über eine Sehenswürdigkeit. Diese Sehenswürdigkeit war bis zum Import nicht verfügbar, jetzt findet der Berater diese ganz einfach über den Filter.

3.3.4 Gruppenerstellung

Ein Tourismusbüroleiter hat einen OSM-Account angelegt und loggt sich damit in die Applikation ein. **(UC1)** Er ist noch nicht Teil einer Gruppe und wird deshalb aufgefordert, eine zu erstellen. **(UC2)** Er erstellt eine Gruppe für sein Tourismusbüro, gibt ihr einen Namen und lädt zwei Mitarbeiter zur Gruppe ein. **(UC3)** Ausserdem personalisiert er die Gruppe noch, indem er das Logo seines Tourismusbüros als Header einrichtet. **(UC4)** Er ist jetzt Gruppenadministrator.
Die Mitarbeiter loggen sich mit ihren OSM Accounts in die Applikation ein und nehmen die Einladung zur Gruppe an. Sie können jetzt damit beginnen Vorlagen zu erstellen.

3.4 Use Cases

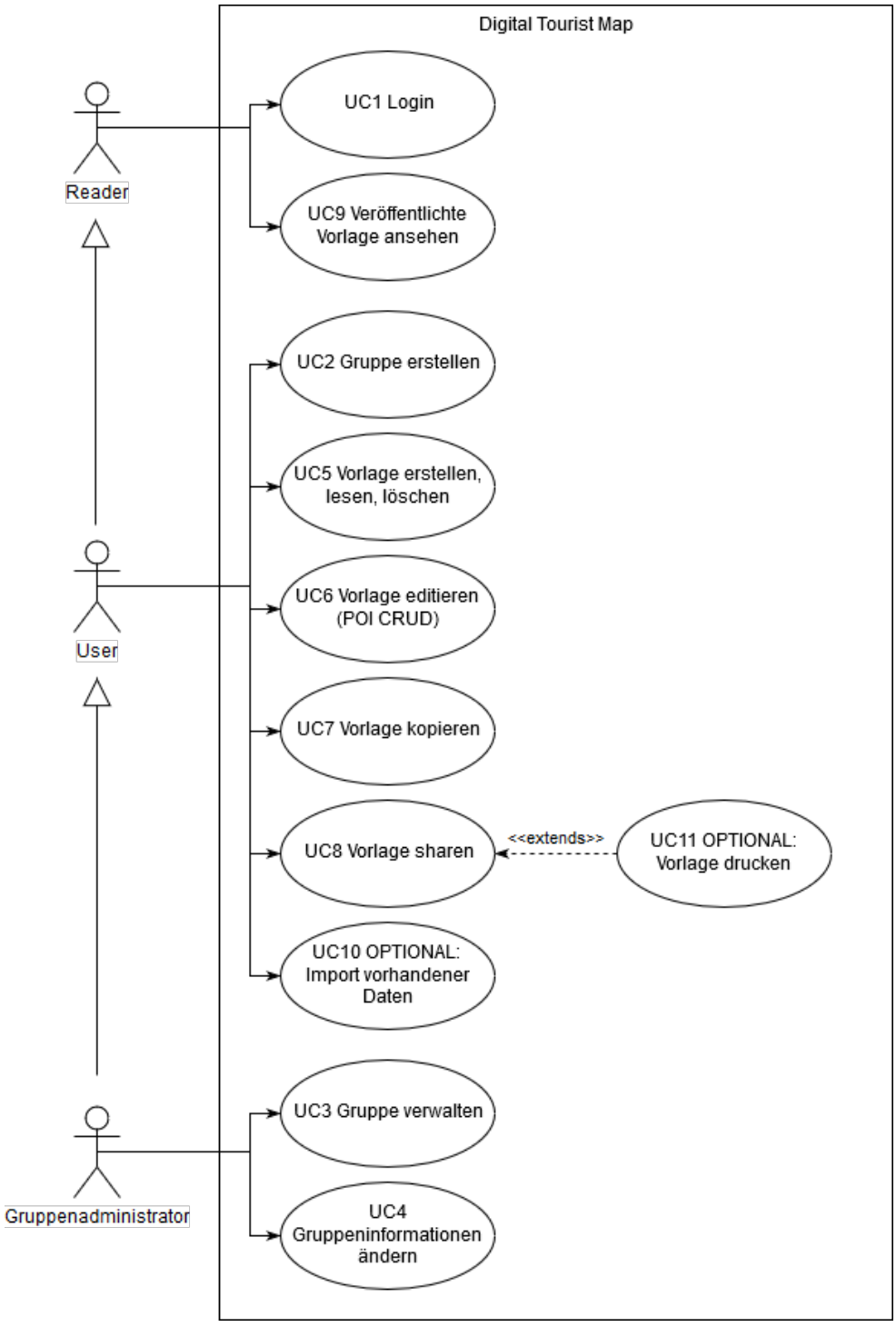

Abbildung 1: Use Case Diagramm

3.4.1 Akteure

- Reader (Anonymous)
- User (Eingeloggter Reader)
- Gruppenadministrator

3.4.2 Use Cases Brief

3.4.2.1 UC1 Login

Akteure: Reader
Beteiligte Systeme: Digital Tourist Map, OSM
Standardszenario:
Der nicht eingeloggte User (Reader) loggt sich mit seinen OpenStreeMap-Accountdetails ein und wird auf die Startseite weitergeleitet.

3.4.2.2 UC2 Gruppe erstellen

Akteure: User
Beteiligte Systeme: Digital Tourist Map
Standardszenario:
Der Benutzer möchte eine neue Gruppe erstellen. Er gibt dazu einen Namen ein und ist dann Administrator der Gruppe.

3.4.2.3 UC3 Gruppe verwalten

Akteure: Gruppenadministrator
Beteiligte Systeme: Digital Tourist Map
Standardszenario:
Der Administrator der Gruppe fügt neue Mitglieder der Gruppe hinzu und vergibt ihnen Rechte (Administrator oder User).

3.4.2.4 UC4 Gruppeninformation ändern

Akteure: Gruppenadministrator
Beteiligte Systeme: Digital Tourist Map
Standardszenario:
Der Administrator setzt den Gruppenstandort oder fügt das Firmenlogo hinzu. Die aktuelle Karte wird auf die Position zentriert.

3.4.2.5 UC5 Vorlage erstellen, lesen oder löschen

Akteure: User
Beteiligte Systeme: Digital Tourist Map
Standardszenario:
Der Benutzer kann im Editiermodus neue Vorlagen erstellen, die von ihm gespeicherten Vorlagen ansehen oder auch wieder löschen.

3.4.2.6 UC6 Vorlage editieren (POI CRUD)

Akteure: User
Beteiligte Systeme: Digital Tourist Map
Standardszenario:
Der Benutzer kann die aktuell gewählte Vorlage bearbeiten. Er wählt die POIs aus, welche er anzeigen möchte und fügt eine Beschreibung auf den Punkten ein. Es steht auch die Möglichkeit offen, verschiedene Punkte wieder zu löschen/auszublenden. Ausserdem kann er mit einem Klick auf eine freie Fläche ein neues POI hinzufügen.

3.4.2.7 UC7 Vorlage kopieren

Akteure: User
Beteiligte Systeme: Digital Tourist Map
Standardszenario:
Der Benutzer kopiert über einen Dialog eine vorhandene Vorlage mit ihren POIs und Beschreibungen in die Karte.

3.4.2.8 UC8 Vorlage sharen

Akteure: User
Beteiligte Systeme: Digital Tourist Map
Standardszenario:
Der Benutzer möchte seine Vorlage veröffentlichen. Er drückt auf den Share-Button, es wird ein Link generiert und das Dialogfeld öffnet sich mit den verfügbaren Optionen.

3.4.2.9 UC9 Veröffentliche Vorlage ansehen

Akteure: Reader oder User
Beteiligte Systeme: Digital Tourist Map
Standardszenario:
Nach dem Veröffentlichen einer Vorlage greift ein Benutzer (Reader oder User) lesend auf die Vorlage zu.

3.4.2.10 UC10 Import vorhandener Daten (optional)

Akteure: User
Beteiligte Systeme: Digital Tourist Map
Standardszenario:
Der Benutzer möchte seine eigenen Daten (Excel-Datei) in das Programm einpflegen. Er drückt auf den Import-Button und kann eine Datei auswählen. Das Programm importiert die POIs und zeigt diese in den Favoriten an.

3.4.2.11 UC11 Vorlage drucken (optional)

Akteure: User
Beteiligte Systeme: Digital Tourist Map
Standardszenario:
Der Benutzer möchte seine aktuell ausgewählte Vorlage drucken. Er drückt auf den Drucken-Button und wählt den Drucker aus. Die Karte mit den POIs und Beschreibungen wird ausgedruckt.

4 Evaluation

4.1 Frontend

4.1.1 Programmiersprache

Da das Projekt mit Webtechnologien umgesetzt werden soll, ist JavaScript die einzig mögliche Wahl. Es gibt jedoch Supersets welche zu JavaScript transpilieren. In modernen Webprojekten ist der Einsatz von Transpilern wie Babel weit verbreitet, da dies erlaubt mit neueren Sprachfeatures zu arbeiten und Polyfills bereitstellt damit der Code in verschiedenen Browsern lauffähig bleibt ohne auf den Komfort von neueren Features verzichten zu müssen. Babel ist der de facto Standard und bietet auch Module für Supersets wie JSX, TypeScript und Flow. Da React JSX einsetzt wird Babel als Transpiler benötigt. Da mit typisiertem Code bereits zur Compilezeit viele Fehlerquellen gefunden werden können, macht es Sinn, TypeScript oder Flow für Typisierung einzusetzen. Die Entscheidung fiel in unserem Fall auf Flow, da dieses JavaScript nur um Typensyntax erweitert und nicht in sich selbst eine neue Sprache ist. TypeScript benötigt auch für Third-Party-Module eine Typendefinition. Aus persönlicher Erfahrung sind Typendefinitionen für bekannte Module oft vorhanden. Es kommt jedoch vor, dass diese veraltet sind oder für kleinere/neuere Module keine Typen gefunden werden können. Wir setzen Flow hauptsächlich ein, um interne Funktionsaufrufe abzusichern.

4.1.2 React

React wurde bereits in der Aufgabenstellung als Frontend-Rendertechnologie festgelegt. React ist sehr flexibel, universal einsetz- und erweiterbar. Alternativen wären beispielsweise Angular oder VueJS. Angular ist für ein Projekt mit diesem Umfang zu gross und komplex. VueJS wäre eine Möglichkeit gewesen, hätte aber eine längere Einarbeitungszeit zur Folge gehabt.

4.1.3 State Management

React bietet State Management nur innerhalb von Komponenten. In den letzten Jahren hat sich Flux als Konzept für globales State Management etabliert. Mit Redux (bekannte Flux Implementation) wird ein globaler State durch Properties von Containerkomponenten an Renderkomponenten weitergegeben. Dies ist eine saubere Lösung und grenzt Komponenten und State sauber voneinander ab.
State-Management-Lösungen wie MobX oder Unstated wurden auch in Betracht gezogen. Da einige dieser Lösungen noch sehr jung sind und sich in diesem Bereich viel bewegt, fiel unsere Entscheidung weiterhin auf Redux, da es solide und bewährte Lösungen für unsere Probleme bietet.

4.2 Backend

4.2.1 Flask vs Django vs Pyramid

4.2.1.1 Flask Flask ist ein Mikrowebframework basierend auf Werkzeug und Jinja2-Templates. Es ist OpenSource und BSD lizensiert. Flask wird von vielen Firmen eingesetzt u.A. Netflix, Pinterest und LinkedIn.

Flask ist ein Mikroframework weil es keine konkreten Tools oder Libraries benötigt. Es hat keine Datenbankschicht, Formularvalidierung oder andere Komponenten, welche man üblicherweise in third-party Libraries findet. Jedoch unterstützt Flask die Integration von Erweiterungen, die so in Flask integriert werden können, als ob sie von Anfang an in Flask selber implementiert worden sind. Für diese Komponenten gibt es eine grosse Auswahl an ORMapper, Formularvalidierung und vielen anderen Anwendungsgebieten.

Die aktuelle Version von Flask ist 1.0.2 und wurde am 02.05.2018 zuletzt aktualisiert. Demnach ist diese Version sehr aktuell.

Vorteile:

1. Klein
2. Man hat nur das drin, was man braucht
3. Für kleinere Projekte/Prototypen

Nachteile:

1. Höherer Integrationsaufwand von zusätzlichen Komponenten
2. Schlechte Async Möglichkeiten

4.2.1.2 Django Django ist Fullstack-Webframework, welches auf dem Pattern Model-View-Template basiert. Auch Django ist OpenSource und BSD lizensiert. Django wird von vielen Firmen eingesetzt u.A. Instagram, Sentry und Mozilla.

Das Hauptziel von Django ist die einfache Erstellung von komplexen, datenbankbasierten Webseiten. Daher kommt Django auch mit vielen vorinstallierten Komponenten:

- ORMapper
- Djangomodelle
- Templating
- Controller für HTTP
- Formularvalidierung
- Cachingframework
- Möglichkeit für Middlewares
- Internationalisierung

Django ist hauptsächlich für grössere Projekte gedacht und liefert daher schon viel out-of-the-box mit. Die aktuelle Version ist 2.0.5 und wurde zuletzt am 02.05.2018 aktualisiert. Das heisst auch Django ist sehr aktuell.

Vorteile:

- Umfangreich
- Gute Dokumentation

Nachteile:

- Vieles vorgegeben
- Zu "monolithisch"

4.2.1.3 Pyramid Pyramid ist ein minimales Webframework, welches klein anfängt und gross aufhören kann. Auch Pyramid ist OpenSource und BSD lizensiert. Pyramid wird nicht im gleichen Ausmass wie Django oder Flask von Firmen eingesetzt. Es wird z.B. von Mozilla, SurveyMonkey und NextGIS eingesetzt.

Pyramid ist für kleinere Projekte gedacht, die irgendwann grösser werden. Dazu gibt es viele third-party-Module die einfach eingebunden werden können. Pyramid besitzt grundsätzlich schon Routing und Authentifizierung, für das Templating und die Datenbankanbindung braucht es aber die externen Module.

Die aktuelle Version von Pyramid ist 1.9.2 und wurde zuletzt am 23.04.2018 aktualisiert. Auch Pyramid ist somit sehr aktuell.

Vorteile:

- Flexibilität
- Testing Fokus
- Async-Möglichkeit

Nachteile:

- Höherer Integrationsaufwand
- Routing out-of-the-box zu schwach
- Kein Kontext wie Flask

4.2.1.4 Entscheidung Schlussendlich haben wir uns für Flask entschieden, auch wenn wir die eingebaute Template-Engine von Flask eigentlich nicht brauchen, da Django zu umfangreich für unser Projekt wäre und wir wirklich nur das Nötigste einbinden wollen. Pyramid wäre die Alternative, aber Flask hat die grössere Community und man findet schneller Hilfe und Beispiele als bei Pyramid.

4.2.2 Overpass vs Openpoiservice

4.2.2.1 Overpass Overpass ist eine read-only API, welche bestimmte benutzerspezifische Daten aus OpenStreetMap sendet. Overpass hat eine sehr mächtige Query-Sprache, welche das Abfragen der vielen verschiedenen Daten erleichtern soll. Es gibt mehrere frei verfügbare Services in verschiedenen Teilen der Welt.

Mittels der Queries können alle Nodes aus der Bounding Box, welche den Tag-Key `tourism` besitzen, ausgegeben werden:

```
[out:json];
node["tourism"]({{bbox}});
out geom tags;
```

Abbildung 2: Overpass QL

und die Antwort von Overpass:


```
{
   "elements":[
      {
         "type":"node",
         "id":392055782,
         "lat":47.2224200,
         "lon":8.8171293,
         "tags":{
            "amenity":"bbq",
            "fireplace":"yes",
            "name":"Fäschtinseli",
            "tourism":"picnic_site"
         }
      }
   ]
}
```


Abbildung 3: Overpass Antwort

Das `out geom tags` gibt bei Ways automatisch die Geometrien der einzelnen Nodes zurück, so dass es für uns leichter ist GeoJSON zu bilden. Overpass selber kann kein richtiges GeoJSON, da die Spezifikation nicht genau mit den OSM-Datentypen umsetzbar ist.

Unsere Lösung beruht auf der Abfrage (Node, Way, Relation) von Overpass und der eigenen Konvertierung zu GeoJSON siehe dazu 8.2.3.2.

4.2.2.2 Openpoiservice Openpoiservice ist ein Open-Source-Projekt der GIScience Research Group der Universität Heidelberg. Das Ziel ist es Points-of-Interest für eine bestimmte OSM-Geometrie zu finden. Das Ganze ist mit Flask geschrieben und importiert die OSM-Tags der verschiedenen Datentypen und versucht sie zu parsen. Der Service ordnet den Tags verschiedene IDs zu, welche in einer Datei aufgelistet sind.

Man kann das Ganze nach dem Import querien. Der Service gibt GeoJSON zurück, welches mit den Kategorie-IDs unter `properties` angereichert worden ist.

Beispielantwort:


```
{
   "type":"FeatureCollection",
   "features":[
      {
         "type":"Feature",
         "geometry":{
            "type":"Point",
            "coordinates":[
               8.817129300000056,
               47.22241999999981
            ]
         },
         "properties":{
            "osm_id":392055782,
            "osm_type":1,
            "distance":0.0,
            "category_ids":{
               "625":{
                  "category_name":"picnic_site",
                  "category_group":"tourism"
               }
            },
            "osm_tags":{
               "name":"F\u00e4schtinseli"
            }
         }
      }
   ]
}
```


Abbildung 4: Openpoiservice Antwort

Wie man sieht, wurde die Kategorie 625, welche für die Kategoriegruppe Tourismus mit Unterkategorie Picknickstelle steht, ausgewählt und zurückgegeben.

Dieser Service bietet eigentlich genau die Funktionalität, die wir selber über Overpass erreichen und könnte anstatt Overpass direkt gebraucht werden. Allerdings muss man mit den vorgefertigten Kategorien (`accomodation, animals, arts_and_culture, education,facilities facilities, financial, healthcare, historic, leisure_and_entertainment, natural, public_places, service, shops, sustenance, transport, tourism`) leben oder einen eigenen Openpoiservice mit den vorhandenen Docker-Images aufsetzen.

5 UI Design

Vor der Entwicklung des Frontends mussten die Views wie auch die Abläufe im User-Interface geplant und definiert werden. Die Szenarien und Use-Cases bildeten die Grundlage, auf welcher die groben ersten Entwürfe gezeichnet wurden. Im ersten Schritt wurden die Views und deren benötigte Elemente auf Papier geplant.

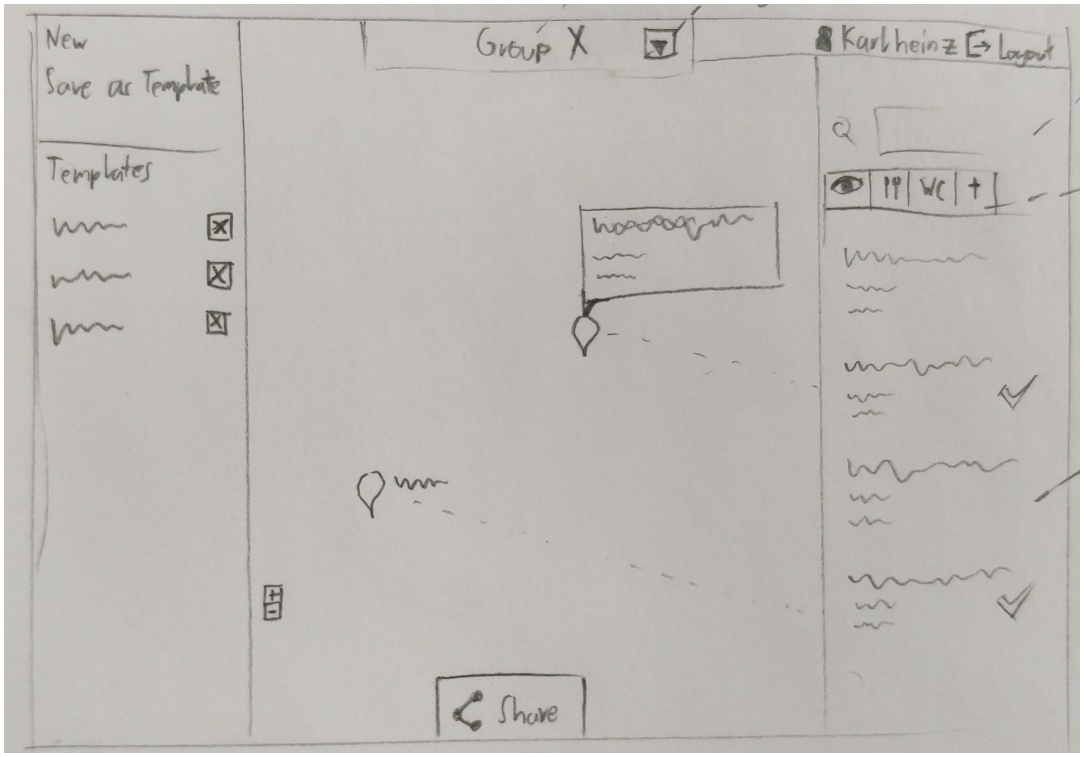

Abbildung 5: Der erste Entwurf der Map Edit View

Nach diesem ersten Schritt wurden die Entwürfe digital verfeinert. Hier wurden Farben und Schriften, Abstände, Icons und Dimensionen genauer geplant. Da hier ebenfalls alle Views geplant wurden, konnten anhand dieser Entwürfe die meisten benötigten UI-Elemente identifiziert werden.

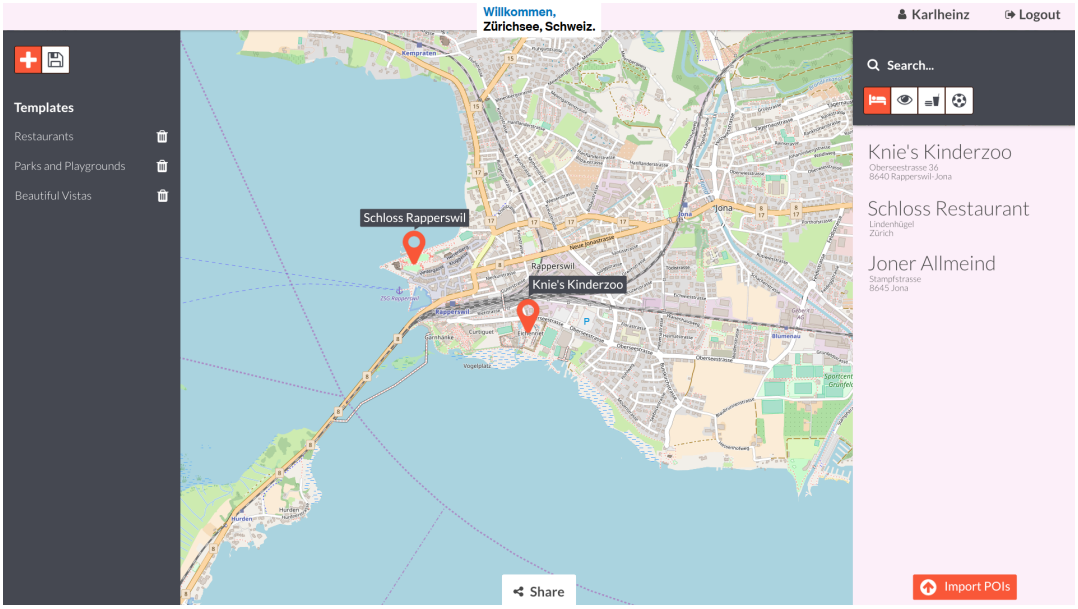

Abbildung 6: Digitaler Entwurf Map Edit View

Anhand dieser Entwürfe wurden die Views dann tatsächlich mit React umgesetzt. Da sich teilweise Anforderungen geändert haben, neu priorisiert wurden oder ganze neue Features hinzugefügt wurden, entspricht das Endresultat nicht eins-zu-eins den geplanten Views.

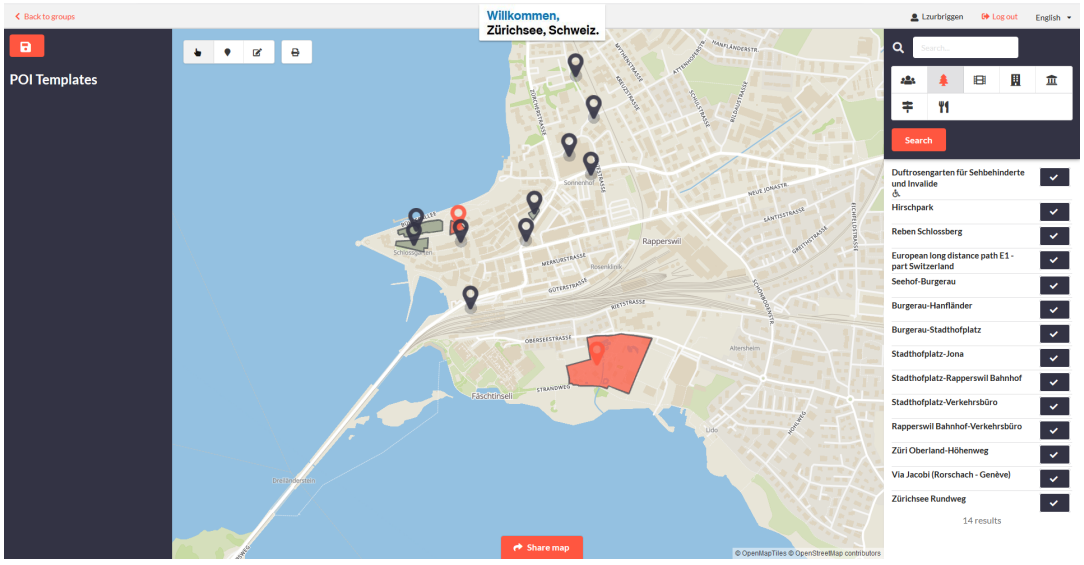

Abbildung 7: Endresultat Map Edit View

6 Technologien

6.1 React

React ist eine JavaScript UI-Library. React bietet von sich aus kaum mehr als Renderinglogik. Die JavaScript-Spracherweiterung JSX dient in fast jedem React-Projekt als eine Art Templatesprache. In idiomatischem React wird die ganze Applikation aus Komponenten aufgebaut. Jede dieser Komponenten wird als Klasse oder Funktion implementiert und kann wie ein HTML-Tag in die Applikation eingebaut werden, unabhängig davon, ob die Komponente selber etwas visuell darstellt oder nur etwa Daten verwaltet und an andere Komponenten weitergibt. Dies macht React extrem flexibel, Komponenten von Dritten können meist problemlos und ohne viel Boilerplate-Code mit eigenen kombiniert werden.
Der zweite grosse Vorteil von React ist, dass sich das UI automatisch neu aufbaut, sobald sich Daten ändern. Das dargestellte UI repräsentiert so immer die darunter liegenden Daten und man muss sich keine Gedanken über Stateübergänge oder One-/Two-Way-Binding machen.

Um React als Alternative zu beispielsweise Angular oder Vue einsetzen zu können, müssen zusätzliche Module installiert werden. So wurden in diesem Projekt für globalen State/Datenverwaltung im Frontend das Modul Redux eingesetzt. Für das Handling von Routen und Navigation bietet sich React-Router an.

6.1.1 Redux

Redux ist eines der verbreitetsten Module, welches das Flux-Konzept implementiert. Mit Flux wird zunächst ein `store` angelegt. Dieser beinhaltet alle Daten, welche global in der Applikation verfügbar sein sollen. Zusätzlich zu diesem Store werden Reducer, welche den Store manipulieren, und Actions, welche den Reducern Befehle erteilen, angelegt. In Komponenten wird definiert, welche Daten vom Store benötigt werden und rendern automatisch neu, sobald sich die relevanten Daten ändern.
Daraus ergibt sich ein unidirektionaler Datenfluss: Komponenten lösen Actions aus, die Reducer ändern den State und durch diese Datenänderung werden Komponenten neu dargestellt.

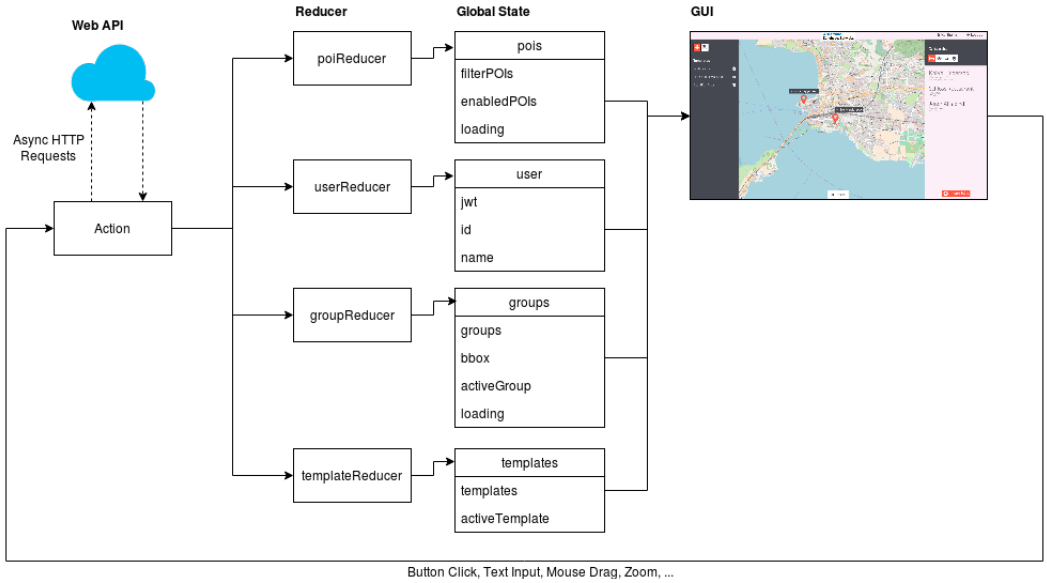

Abbildung 8: Flux

6.1.2 React-Router

React-Router bietet die Möglichkeit im Frontend über Links zu verschiedenen Views zu navigieren, wie es bei traditionellen Webseiten möglich ist. Da React aber für Single-Page-Applikationen eingesetzt und bei einfacher Navigation keine Kommunikation mit einem Server stattfindet, wird dieses Verhalten komplett mit JavaScript implementiert. Die Routerkomponente wird üblicherweise fast als erste Komponente gerendert. Sie beinhaltet dann Routerkomponenten welche ihrerseits Viewkomponenten rendern. Der Router entscheidet dann anhand einer `history`, welche der Routen und somit welche View dargestellt werden soll.
Mit etwas zusätzlichem Aufwand können so auch Routen gesperrt werden, welche für Benutzer ohne Login nicht ersichtlich sein sollen.

```
1 class App extends Component {
2   render() {
3     return (
4       <Router history={history}>
5         <Switch>
6           <PrivateRoute path="/groups" component={GroupsView} />
7           <PrivateRoute path="/grouppreferences/:id" component={GroupPreferencesView} />
8           <PrivateRoute path="/editmap/:groupId" component={MapEditView} />
9           <Route path="/map/:source" component={MapViewContainer} />
10          <Route path="/authsuccess" component={AuthSuccessContainer} />
11          <Route path="/" component={WelcomeViewContainer} />
12        </Switch>
13      </Router>
14    );
15  }
16 }
```

Abbildung 9: Beispiel mit React-Router

6.1.3 Mapbox-Style

Die Grundlage für das Anzeigen von Karten im Frontend bietet ein sogenannter Mapbox-Style. Ein Style wurde als JSON-Objekt spezifiziert und definiert das Aussehen und weitere Eigenschaften der Kartenelemente. Etwas verwirrend kann sein, dass der Style ebenfalls die Quellen für Tiledaten beinhaltet. Vektortiles werden in Digital Tourist Map von Openmaptiles.com/Tilehosting.com geladen. Tilehosting.com bietet einen kostenlosen Plan mit einem Mapbox-Style und Tile-Quelle für bis zu hunderttausend Requests pro Monat. Für kommerzielle Nutzung müsste auf einen kostenpflichtigen Plan oder einen anderen Anbieter umgestellt werden.

6.2 Technologien Backend

6.2.1 OAuth

6.2.1.1 Was ist OAuth?

OAuth steht für Open Authorization und ist ein offenes Protokoll, das eine standardisierte, sichere API-Autorisierung erlaubt. Der Benutzer kann mit Hilfe dieses Protokolls einer Anwendung (third-party) den Zugriff auf seine Daten erlauben, welche von einer externen Anwendung stammen, ohne dabei geheime Details seiner Zugangsberechtigung für die externe Anwendung der third-party-Anwendung zu übermitteln. Der Benutzer kann so Dritte damit beauftragen, in seinem Namen einen Dienst zu konsumieren. Typischerweise wird dabei die Übermittlung von Passwörtern an Dritte vermieden. Auch kann so auf das Login-System des Dienstes zugegriffen werden, ohne dabei ein eigenes Login mit anderem Passwort auf der third-party-Applikation verwenden zu müssen.

Die Idee für OAuth kam 2006 auf, als Twitter die OpenID-Implementierung weiter vorantreiben wollte. Am 3. Oktober 2007 wurde OAuth Core 1.0 veröffentlicht. Nachfolgend wurde eine OAuth-Arbeitsgruppe in der IETF eingerichtet.

6.2.1.2 Version: 1.0

2009 wurde eine Session-Fixation-Attacke gefunden, welche auf alle OAuth 1.0 Provider angewandt werden konnte. Deshalb wurde mittels der Version 1.0 Revision A dieses Problem adressiert und behoben. Für alle Informationen betreffend Funktionsweise siehe 6.2.1.3. Nachfolgend wird der Angriffsvektor beschrieben und die Änderung, welche mit Revision A umgesetzt worden ist, erläutert.

Angriffsvektor "The basic idea is that an attacker tricks an application using an OAuth API (a Consumer) to give it access to someone else's resources (via an Access Token). The attacker never gets the Access Token itself, just the ability to use the application. But since the application has the Access Token, the attacker can use it to access the victim's resources." [1]

Mechanismus normal Der Benutzer startet den Prozess, indem er die Applikation anweist auf seine Ressourcen zuzugreifen. Die Applikation fragt ein Request-Token vom Provider (Google etc.) an und leitet den Benutzer an diesen Provider für die Autorisierung weiter. Der Benutzer loggt sich in diesen Provider ein und gewährt der Applikation Zugriff. Nach dem Erlauben des Zugriffs wird der Benutzer wieder auf die Applikation zurück geleitet um den ganzen Vorgang zu beenden und die Applikation kann auf die Benutzerdaten zugreifen.

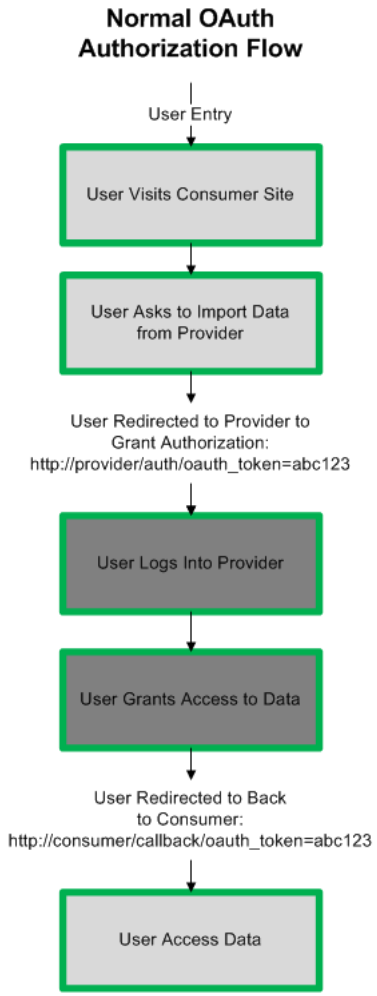

Abbildung 10: Normaler Authorization-Flow [1]

Angriff Der Angreifer startet den Prozess, indem er die Applikation benutzt und die Frage nach dem Zugriff startet. Dadurch erhält die Applikation ein Request-Token und leitet den Angreifer zum Provider weiter. An diesem Punkt stoppt der Angreifer und notiert sich das Token, sowie den Callback (Beides Query-Parameter). Der Angreifer nutzt einen Trick um das Opfer auf diesen Link des Providers umzuleiten (Social Engineering). Das Opfer sieht nun die Seite des Providers welche um Erlaubnis bittet für die Applikation. Der Benutzer loggt sich ein und gewährt der Applikation Zugriff. Selbst für erfahrene Nutzer ist diese Methode schwer zu erkennen. Sobald der Benutzer Zugriff gewährt hat, kann der Angreifer den Callback rekonstruieren und auf die Applikation zurückkehren (der eigentliche Callback des Angreifers). An diesem Punkt ist der Account des Angreifers mit den Daten des Benutzers verlinkt. Die Applikation könnte dann die Daten des echten Benutzers auslesen.

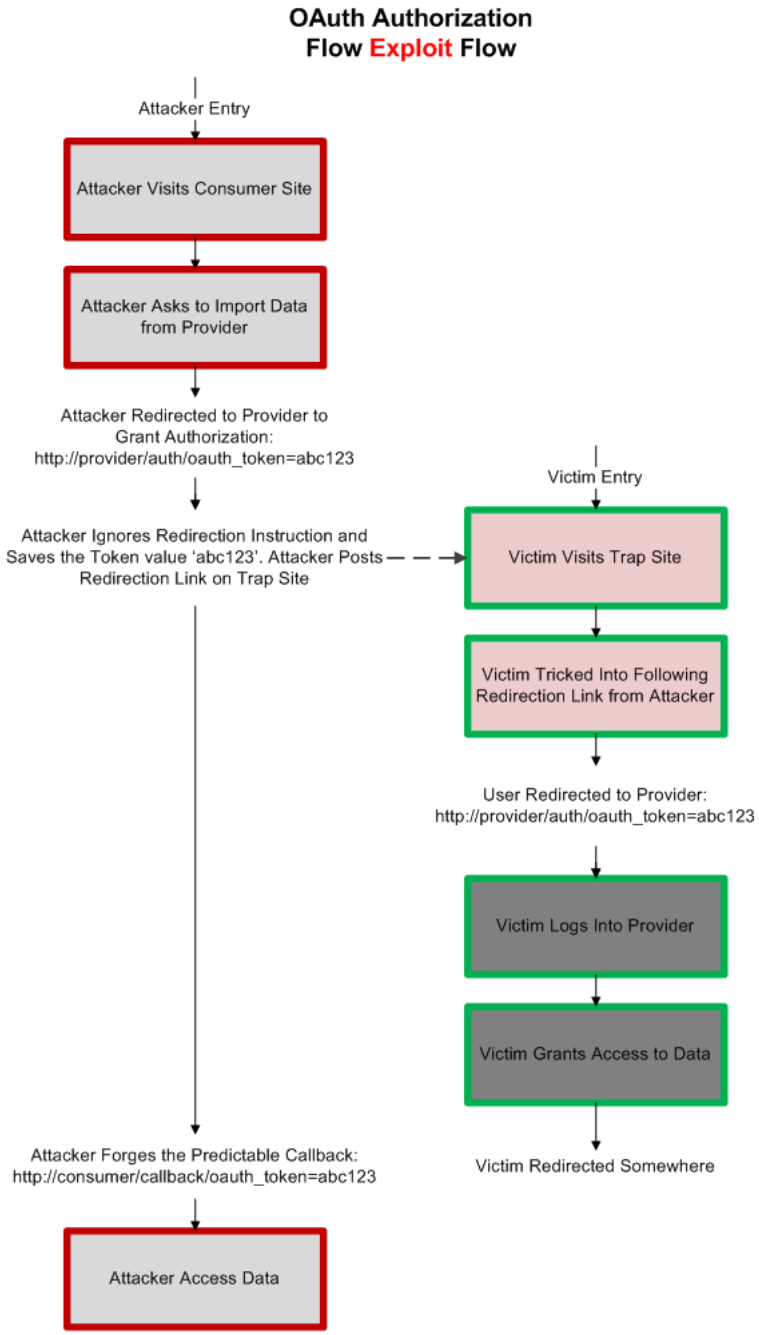

Abbildung 11: Exploit Authorization-Flow [1]

Lösung mit den Änderungen von OAuth 1.0 zu OAuth 1.0a: [2]

1. Applikation fragt den Provider für ein Request-Token an. Änderungen in der Anfrage (Header):
   - `oauth_callback: OPTIONAL`. URL zu welcher der Benutzer zurückgeleitet wird, falls die Autorisierung abgeschlossen wurde.
2. Der Provider erstellt ein Unauthorized-Request-Token und muss den Callback eindeutig diesem Request-Token zuweisen können (entweder durch den Local-Storage oder durch Verschlüsselung der Callback-URL im Request-Token). Die Antwort des Providers enthält unter anderem (Änderung):
   - `oauth_callback_accepted=1`
3. Die Applikation sieht, dass der Provider den `oauth_callback_accepted` Header gesetzt hat. Den Callback hier zu setzen wird damit obsolet. Danach leitet die Applikation den Benutzer an den Provider weiter:
   - `oauth_callback: ENTFERNT`
4. Der Provider leitet den Benutzer wieder zurück zur Callback-URL. Das passiert aber nur, wenn ein Callback in Schritt 1 mitgegeben worden ist. In dieser Anfrage unteranderem enthalten:
   - `oauth_verifier: ZWINGEND NÖTIG`. Der oauth_verifier muss mit der OAuth-Applikation sowie dem Request-Token assoziierbar sein.
5. Die Applikation fragt den Provider für ein Access-Token an. Unter anderem in dieser Anfrage:
   - `oauth_verifier: ZWINGEND NÖTIG`. Der Verifizierer, welcher man in Schritt 5 erhalten hat. Der Provider muss nun überprüfen, dass dieser Verifizierer ursprünglich für diese Applikation und Request Token gebildet worden ist.

6.2.1.3 Version: OAuth 1.0a [3]

Grundsätzlich wird zwischen 1-legged (1 Request an den Provider), 2-legged (2 Requests an den Provider) und 3-legged (3 Requests an den Provider) unterschieden.

1–legged Bei nur einem Request kann man vom Benutzer keine Autorisierung verlangen. Das Ganze läuft eher als eine Art Authentifizierung der Applikation am Provider. Entweder kann der Provider entscheiden er will keine benutzerrelevanten Daten an die Applikationen weiterleiten oder alle haben Zugriff auf alles, was aber nicht sehr klug wäre wenn personenbezogene Daten enthalten sind. Daher sollte man eher auf die "normale" Form von OAuth 1.0a (3-legged) bauen. Zwei optionale Header regeln die Versionsverwaltung und den Benutzerzugriff.

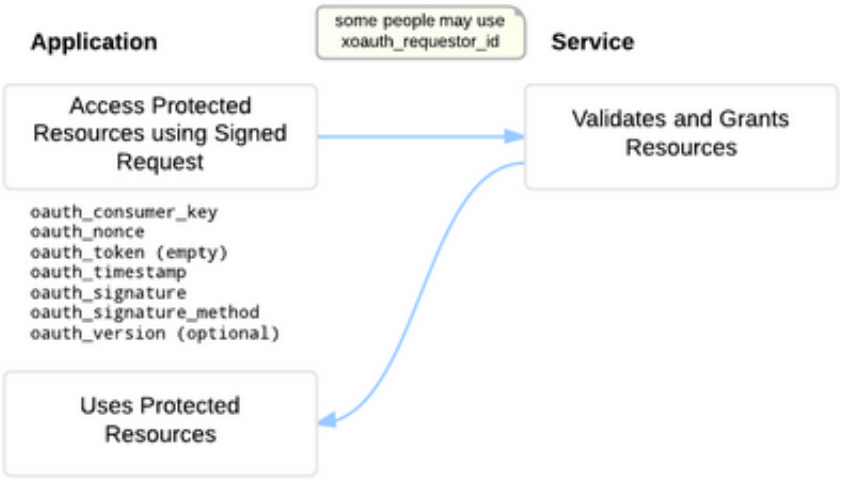

Abbildung 12: OAuth 1.0a 1-legged [3]

2-legged Die am seltensten verwendete Variante um OAuth 1.0a zu implementieren ist die 2-legged Variante.

1. Die Applikation schickt eine signierte Anfrage an den Provider (ähnlich zu 1-legged).

2. Der Provider schickt danach ein Request-Token (kurzlebig, nur für diesen Request) an die Applikation, eventuell können auch mehr Informationen mitgeschickt werden.

3. Die Applikation schickt das Request-Token signiert an den Provider.

4. Der Provider schickt ein Access-Token (langlebig) an die Applikation zurück (ähnlich zu Schritt 2).

5. Die Applikation kann nun Anfragen mit dem Access-Token verschicken.

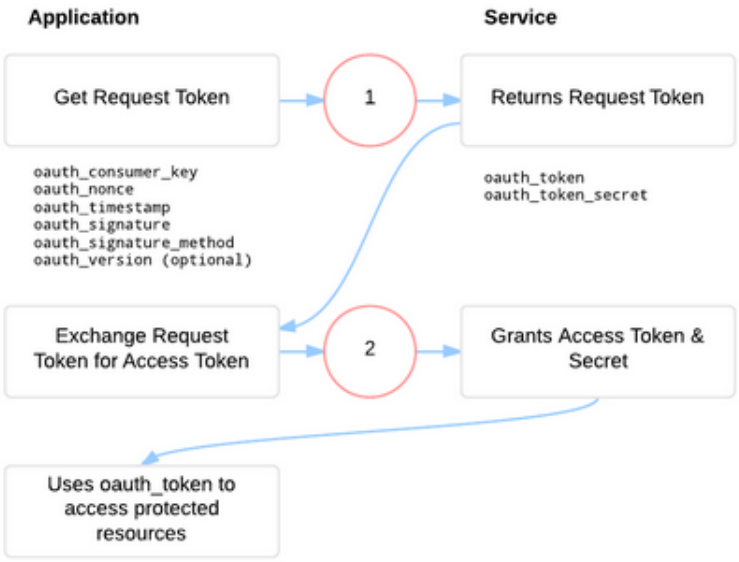

Abbildung 13: OAuth 1.0a 2-legged [3]

Der Benutzer wurde auch bei dieser Variante nicht miteinbezogen und hat eigentlich die gleichen Nachteile wie die 1-legged Variante und zusätzlich ist es noch komplizierter. Das wäre auch der Grund warum diese Variante nicht weit verbreitet ist.

3-legged Die am häufigsten verwendete Variante für OAuth 1.0a, welche auch von OpenStreetMap verwendet wird.

1. Die Applikation schickt eine signierte Anfrage an den Provider (ähnlich zu 1-legged und 2-legged) mit einem zusätzlichen Header für eine Callback-URL.

2. Der Provider schickt ein Request-Token (kurzlebig, nur für diesen Request) zurück an die Applikation (ähnlich zu 2-legged) mit einem zusätzlichen Header für die Callback-Bestätigung.

3. Die Applikation leitet den Benutzer an den Provider weiter. Bei dieser Weiterleitung wird das Request-Token mitgeschickt (ohne Secret).

4. Der Provider fragt den Benutzer nach Erlaubnis.

5. Der Benutzer erteilt der Applikation Erlaubnis.

6. Der Provider leitet den Benutzer wieder an die Applikation weiter. Dieses Mal erhält die Applikation einen Verifier. Die Weiterleitung erfolgt an die Callback-URL von Schritt 1.

7. Die Applikation schickt eine signierte Anfrage mit dem Access-Token und dem Verifier an den Provider.

8. Der Provider schickt ein Access-Token (langlebig) and die Applikation zurück (ähnlich zu 2-legged).

9. Die Applikation kann nun Anfragen mit dem Access-Token verschicken.

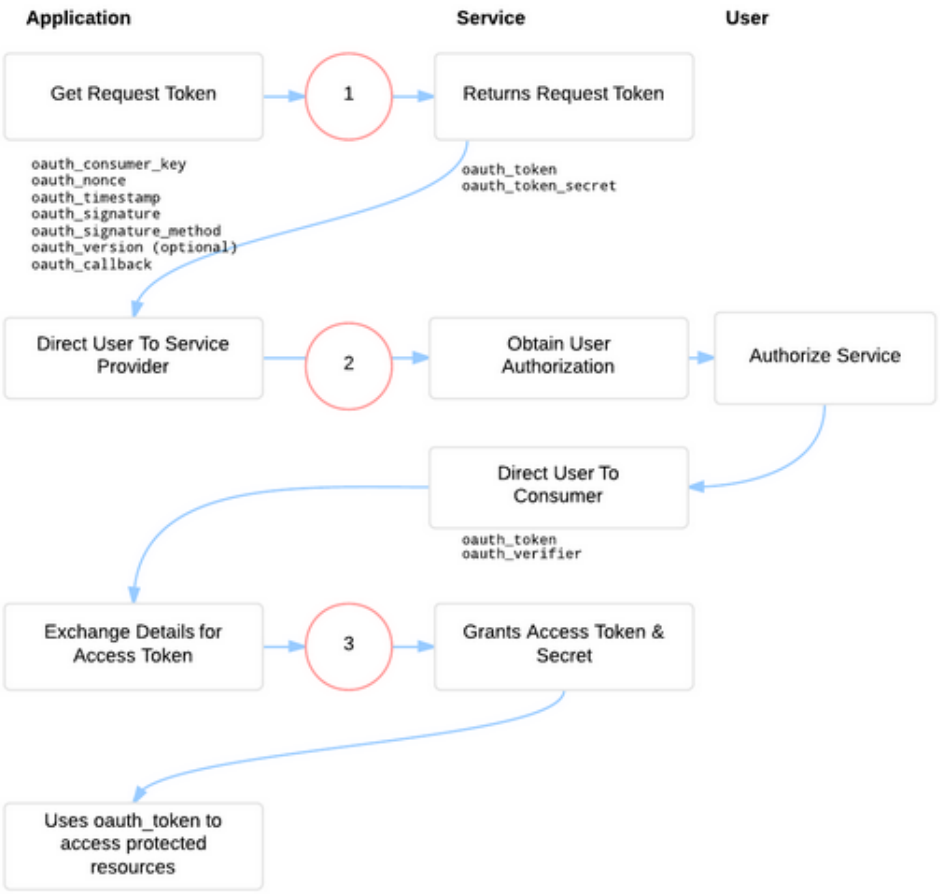

Abbildung 14: OAuth 1.0a 3-legged [3]

Diese Variante ist die sicherste der drei, weil der Benutzer in den Handshake miteinbezogen wird.

6.2.1.4 Version: OAuth 2.0 [3]

OAuth 2.0 ist für die Einfachheit während der Integration konzipiert worden und spezifiziert einen Autorisierungsfluss für Web-, Desktop-, Mobileapplikationen und viele mehr. Es gibt grundsätzlich drei verschiedene Varianten.

2-legged Die am einfachsten zu erklärende Variante.

1. Die Applikation schickt eine Anfrage an den Provider mit einem Autorisierungs-Header, welcher die Client-ID und das Client-Secret enthält.

2. Der Provider schickt ein Access-Token zurück, welcher bis zur Ende der Gültigkeit genutzt werden kann.

3-legged Das normale Verfahren für OAuth 2.0.

1. Die Applikation leitet den Benutzer an den Provider weiter und schickt dabei die Client-ID, die Redirect-URI und den Response-Type als Query-Parameter mit.

2. Der Benutzer meldet sich an und gibt der Applikation Zugriff auf die Daten.

3. Der Provider leitet den Nutzer an die im Schritt 1 spezifizierte Redirect-URI weiter. Dabei schickt der Provider einen Code und einen State mit.

4. Die Applikation tauscht den Code gegen ein Access-Token aus. Zusammen mit der Client-ID dem Client-Secret und einer Redirect-URI wird die Anfrage gesendet.

5. Wenn die ID und das Secret richtig sind, wird der Provider die Callback-URI mit einem Access-Token aufrufen.

6. Die Applikation speichert das Token und kann es für weitere Aufrufe verwenden.

Refresh-Token Bei OAuth 2.0 hat der Access-Token meistens eine begrenzte Gültigkeit. Deshalb gibt es einen speziellen Endpoint um die Access-Tokens aufzufrischen.

1. Die Applikation schickt dem Provider eine Anfrage an die Refresh-Token-URI. Dabei gibt er das Token, die Client-ID und das Secret mit.

2. Der Provider validiert die Anfrage und schickt ein neues Access-Token mit.

6.2.1.5 Unterschied zwischen den Versionen

OAuth 1.0a Von OAuth 1.0a glaubt man, dass es sicherer aber weniger flexibel und komplizierter ist als OAuth 2.0. Es basiert auf Shared-Secrets um die Signatur zu berechnen. Das erlaubt die Verifikation der Authentizität der API-Aufrufe. Es braucht einen zusätzlichen Schritt im Vergleich zu OAuth 2.0. Die Applikation braucht einen Request-Token, welcher gegen einen Access-Token ausgetauscht werden kann. Ein entscheidender Vorteil von OAuth 1.0a ist, dass die Token-Secrets nie direkt übertragen werden. Das macht das Mitbekommen in der Weiterleitung des Traffics unmöglich. Das Ganze funktioniert auch ohne SSL und sollte dann eingesetzt werden, wenn die Daten sehr sensitiv sind. Allerdings bedeutet diese Signatur und die zwei Tokens einen Mehraufwand für den Provider.

OAuth 2.0 OAuth 2.0 braucht keine Signaturen und basiert auf SSL, darum glaubt man auch, dass es weniger sicher aber einfacher zu implementieren ist. Das Problem mit SSL ist, dass die Verbindung auch dann funktionieren kann, wenn die Verifikation des Zertifikats fehlschlägt. [4]

6.2.2 Flask

6.2.2.1 Übersicht Flask ist ein Mikrowebframework in Python. Das Mikro bedeutet, dass Flask versucht den Core simpel und erweiterbar zu halten. Es versucht nicht irgendetwas spezifisches aufzuzwingen und alle wichtigen Entscheidungen sind dem Benutzer überlassen. Flask unterstützt die Einbindung weiterer Funktionalitäten, so als ob sie schon immer darin vorhanden gewesen wären.

6.2.2.2 Hello World Eine minimale Flask-Applikation sieht so aus:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route('/')
5 def hello_world():
6     return 'Hello, World!'
```

Abbildung 15: Hello World Flask

Es wird die Flask-Klasse importiert und instanziert. Der Name der Applikation wird mittels `__name__` weitergegeben. Danach braucht man den Route-Decorator um Flask mitzuteilen, welche Funktion ausgeführt werden soll, wenn die URL aufgerufen wird. Die `hello_world`-Funktion gibt einen String zurück, der dann dem Benutzer angezeigt wird.

Um das Ganze laufen zu lassen, gibt man folgende Befehle ein:

```
$ export FLASK_APP=hello.py
$ python -m flask run
 * Running on http://127.0.0.1:5000/
```

Abbildung 16: Flask laufenlassen

6.2.2.3 Routing Flask verwendet einen Routing-Decorator um die URLs zu definieren.

```
@app.route('/hello')
def hello():
    return 'Hello, World'
```

Abbildung 17: Routing Decorator

Man kann auch variable Routen erzeugen, die den Datentyp o.Ä. bestimmen. Mittels `<Converter:Variable_name` werden die Routen variabel und man muss den Variablennamen auch in der Funktion verwenden.

```
@app.route('/user/<username>')
def profile(username):
    return '{}\'s profile'.format(username)
```

Abbildung 18: Variables Routing

Es gibt folgende vordefinierten Converter:

- string: Akzeptiert jeden Text ohne /
- int: Akzeptiert eine positive Ganzzahl
- float: Akzeptiert eine positive Gleitkommazahl
- path: Akzeptiert das Gleiche wie `string` aber auch mit /
- uuid: Akzeptiert UUID strings

Trailing-Slashes in den Decorators werden als andere Route verstanden. `/home` ist nicht das Gleiche wie `/home/`.

Mittels `url_for` kann die Route der Funktion dynamisch zusammengebaut werden. Dies ermöglicht einfaches Testen und nach einer Änderung der URL sollte dies ohne Anpassungen weiter möglich sein.

```
1 print(url_for('profile', username='John Doe'))
2 #/user/John Doe
```

Abbildung 19: Dynamisches Routing

Standardmässig wird die GET-Route durch den Decorator angesteuert. Will man das ändern oder erweitern benutzt man den `methods`-Parameter.

```
1 @app.route('/login', methods=['GET', 'POST'])
2 def login():
3     if request.method == 'POST':
4         return do_the_login()
5     else:
6         return show_the_login_form()
```

Abbildung 20: URL Methoden definieren

6.2.2.4 Request Data

Flask besitzt ein globales Request-Objekt, welches für den Gebrauch der Parameter der Anfrage gebraucht wird. Es ist kontextlokal, das heisst es ist eigentlich global, wird aber zur Laufzeit zum richtigen Kontext gebunden und ist daher trotzdem noch threadsafe. Das Request-Objekt definiert viele Attribute die die Anfrage beschreiben. Z.B. `method` für die HTTP-Methode, `full_path` für die aktuelle URL, `args` für die Queryparameter, `form` für den HTTP-Body und viele mehr.

```
1  @app.route('/login', methods=['POST', 'GET'])
2  def login():
3      error = None
4      if request.method == 'POST':
5          if valid_login(request.form['username'],
6                         request.form['password']):
7              return log_the_user_in(request.form['username'])
8          else:
9              error = 'Invalid username/password'
10     # the code below is executed if the request method
11     # was GET or the credentials were invalid
12     return render_template('login.html', error=error)
```

Abbildung 21: Das Request-Objekt

6.2.2.5 Redirects und Errors

Um einen Benutzer weiterzuleiten kann man die Methode `redirect()` benutzen. Um einen Request frühzeitig zu verlassen bietet sich `abort()` an.

```
1  from flask import abort, redirect, url_for
2
3  @app.route('/')
4  def index():
5      return redirect(url_for('login'))
6
7  @app.route('/login')
8  def login():
9      abort(401)
10     this_is_never_executed()
```

Abbildung 22: Weiterleiten mittels redirect

Man könnte noch den Default-Errorhandler überschreiben, damit nicht nur eine schwarz-weisse Seite angezeigt wird:

```
1 from flask import render_template
2
3 @app.errorhandler(404)
4 def page_not_found(error):
5     return render_template('page_not_found.html'), 404
```

Abbildung 23: Errorhandling

6.2.2.6 Response

Die Rückgabewerte aus einer Funktion werden automatisch in ein Response-Objekt konvertiert. Wenn der Rückgabewert ein `String` ist, wird er in ein Response-Objekt mit einem `String` als Response-Body, 200 OK Status und `text/html` als MIME-Type umgewandelt. Die Logik mit der Flask die Response-Objekte zusammenbaut funktioniert folgendermassen:

1. Wenn ein Respons-Objekt mit dem richtigen Typ zurückgegeben wird, wird dieses ohne Änderungen direkt zurückgegeben.

2. Wenn es ein `String` ist, wird ein Respons-Objekt mit den Daten und Default-Parameter kreiert.

3. Mittels einem Tuple kann man mehr Informationen mitgeben. Es muss in der Form (response,status,header) oder (response,header) gebraucht werden und überschreiben die Default-Parameter entsprechend.

4. Wenn das nicht funktioniert, geht Flask davon aus, dass es sich beim Rückgabebjekt um eine WSGI-Applikation handelt und wandelt diese in ein Response-Objekt um.

6.2.2.7 Session

Zusätzlich zum Request-Objekt gibt es noch ein weiteres Objekt. Das Session-Objekt erlaubt es zusätzliche, benutzerspezifische Informationen von einem Request zu einem nächsten Request zu speichern. Es baut auf dem Cookie auf und signiert es zusätzlich. Das heisst man kann den Inhalt lesen, aber nicht ändern (Nur wenn man das Secret kennt).

```
1  from flask import Flask, session, redirect, url_for, escape, request
2
3  app = Flask(__name__)
4
5  # Set the secret key to some random bytes. Keep this really secret!
6  app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'
7
8  @app.route('/')
9  def index():
10     if 'username' in session:
11         return 'Logged in as %s' % escape(session['username'])
12     return 'You are not logged in'
13
14 @app.route('/login', methods=['GET', 'POST'])
15 def login():
16     if request.method == 'POST':
17         session['username'] = request.form['username']
18         return redirect(url_for('index'))
19     return '''
20         <form method="post">
21             <p><input type=text name=username>
22             <p><input type=submit value=Login>
23         </form>
24     '''
25
26 @app.route('/logout')
27 def logout():
28     # remove the username from the session if it's there
29     session.pop('username', None)
30     return redirect(url_for('index'))
```

Abbildung 24: Das Session-Objekt

6.2.2.8 Logging

Wenn man gewisse Sachen loggen möchte ist seit Version 0.3 ein vorkonfigurierter Logger verfügbar. Der Logger ist ein Standard-Logging-Logger von Python.

```
1 app.logger.debug('A value for debugging')
2 app.logger.warning('A warning occurred (%d apples)', 42)
3 app.logger.error('An error occurred')
```

Abbildung 25: Logging

6.2.2.9 Testing

Es gibt ein Modul `Flask-testing`, welches das normale Testskelett von Flask schon implementiert. Es ermöglicht das einfache Unit-Testing von Routen.

Man muss eine Flask-Instanz mit einer Methode zurückgeben:

```
1 from flask import Flask
2 from flask_testing import TestCase
3
4 class MyTest(TestCase):
5
6     def create_app(self):
7         app = Flask(__name__)
8         app.config['TESTING'] = True
9         return app
```

Abbildung 26: Testing mit Flask-Testing

Danach kann man schon anfangen Antworten zu testen:

```
1 @app.route("/ajax/")
2 def some_json():
3     return jsonify(success=True)
4
5 class TestViews(TestCase):
6     def test_some_json(self):
7         response = self.client.get("/ajax/")
8         self.assertEquals(response.json, dict(success=True))
```

Abbildung 27: Testing JSON

Um die Datenbankverbindung in den Tests mit einzubeziehen, kann man das DB-Objekt von SQLAlchemy einbinden.

```
1  from flask_testing import TestCase
2
3  from myapp import create_app, db
4
5  class MyTest(TestCase):
6      SQLALCHEMY_DATABASE_URI = "sqlite://"
7      TESTING = True
8
9      def create_app(self):
10         # pass in test configuration
11         return create_app(self)
12
13     def setUp(self):
14         db.create_all()
15
16     def tearDown(self):
17         db.session.remove()
18         db.drop_all()
```

Abbildung 28: Testing SQLAlchemy

Man muss nur aufpassen mit der DB-Session, da SQLAlchemy für Flask die Session nach jedem Request schliesst um threadsafe zu sein. Das heisst nach einem Test muss man wieder eine neue Session initialisieren.

Um die Tests laufen zu lassen, wird der Testloader gebraucht. Dieser kann auch automatisch alle Tests im Ordner entdecken und ausführen, somit müssen nicht alle Tests in einer Datei sein.

```
1 @cli.command()
2 def test():
3     """ Runs the tests without code coverage"""
4     tests = unittest.TestLoader().discover('project/tests',
5                                            pattern='test*.py')
6     result = unittest.TextTestRunner(verbosity=2).run(tests)
7     if result.wasSuccessful():
8         return 0
9     return 1
```

Abbildung 29: Test-Runner

6.2.2.10 Application Context

Der Applikationskontext beinhaltet Daten auf Applikationslevel während einem Request, einem CLI-Command oder anderen Aktivitäten. Damit dieses Objekt nicht jeder Funktion mitgegeben werden muss, kann man mittels `current_app`- oder `g`-Proxies darauf zugreifen. Dies funktioniert ähnlich zum Request-Kontext.

Das Kontextobjekt besitzt Attribute wie z.B. `config`, welche nützlich innerhalb Views oder CLI-Commands sein können. Ein Problem besteht, wenn man in verschiedenen Modulen die `app`-Instanz importiert. Dann können zirkuläre Referenzen auftreten. Um das zu verhindern kann man das App-Factory-Pattern, Blueprints oder Extensions benutzen, so dass es eigentlich keine globale `app` Instanz zum Importieren gibt. Man kann dann mittels `current_app`-Proxy auf die Instanz zugreifen.

Flask pusht automatisch den Applikationskontext während der Verarbeitung eines Requests. Alle Funktionen, welche während eines Requests laufen, haben Zugriff auf das `current_app`-Proxy. Auch CLI-Commands, die mit `Flask.cli` registriert wurden haben darauf Zugriff.

Der Applikationskontext hat typischerweise die gleiche Lebensdauer wie der Request-Kontext und wird nach einem Request wieder entfernt.

Wenn man keinen Zugang zu `current_app` hat, wenn man z.B. die Applikation konfiguriert und eine Extension hinzufügt, hat man normalerweise direkt auf `app` Zugriff und man kann mittels einem `with`-Block den Kontext brauchen.

```python
def create_app():
    app = Flask(__name__)

    with app.app_context():
        init_db()

    return app
```

Abbildung 30: With Block um current_app zu nutzen

Der Applikationskontext ist ein guter Ort um gemeinsame Daten für einen Request oder CLI-Command zu speichern. Dafür gibt es das `g`-Objekt. Es ist ein simples Namespace-Objekt, das die gleiche Lebensdauer wie der Applikationskontext besitzt. `g` steht für global, ist aber nur kontextglobal und wird nach dem Request wieder zerstört. Um Daten über mehrere Requests zu speichern kann man `session` oder eine Datenbank brauchen.

Normalerweise braucht man das `g`-Objekt um Ressourcen zu managen:

1. `get_X()` kreiert eine Ressource X, falls sie noch nicht besteht und cacht sie in `g.X`

2. `teardown_X()` schliesst oder deallociert die Ressource, falls sie existiert. Es muss mit einem `teardown_appcontext()`-Handler registriert werden um nach einem Request auch wieder abgebaut zu werden.

```python
from flask import g

def get_db():
    if 'db' not in g:
        g.db = connect_to_database()

    return g.db

@app.teardown_appcontext
def teardown_db():
    db = g.pop('db', None)

    if db is not None:
        db.close()
```

Abbildung 31: DB Verbindung managen

Während einem Request wird nun jeder Aufruf von `get_db()` die gleiche Verbindung zurückgeben und sie wird automatisch nach dem Request wieder geschlossen. Man kann auch ein `LocalProxy` erstellen um einen neuen lokalen Kontext von `get_db()` zu erstellen.

```
1 from werkzeug.local import LocalProxy
2 db = LocalProxy(get_db)
```

Abbildung 32: LocalProxy

Jetzt werden Aufrufe von db intern `get_db()` aufrufen. Dies ist die gleiche Funktionsweise wie bei `current_app` der die `app`-Instanz zurückgibt.

6.2.2.11 Blueprints Flask benutzt das Konzept von Blueprints um verschiedene Applikationskomponenten zu erstellen und gemeinsame Patterns zu kapseln. Blueprints können die Applikation sehr vereinfachen oder auch das Arbeiten mit grösseren Applikationen ermöglichen. Es ist ähnlich zu einem Flask-Objekt, aber ist eigentlich nur ein Bauplan (Blueprint) wie man das Objekt erstellt.

```
1  from flask import Blueprint, render_template, abort
2  from jinja2 import TemplateNotFound
3
4  simple_page = Blueprint('simple_page', __name__,
5                          template_folder='templates')
6
7  @simple_page.route('/', defaults={'page': 'index'})
8  @simple_page.route('/<page>')
9  def show(page):
10     try:
11         return render_template('pages/%s.html' % page)
12     except TemplateNotFound:
13         abort(404)
```

Abbildung 33: Einfacher Blueprint

Dies ist ein sehr einfacher Blueprint. Dieser rendert die Seiten welche angegeben werden oder gibt sonst ein 404 (Not found) zurück, falls das Template nicht existiert. Wenn man die Funktion mittels dem `@blueprint_name.route`-Decorator bindet, wird das Blueprint bei der Registrierung die Funktion `show` bei der Applikation unter der entsprechenden URL anmelden. Um das Blueprint zu registrieren, gibt es die Methode `register_blueprint`.

```
1 from flask import Flask
2 from yourapplication.simple_page import simple_page
3
4 app = Flask(__name__)
5 app.register_blueprint(simple_page)
```

Abbildung 34: Registrierung eines Blueprints

Wenn man die registrierten Regeln ansieht, wird man auch die Blueprint Routen finden.

```
1 [<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
2  <Rule '/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
3  <Rule '/' (HEAD, OPTIONS, GET) -> simple_page.show>]
```

Abbildung 35: Regeln mit Blueprint

Die erste Regel ist von der Applikation für statische Dateien und die beiden anderen sind vom Blueprint. Man kann die Blueprints auch auf verschiedene Orte mounten.

```
1 app.register_blueprint(simple_page, url_prefix='/pages')
```

Abbildung 36: Prefix eines Blueprints

Jetzt werden die Blueprint Routen mit `/pages` vorangestellt eingefügt.

```
1 [<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
2  <Rule '/pages/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
3  <Rule '/pages/' (HEAD, OPTIONS, GET) -> simple_page.show>]
```

Abbildung 37: Regeln mit Prefix und Blueprints

6.2.2.12 Patterns

Einige Dinge sind gleich genug damit die Chance hoch ist, dass die meisten Webapplikationen diese benutzen. Das sind z.B. Verbindungen zu relationalen Datenbanken, User-Authentifizierung usw. Für diese Funktionalitäten gibt es bestimmte Patterns und Snippets, die man verwenden kann. Nachfolgend sind einige dieser Patterns und Snippets:

- Für grössere Applikationen kann man Subpackages mittels Blueprints erstellen und erreicht damit eine gute Struktur der gesamten Applikation.
- Um die Applikation zu erstellen kann man Application-Factories benutzen.
- Um mehrere Flask-Applikationen zusammen zu verwalten kann man Application-Dispatching verwenden.
- Um gemeinsame Exceptions zu verwalten kann man eigene Exceptions erstellen.

Diese Patterns können gut in der Flask-Dokumentation nachgelesen werden und sind für Neueinsteiger sehr hilfreich. Allerdings sollte man zuerst die anderen Basics wie z.B. Kontext, Blueprints usw. verstanden haben.

7 Software Architektur

7.1 Systemübersicht

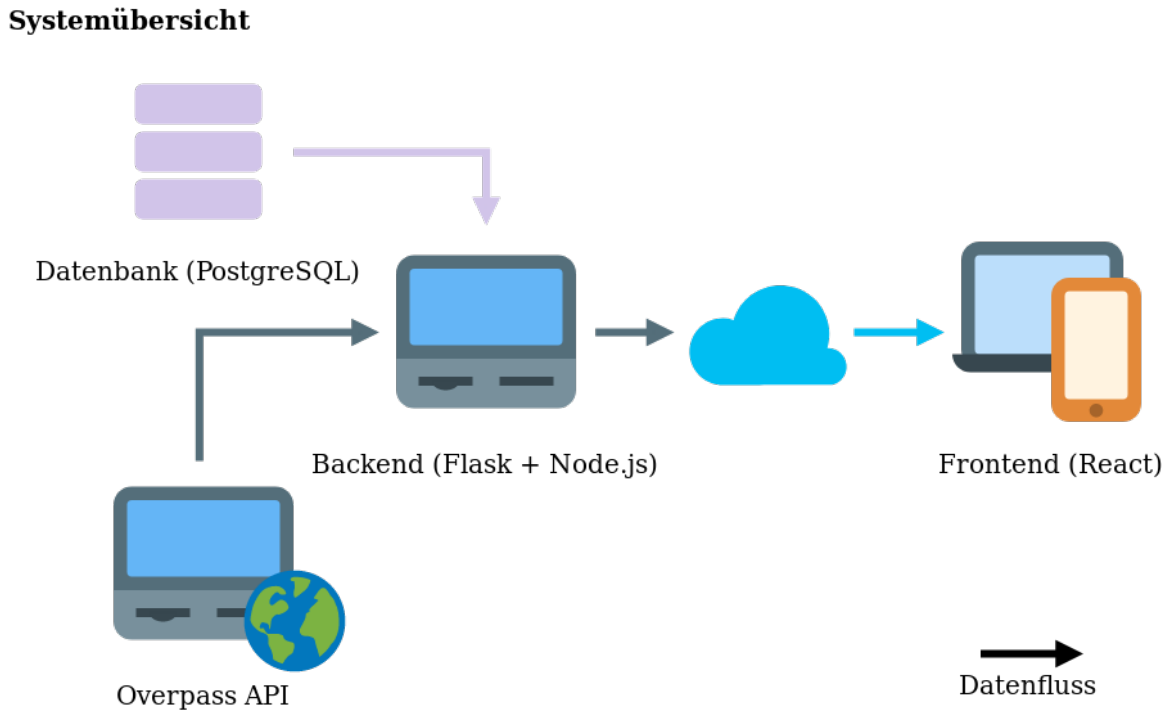

Abbildung 38: Systemübersicht

Die Systemübersicht gibt einen Überblick über die verschiedenen Komponenten der Webapplikation. Mittels Datenbank werden die Vorlagen, gesharten Karten und Benutzerangaben gespeichert. Mittels Overpass-API können die interessanten Punkte von OpenStreetMap abgefragt und im Frontend angezeigt werden. Das Backend bündelt die Funktionalität der Datenspeicherung und Verfügbarkeit, während das Frontend dem Benutzer die abgefragten Daten anzeigt und auf Benutzereingaben reagiert.

7.2 Docker-Landschaft

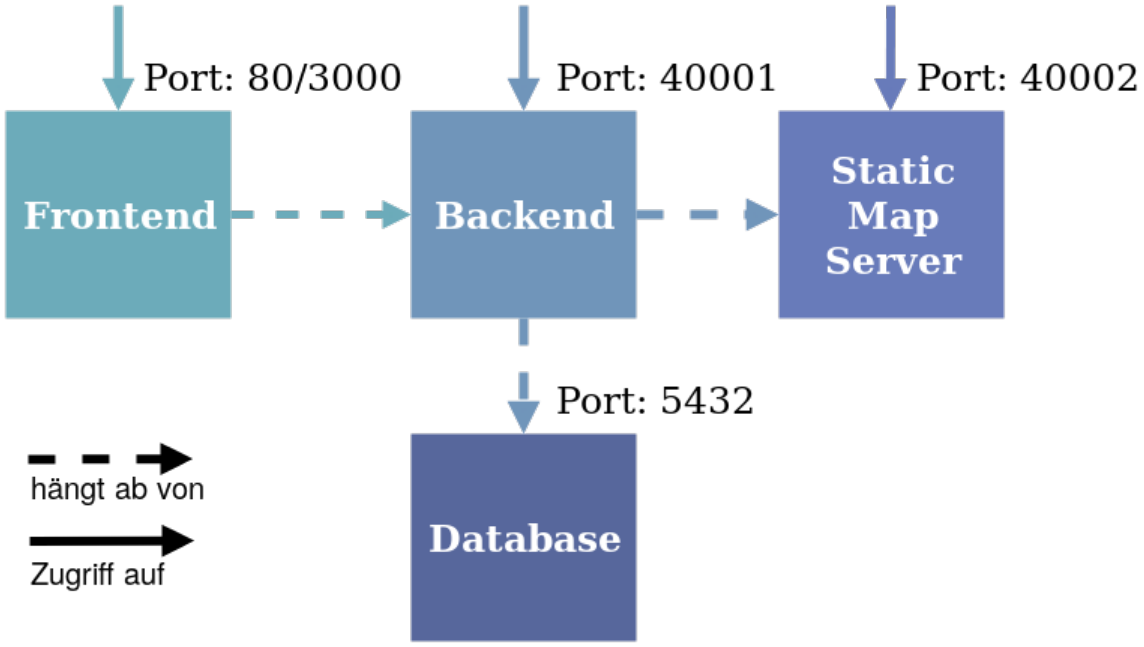

Abbildung 39: Docker-Landschaft

Die Docker-Landschaft gibt einen Überblick über die verschiedenen Docker-Container, deren Abhängigkeiten und die verfügbaren Ports. Der Frontend-Container hat in der Produktivumgebung Port 80 offen und für die Entwicklung Port 3000. In der Produktivumgebung läuft ein Nginx um die kompilierten React-Dateien auszuliefern und in der Entwicklung wird mit dem Server von create-react-app gearbeitet um das hot-Reloading nutzen zu können. Das Frontend braucht ein gestartetes Backend. Dieses hat Port 40001 gegen aussen geöffnet, da diese Ports auf dem HSR-Server schon öffentlich verfügbar sind. Im Backend-Container läuft in der Produktivumgebung ein Gunicorn-Server mit vier Workern und Zugriff auf die glsFlask-Applikation um multithreadingfähig zu sein. In der Entwicklung läuft Flask. Der StaticMap-Server liefert die hochaufgelösten Daten für den Druck. Dieser ist unter dem Port 40002 erreichbar und läuft mit Node.js und JavaScript. Die Datenbank läuft intern und ist nicht gegen aussen verfügbar auf dem Standardport 5432.

7.3 Frontend

Da es sich um eine React-Applikation handelt, wird die Architektur auch mit Komponenten beschrieben. Ein konzeptioneller Unterschied welcher hier beachtet werden muss, ist der zwischen sogenannten Container- und Presentational-Komponenten. Presentational-Komponenten sind dumm. Sie wissen nicht, woher Daten stammen, welche sie darstellen sollen. Sie definieren lediglich, welche Art von Daten (und Funktionen) sie benötigen und bekommen diese mittels Properties von anderen (oft Container-) Komponenten. Containerkomponenten definieren, welche Daten vom globalen Store und welche Actions an die Presentational-Komponenten weitergegeben werden sollen. Durch diese Trennung können Presentational-Komponenten mit verschiedenen Daten wiederverwendet werden.

Digital Tourist Map ist vom Umfang her eher übersichtlich, weshalb die Architektur und Struktur auch simpel belassen werden konnten.

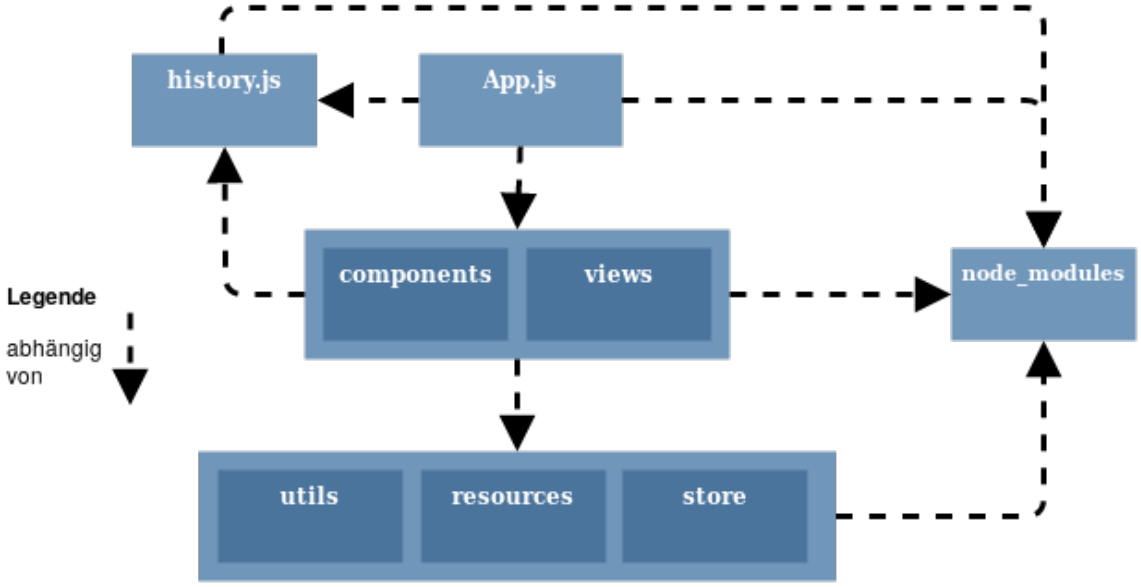

Abbildung 40: Frontend Architektur

JavaScript kennt keine Packages oder Namespaces, deshalb werden im Diagramm einzelne Files und Directories aufgezeigt. Presentational- und Containerkomponenten werden für bessere Übersicht in der Filestruktur am selben Ort abgelegt (in `components` und `views`), auch wenn nur Containerkomponenten direkt von `store` abhängig sind.

7.4 Backend

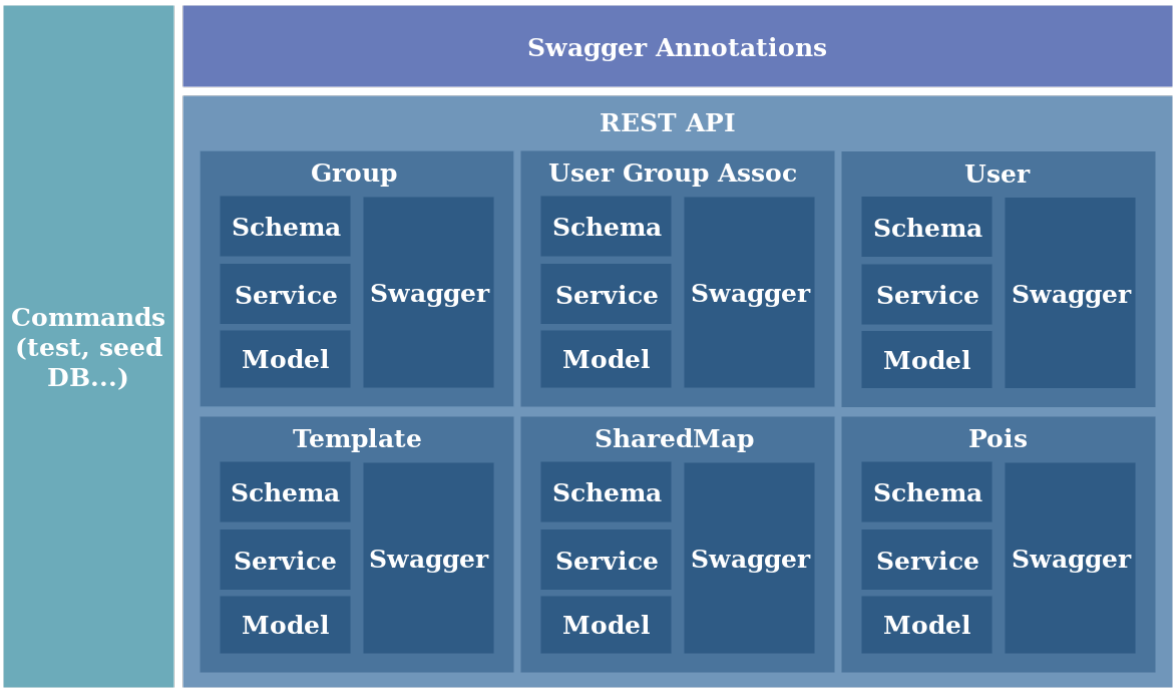

Abbildung 41: Backend Übersicht

Die Backendübersicht zeigt die Aufteilung der Backend-Applikation. Die RESTful-API besteht aus einzelnen, in sich fast gänzlich geschlossenen Subpackages. Diese wiederum stellen eine Subroute auf der API dar (z.B. User: `/user`). Sie besitzen eigene Swagger-Annotationen für das SwaggerUI, ein Schema für die Validierung und Konvertierung zum Model, einen Service für die Operationen mit dem Model und ein Datenbank-Model, welches auf der Datenbank abgebildet wird. Diese Subpackages implementieren eigentlich einen Blueprint, welcher dann im Hauptprogramm geladen wird. Dies ermöglicht eine fast gänzlich eigenständige Entwicklung der Subpackages und eine gute Wiederverwendbarkeit. Um das Testen angenehmer zu machen wurden noch verschiedene CLI-Commands implementiert, die z.B. eine Datenbank mit Demodaten befüllen.

7.5 Datenbank

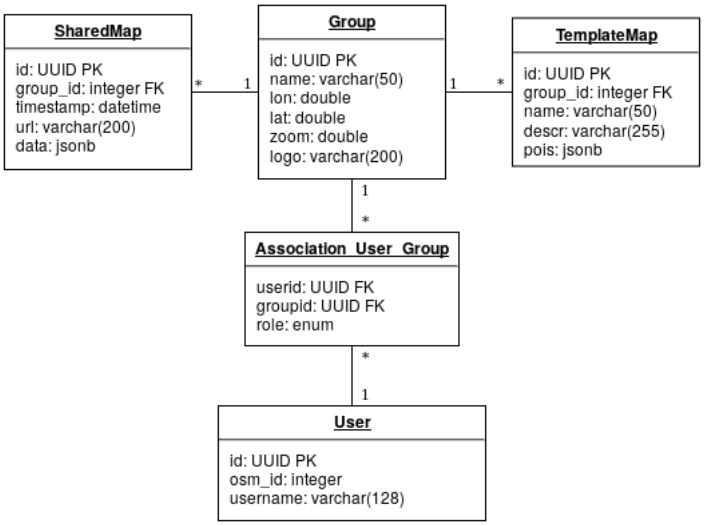

Abbildung 42: Domainmodell

Das Domainmodell zeigt die Aufteilung in der Datenbank. Die n-n-Beziehung zwischen Benutzer und Gruppe wurde mit Hilfe einer Zwischentabelle getrennt. Darauf befindet sich die entsprechende Rolle des Benutzers in der Gruppe. Eine Gruppe kann beliebig viele Vorlagen und gesharte Karten besitzen. Diese speichern die Daten als JSON direkt in die Datenbank. Alle IDs werden als UUID gespeichert um keine Rückschlüsse auf die nachfolgenden Objekte zu erhalten. Die redundanten Daten (gleiche POIs in verschiedenen Gruppen und Vorlagen) wurde bewusst so gehalten, damit ein Update einer Gruppe keinen Einfluss auf die POIs einer anderen Gruppe hat.

7.6 RESTful API

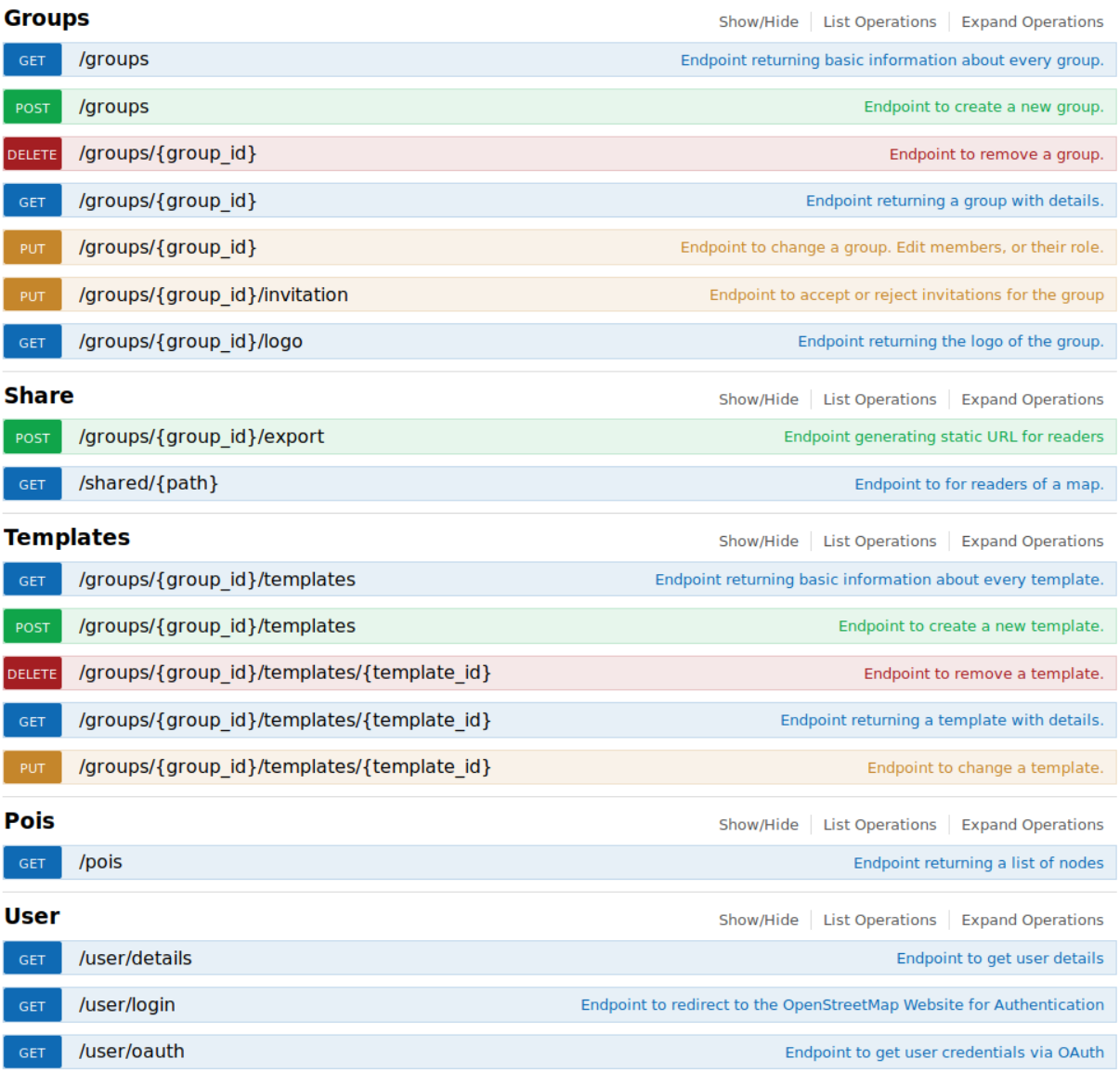

Abbildung 43: API Swagger UI

Das SwaggerUI zeigt die Aufteilung der verschiedenen Subrouten und deren Methoden. Die einzelnen Routen sind in den jeweiligen Implementationskapiteln beschrieben. Mit Hilfe des SwaggerUIs kann man die verschiedenen Methoden ausprobieren und die Datenmodelle anschauen, um einen möglichst einfachen Einstieg in die API zu erhalten.

8 Implementation

8.1 Frontend

8.1.1 React-Map-Gl

8.1.1.1 Problem

In der Applikation müssen an verschiedenen Stellen Karten eingebettet werden. Die Karten müssen ein relativ eingeschränktes Feature-Set unterstützen. Es muss möglich sein den Kartenausschnitt und die Zoomstufe zu verändern. Es müssen Punkte mit Zusatzinformationen auf der Karte dargestellt werden können. Idealerweise kann die Karte mit Daten im GeoJSON-Format umgehen und Vektorkacheln darstellen.

8.1.1.2 Lösung

Da die Karten Teil der Kernfunktionalität von Digital Tourist Map sind, wurde anfänglich eine Evaluation mit Nutzwertanalyse (Seite 112) durchgeführt. Die Entscheidung fiel auf `React-Map-Gl`. Andere Kandidaten waren das verbreitete `Leaflet`, `React-Mapbox-Gl` sowie `React-Geo`.

8.1.1.3 Umsetzung

Um den Einsatz der Komponente zu vereinfachen, wurde dafür eine Wrapper-Komponente `Map` erstellt. Diese nimmt den Viewport (Koordinaten und Zoomstufe), Dimensionen, Event-Callbacks, POIs und eingebetteten Inhalt als Properties entgegen und stellt die Karte sowie Icons und Polygone dar (mit Erweiterung `Deck-Gl`).

```
<ReactMapGL
  {...this.props.viewport}
  width={this.props.width}
  height={this.props.height}
  onClick={e => this.props.onMapClick && this.props.onMapClick(e)}
  mapStyle={STYLE}
  onViewportChange={viewport => this.props.updateViewport && this.props.updateViewport(viewport)}
>
  <DeckGL
    width={this.props.width}
    height={this.props.height}
    {...this.props.viewport}
    layers={[geoJsonLayer, iconLayer]}
  />
  { this.props.content }
</ReactMapGL>
```

Abbildung 44: Map.js

Der Map-Style ist konstant und definiert das Aussehen der Karte und die Adresse, von welcher die Vektorkacheln geladen werden. Das Laden der Kacheln übernimmt React-Map-Gl automatisch, in diesen Prozess wird nicht eingegriffen.

8.1.1.4 Ergebnis/Ausblick

React-Map-Gl lässt nicht viel zu wünschen übrig. Um zusätzlich zu den Icons auch noch z.B. ein Textlabel anzeigen zu können wäre ein Text-Layer ideal. Zum Zeitpunkt der Implementation gab es bereits ein experimentelles Text-Layer, dieses unterstützte aber nur eine einzelne Schriftart und keine Umlaute, deshalb ist es für unsere Zwecke ungeeignet.

8.1.2 Flow

8.1.2.1 Problem

JavaScript hat dynamische Typen, weshalb nicht geprüft oder definiert werden kann welche Typen Daten haben müssen, wenn diese an Funktionen oder Komponenten weitergegeben werden. Dies kann besonders bei grösseren und komplexeren Projekten zu Problemen führen.

8.1.2.2 Lösung

In den letzten Jahren haben sich vor allem zwei Ansätze etabliert, um dieses Problem zu lösen. TypeScript und Flow sind beides JavaScript-Supersets, welche Typen unterstützen und zur Compilezeit prüfen können.
TypeScript bietet komplett neue Sprachkonstrukte und -erweiterungen und benötigt Typendefinitionen von externen Packages. Dies kann sehr umständlich sein und ist ein eher starker Bruch von traditionellem JavaScript.
Flow beschränkt sich darauf, die Sprache nur um optionale Typen zu erweitern und diese zu prüfen. Flow ist Opt-In, heisst Files mit Flow-Typen müssen markiert werden, damit der Typechecker diese überprüft. Flow bietet genau die Features, welche benötigt werden um das Problem zu lösen, nicht mehr und nicht weniger und wurde deshalb in Digital Tourist Map eingesetzt.

8.1.2.3 Umsetzung

Der wichtigste Ort, an welchem Flow zum Einsatz kommt, sind Properties von Komponenten. Da mittels Properties Daten an Komponenten mitgegeben werden, soll dort geprüft werden, ob die Daten auch die richtigen Typen haben.

Nebst primitiven Typen wie Nummern oder Strings müssen für komplexere Strukturen Typendefinitionen angelegt werden.

```
export type POI = {
    type: string,
    id: number,
    geometry: Geometry,
    properties: {
        description: string,
    }
}

export type Viewport = {
    longitude: number,
    latitude: number,
    zoom: number,
}
```

Abbildung 45: Typendefinitionen

Diese Typen können dann genutzt werden, um beispielsweise die Typen von Properties zu beschreiben.

```
type Props = {
    enabledPOIs: Array<POI>,
    inactivePOIs: Array<POI>,
    viewport: Viewport,
    width: number,
    height: number,
    updateViewport: Viewport => any,
};

```

Abbildung 46: Typen für Properties

8.1.2.4 Ergebnis/Ausblick

Besonders die Typen von Funktionssignaturen beinhalten oft noch den `any`-Typ. Um Abhilfe zu schaffen, müssten Typen für die komplexeren API-Resultate angelegt werden. Durch die Typen wäre dann auch explizit dokumentiert, welche Resultate genau vom Frontend erwartet werden.

8.1.3 Internationalisierung (i18n)

8.1.3.1 Problem

Das Frontend wurde zunächst nur auf Englisch umgesetzt, da die primäre Zielgruppe von Digital Tourist Map Mitglieder von Tourismusbüros und Hotels sind und von diesen Englischkenntnisse erwartet werden können. Um die Applikation für weitere Personengruppen einfach zugänglich zu machen, sollten mehrere Sprachen unterstützt werden und die Sprache nach Bedürfnis änderbar sein.

8.1.3.2 Lösung

Für die Internationalisierung wurde das Package React-i18next eingesetzt. i18next ist eine relativ weit verbreitete Lösung und die Implementation für React sehr beliebt. Die meisten i18n-Packages bieten etwa denselben Funktionsumfang. Kernfunktion ist das Ersetzen von fixen Strings im Code. Benötigt werden aber auch erweiternde Features wie alternative Texte für Plurale oder Texte mit eingebetteter Formatierung (wie Links oder fett und kursiv markierte Textteile). React-i18next bietet auch Funktionalität um die gewünschte Sprache im Browser festzustellen und zu persistieren.

8.1.3.3 Umsetzung

Zunächst benötigt i18next ein Konfigurationsobjekt, in welchem die Übersetzungen mitgegeben und z.B. eine Fallbacksprache definiert wird.

```
1  i18n.use(LanguageDetector)
2      .init({
3        resources: translation,
4        fallbackLng: 'en',
5        debug: true,
6        interpolation: {
7          escapeValue: false,
8        },
9        react: {
10         wait: false,
11         bindI18n: 'languageChanged loaded',
12         bindStore: 'added removed',
13         nsMode: 'default'
14       },
15 });
```

Abbildung 47: i18n Konfiguration

Die in der Konfiguration benötigten `resources` sind verschachtelte Übersetzungsobjekte welche die Textteile in verschiedenen Sprachen beinhalten.

```
const translation = {
  de: {
    translations: {
      edit: {
        tools: {
          move: 'POIs de-/aktivieren',
          place_poi: 'Platzieren Sie neue POIs auf der Karte',
          edit: 'Editeren Sie den Namen und die Beschreibung von POIs',
          print: 'Karte drucken'
        }
      },
      ...
    },
  },
  en: {
    translations: {
      edit: {
        tools: {
          move: 'Enable/disable POIs',
          place_poi: 'Place new POIs by clicking on the map',
          edit: 'Edit the name and description of POIs',
          print: 'Print map',
        }
      },
      ...
    },
  },
};
```

Abbildung 48: Beispiel Übersetzung

Um Komponenten übersetzen zu können, muss weit oben in der Komponentenhierarchie ein Provider erstellt werden, welcher sich die aktuelle Sprache merkt und die Übersetzungen an die Komponenten weitergibt. Der Provider nimmt das vorherig erstellte Konfigurationsobjekt als Property entgegen.

```
<I18nextProvider i18n={ i18n }>
  ...
</I18nextProvider>
```

Abbildung 49: i18n Provider

Komponenten, welche übersetzt werden sollen, müssen irgendwo innerhalb des Providers eingebunden werden und benötigen zusätzlich eine Higher-Order-Component (HOC). Um eine Komponente mit der HOC zu wrappen wird eine Funktion `translate` zur Verfügung gestellt.

```
1 class WelcomeView extends Component {
2     ...
3 }
4
5 export default translate('translations')(WelcomeView);
```

Abbildung 50: i18n HOC

Die HOC sorgt dafür, dass die Übersetzungsfunktion `t` als Property an die Komponente weitergegeben wird. Der Übersetzungsfunktion muss nur der Platzhalter als Argument mitgegeben werden und liefert als Resultat die Übersetzung in der aktuellen Sprache.

```
1 t('edit.tools.move')
```

Abbildung 51: Übersetzungsfunktion

Eine angenehmere Alternative zur Übersetzungsfunktion ist die `Trans`-Komponente. Auch dieser Komponente muss nur der Platzhalter mitgegeben werden, sie hat jedoch den zusätzlichen Vorteil, dass die ursprüngliche Übersetzung als Content der Komponente belassen werden kann und nicht erneut im Übersetzungsobjekt erfasst werden müssen. Aus diesem Grund beinhaltet das Übersetzungsobjekt mehr deutsche Übersetzungen als englische, da die englischen bereits im Code vorhanden sind.

```
1 <Trans i18nKey="welcome.welcome">Welcome!</Trans>
```

Abbildung 52: Transkomponente

Abschliessend wurde in der Menubarkomponente ein Sprachwechsler eingefügt. Der Sprachwechsler zeigt die aktuelle Sprache an und teilt dem Provider mit, wenn die Sprache ändern soll.

```
<Dropdown
  value={i18n.language}
  closeOnChange
  compact
  onChange={(e, { value }) => this.props.setLang(value)}
  options={[
    { key: 'En', text: 'English', value: 'en' },
    { key: 'De', text: 'Deutsch', value: 'de' },
  ]}
/>

```

Abbildung 53: Sprachwechsler

8.1.3.4 Ergebnis/Ausblick

Das Ziel der Internationalisierung war nicht nur die deutsche Übersetzung, sondern vor allem die Struktur und Möglichkeit zur Erweiterung. Um weitere Sprachen, wie etwa Französisch, zur Applikation hinzuzufügen, muss nur eine Übersetzung gemacht und ein Eintrag im Sprachwechsler erstellt werden.

8.1.4 Drucken

8.1.4.1 Problem

Nebst dem digitalen Teilen von Karten via Link oder QR-Code an andere Geräte ist das Drucken auf Papier immer noch wichtig für Touristen. Oft haben diese keine Internetverbindung oder Möglichkeit, die Akkus ihrer Geräte zu laden. Eine physische Karte bietet deshalb immer noch Vorteile. Das grösste technische Problem stellt hier die Kartenauflösung dar. Ein Screenshot des aktuellen Kartenausschnitts wäre schnell erstellt, bietet aber bei durchschnittlichen Monitoren keine genug hohe Auflösung für einen scharfen und sauberen Druck.

8.1.4.2 Lösung

Für die Lösung des Druckproblems mussten mehrere Tage für Experimente investiert werden. Zunächst wurde abgeklärt, ob das Problem komplett Clientseitig mit JavaScript gelöst werden kann, unter anderem um Serverressourcen zu sparen. Theoretisch wäre dies möglich gewesen, hätte aber einen tieferen Eingriff in die MapGl-Packages zur Folge gehabt.
Die resultierende Lösung setzt auf ein serverseitig generiertes Kartenbild (Static-Map-Generator) worauf dann clientseitig POIs mit Beschreibung platziert werden.

8.1.4.3 Umsetzung

Der Druckvorgang kann im Frontend kann über den Print-Button neben den Tools gestartet werden. Zunächst öffnet sich ein Modaldialog, in welchem eine Karte im A4-Format abgebildet ist und der Benutzer den Bereich und die Zoomstufe fixieren kann.

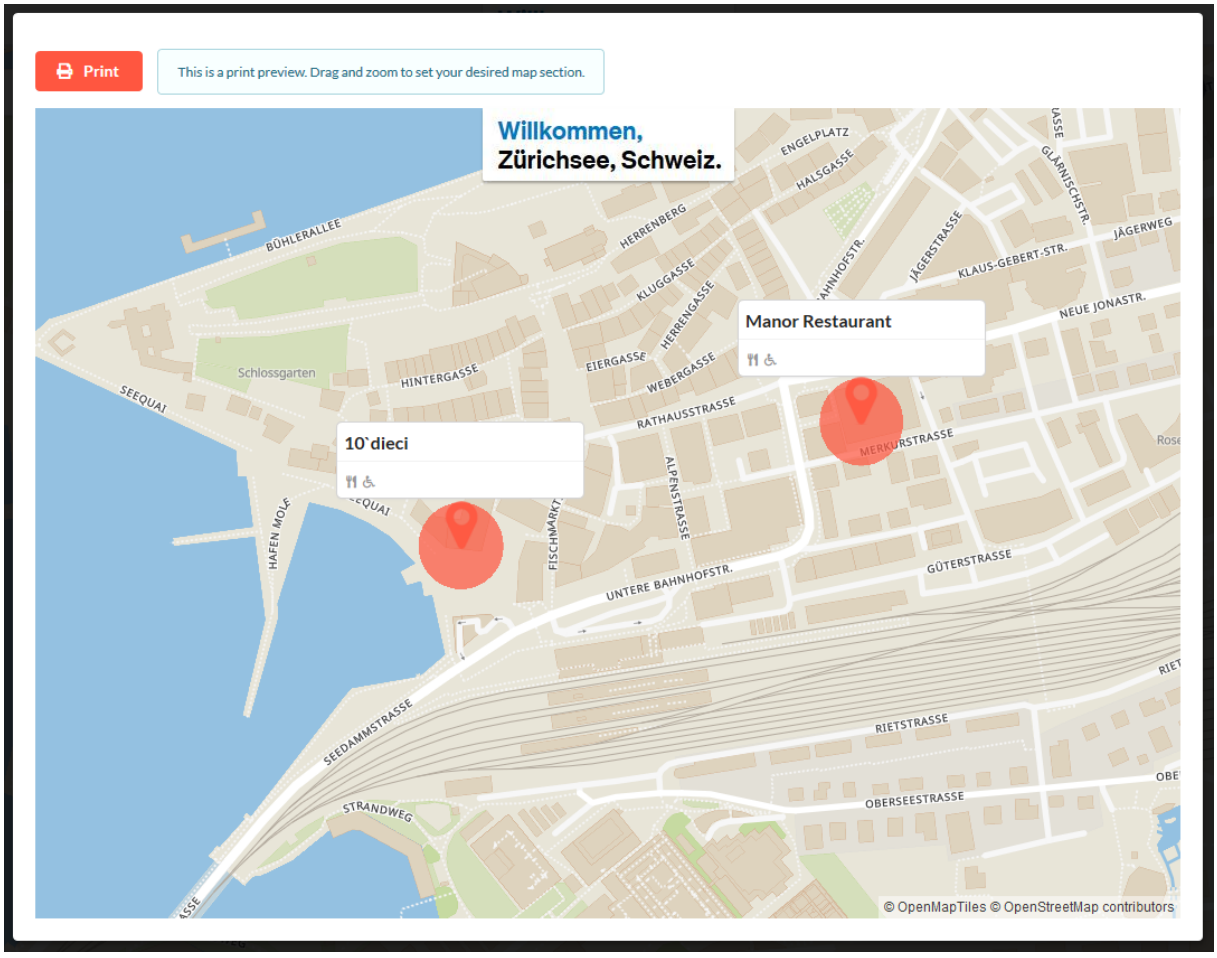

Abbildung 54: Druckvoransicht

Sobald in diesem Dialog der Druck bestätigt wird, öffnet sich ein neues Fenster. In diesem Fenster wird das hochaufgelöste Kartenbild vom Server geladen und die POIs mit Beschreibung relativ zum Bild positioniert. Dies erforderte das Umrechnen von Web-Mercator-Koordinaten in prozentuale Angaben vom Kartenbild in A4-Format.

Das resultierende Bild kann mit etwa 260 DPI gedruckt werden.

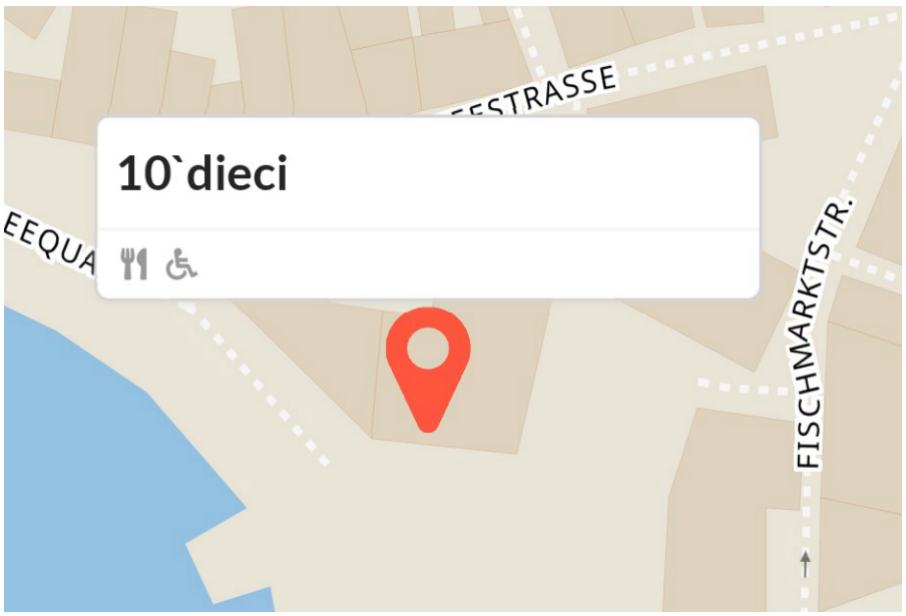

Abbildung 55: Hochaufgelöstes Druckresultat

8.1.4.4 Ergebnis/Ausblick

Die umgesetzte Lösung unterstützt leider nicht den vollen Funktionsumfang der Karte wie sie im Browser ersichtlich ist. Polygone und Linien können nicht dargestellt werden, was bei möglichen Erweiterungen der Applikation problematisch sein könnte. Ein gewünschtes Feature ist das Einzeichnen von Pfaden oder Bereichen auf der Karte. Sobald diese Tools implementiert werden, müsste auch der Druck erweitert werden.

8.2 Backend

8.2.1 API Design

8.2.1.1 Groups

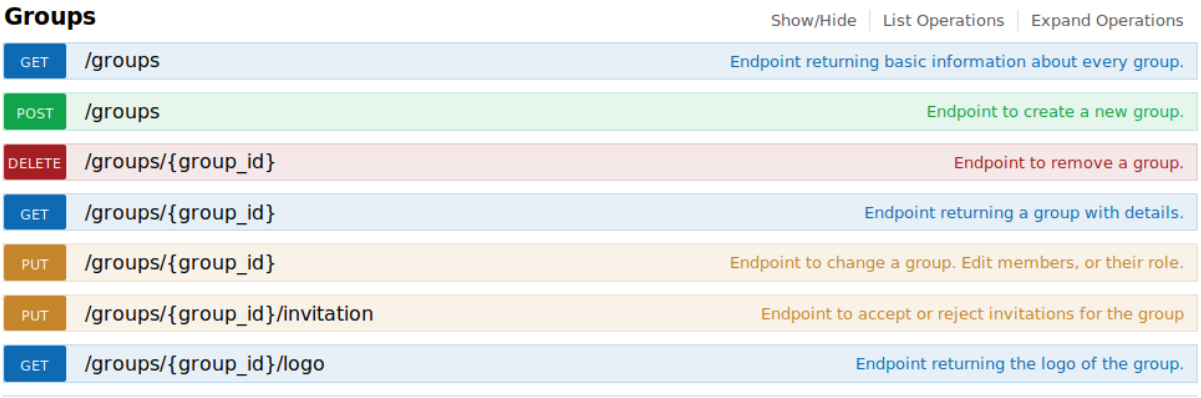

Abbildung 56: Groups API

Die Gruppenrouten sind für das Management von Gruppen. Das Erstellen, Ändern, Löschen und Abfragen der Gruppen kann mit den entsprechenden Methoden `POST`, `PUT`, `DELETE` und `GET` unternommen werden.

```
{
  "groups": [
    {
      "current_role": "owner",
      "id": "3de1a85c-6adc-4d35-807f-d47684a61225",
      "name": "TestGroup"
    },
    {
      "current_role": "pending",
      "id": "152505bc-b145-462e-aaad-4df673a9c6d4",
      "name": "TestGroup"
    }
  ],
  "invites": [
    {
      "id": "152505bc-b145-462e-aaad-4df673a9c6d4",
      "name": "TestGroup"
    }
  ]
}
```


Abbildung 57: Gruppenliste JSON

Bei einem `GET` auf alle Gruppen erhält man eine Liste von Gruppen mit einer Grobübersicht, sowie eine Liste von Einladungen des aktuellen Benutzers. Dies dient vor allem dazu, das Frontend zu entlasten, so dass es nicht zwei eigene Anfragen machen muss. Auch die Übersicht reicht für das Anzeigen im Frontend.

```
{
  "current_role": "owner",
  "id": "3de1a85c-6adc-4d35-807f-d47684a61225",
  "image": null,
  "lat": 1.2345,
  "lon": 1.2345,
  "name": "TestGroup",
  "roles": [
    {
      "role": "owner",
      "username": "ExampleUser"
    },
    {
      "role": "user",
      "username": "ExampleUser2"
    }
  ],
  "zoom": 1.2345
}
```


Abbildung 58: Gruppe JSON

Wenn man Zugriff auf die entsprechende Gruppe hat, kann man sie ändern oder löschen. Dabei wird das komplette Gruppenobjekt verwendet und geändert. Das gleiche Objekt wird bei einem `POST` erstellt und bei Erfolg zurückgegeben. Beim Ändern kann man ein Bild hinzufügen und deshalb erwartet diese Route `multipart/form-data`. Das heisst man kann das Gruppenobjekt zusätzlich zum neuen Bild hochladen.


```
{
  "username": "ExampleUser",
  "accept":true
}
```


Abbildung 59: Einladung annehmen/ablehnen JSON

Um eine Einladung anzunehmen oder abzulehnen ist die Route `/invitation` gedacht. Es wird ein einfaches Objekt mit einer Zusage oder Absage erwartet. Um einfach nur das Bild in der gespeicherten Gruppe zu erhalten, wurde die Route `/logo` erstellt. Da dies ein Bild ist, ist es nicht so einfach dieses eingebettet in JSON zurückzugeben.

8.2.1.2 Share

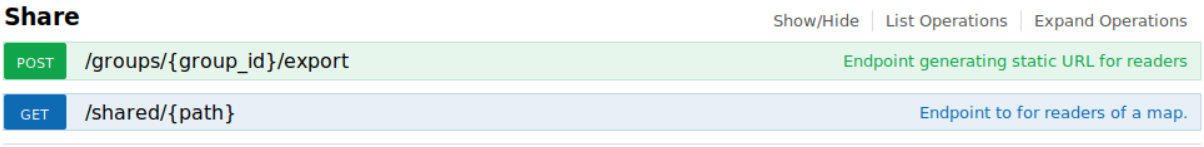

Abbildung 60: Share API

Die Sharing-Routen sind für die gesharten Karten. Um eine neue Karte zu erstellen nutzt man die `POST`-Route und um die Daten dieser Karte abzufragen nutzt man die `GET`-Route. Die POIs kann man eigentlich in jedem Format abspeichern, da sie direkt als JSON in die Datenbank gespeichert werden, aber wir haben uns darauf geeinigt, dass wir GeoJSON verwenden. Damit wird das Auslesen und Anzeigen einfacher.

```
{
  "features": [{
    "geometry": {
      "coordinates": [
        9.4756194,
        47.2853322
      ],
      "type": "Point"
    },
    "id": 272045858,
    "properties": {
      "hiking": "yes",
      "information": "guidepost",
      "tourism": "information"
    },
      "type": "Feature"
  }]
}
```

Abbildung 61: Gesharte Karte JSON

8.2.1.3 Templates

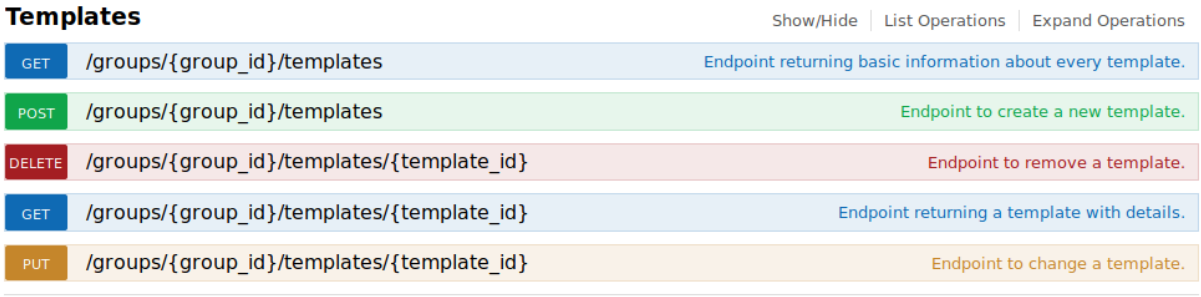

Abbildung 62: Templates API

Die Templates-Routen sind für die Kartenvorlagen. Auch hier wurden die Routen RESTful gestaltet. Alle Methoden sind für die Erstellung, Veränderung, Löschen oder Ansicht dieser Vorlagen. Dementsprechend ist das Vorlagenobjekt ziemlich selbsterklärend.

```
1 {
2   "templates": [
3     {
4       "description": "TestTemplate description",
5       "id": "19ba7f90-cf30-4595-b9d5-94ec991ba350",
6       "name": "TestTemplate"
7     }
8   ]
9 }
```

Abbildung 63: Vorlagenliste JSON

Auch bei dieser Detailansicht erhält man, ähnlich zu den Gruppen, noch zusätzliche Informationen wie z.B. die POIs im abgespeicherten Format.

```
{
  "description": "TestTemplate description",
  "group": "3de1a85c-6adc-4d35-807f-d47684a61225",
  "id": "19ba7f90-cf30-4595-b9d5-94ec991ba350",
  "name": "TestTemplate",
  "features": [
    {
      "type": "Feature",
      "id": 26860577,
      "geometry": {
        "type": "Point",
        "coordinates": [
          9.1942802,
          47.3066078
        ]
      },
      "properties": {
        "descr": "This is a test description",
        "tags": {
          "name": "Bächli",
          "tourism": "camp_site"
        }
      }
    }
  ]
}
```


Abbildung 64: Vorlage JSON

8.2.1.4 POIs

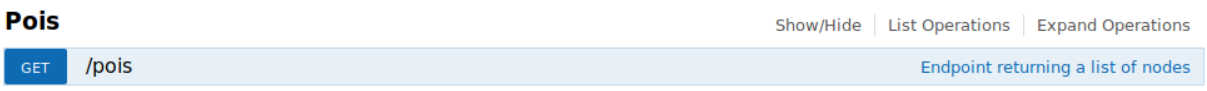

Abbildung 65: POIs API

Die POIs-Route dient für die Abfrage und das Filtern der interessanten Punkte auf der Karte. Die Rückgabe wird im GeoJSON-Format zurückgegeben um eine einfache Wiederverwendbarkeit zu gewährleisten. Dabei werden die OpenStreetMap-Datentypen in GeoJSON-Datentypen umgewandelt. Dabei werden Nodes, Ways und einfache Relations unterstützt (8.2.3.2).

8.2.1.5 User

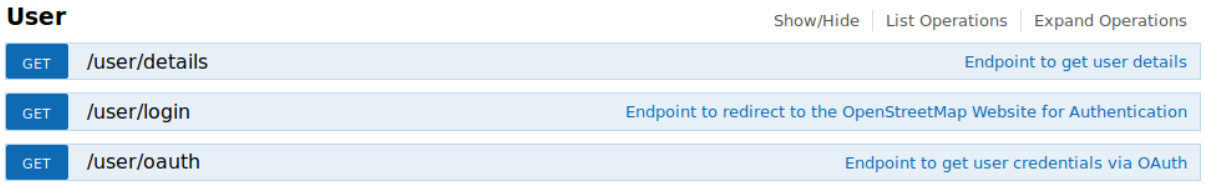

Abbildung 66: User API

Die User-Routen sind für das Login und Abfragen der OpenStreetMap-Benutzer. Dabei werden nur die wirklich benötigten Daten abgespeichert und zurückgegeben.

```
1 {
2   "osm_id": 12345,
3   "username": "ExampleUser"
4 }
```

Abbildung 67: Benutzer JSON

8.2.2 Aufteilung

Die Aufteilung der ganzen Backend-Applikation wurde mit sogenannten Submodules gemacht. Normalerweise könnte man kleinere Python-Programme in einer einzigen Datei oder einem kleinen Ordner schreiben. Wenn dies jedoch nicht ausreicht kann man noch mehrere Unterordner einbeziehen, was wir in diesem Projekt gemacht haben. Jeder Unterordner beschreibt eine Subroute in der API, also User für `/user` und Group für `/group`. Die Abhängigkeiten zwischen einander wurden bestmöglich aufgehoben und deshalb sollten diese Ordner auch für andere Flask-Applikationen wiederverwendbar sein. Zusätzlich wurde auf eine gute Struktur innerhalb dieser Submodules geachtet. Man hat ein Datenbankmodell, welches durch den OR-Mapper abgebildet wird. Ein zusätzliches Schema sorgt für die Validierung der einkommenden Daten und für die Umwandlung von Datenmodell zu Viewmodell. Ein Service abstrahiert die Datenbankzugriffe, sodass der Controller keine direkte Abhängigkeit auf die Datenbank hat und auch dieser wiederverwendbar ist. Und ein Unterordner mit allen Swagger-Konfigurationen wird für die Beschreibung der Route verwendet. Damit sind die Submodules möglichst self-contained und können beliebig wiederverwendet oder ausgetauscht/weggelassen werden.

8.2.3 Technologien

8.2.3.1 Flasgger

`Flasgger` ist eine Flask-Extension um die OpenAPI-Spezifikation aus verschiedenen API-Routen zu extrahieren. Zudem kommt `Flasgger` mit eingebauten SwaggerUI, so dass man sehr einfach die verschiedenen Routen ausprobieren und einsehen kann. Man kann die Spezifikation mittels Docstrings, externe YAML-Dateien, Dictionaries oder Marshmallow-Schemata definieren und `Flasgger` wird daraus zur Laufzeit eine OpenAPI-Spezifikation erstellen und auf `/apidocs` via SwaggerUI darstellen.

Problem Das Problem ist es eine REST-API zu dokumentieren. Swagger (neu OpenAPI) greift genau dieses Thema auf und versucht es mittels YAML-Definitionen zu lösen und zu vereinfachen. Das automatische Extrahieren dieser Spezifikation ist sehr willkommen und man kann die Dokumentation sehr schnell anpassen. Allerdings unterstützt Swagger und daher auch `Flasgger` kein Hypermedia, welches nach Leonard Richardson das letzte Level von REST darstellen würde.[8]
Man könnte auch die ganze API von Hand in einer Textdatei (HTML, Markdown o.Ä.) spezifieren. Dies bringt jedoch eher Probleme mit der Wiederverwendbarkeit und Instandhaltung dieser Dokumente.

Lösung OpenAPI ist de facto Standard um eine REST-API so gut wie möglich zu beschreiben. Es gibt viele Tools, die das Arbeiten für Entwickler vereinfachen und z.B. wie `Flasgger` die Spezifikation automatisch extrahieren.
Ein anderes mögliches Modul für Swagger mit Flask wäre auch `flask-swagger` gewesen. Dieses unterstützt das Extrahieren der Spezifikation aus den Docstrings oder aus YAML-Dateien. Der Export basiert allerdings nur auf einer statischen Datei und bindet die Benutzeroberfläche nicht mit ein. Deshalb haben wir uns für `Flasgger` entschieden und die Implementierung gestaltete sich als unkompliziert.

Umsetzung Zuerst muss die Flask-Applikation von `Flasgger` gewrappt werden.

```
1 from flasgger import Swagger
2 Swagger(app, template_file='api/swaggerdoc/definitions.yml')
```

Abbildung 68: Flasgger Initialisierung

Dies extrahiert die allgemeinen Informationen aus einer YAML-Datei. Dort sind die Sicherheitseinstellungen sowie die Error-Typen definiert, da diese im ganzen Backend gebraucht werden.

```python
@swag_from('swagger/groups_get.yml')
def groups_get() -> Tuple[any, int]:
    """
    Get a list of groups of the user through JWT
    :return: The json data
    :rtype flask.Response
    """
    pass
```

Abbildung 69: `Flasgger` Decorator

Mittels Decorators kann eine YAML-Datei pro Route angegeben werden. Dies ermöglicht eine gute Trennung der einzelnen Funktionen. Die einzelnen Dateien werden zusammengesetzt und mittels Benutzeroberfläche kann darauf zugegriffen werden.

```yaml
Endpoint returning basic information about every group.
This is using docstrings for specifications.
---
tags:
  - Groups

security:
  - Bearer:
    import: "project/api/swaggerdoc/security.yml"

responses:
  200:
    description: List with group overviews
    schema:
      import: "project/api/group/swagger/definitions/group_list.yml"
```

Abbildung 70: `Flasgger` YAML-Datei

Die Implementierung der Importfunktion der YAML-Datei konnte erst nach längerer Zeit implementiert werden, da durch mangelnde Dokumentation die genaue Funktion in `Flasgger` unbekannt war. Schlussendlich haben wir es aber doch noch geschafft andere YAML-Dateien in eine einzelne Datei einzubinden. Man muss lediglich den Root-Pfad des Projektes angeben, denn der `Flasgger`-Parser wird von dort aus gestartet und kommt sonst mit den relativen Pfaden innerhalb der Submodules nicht klar.

Ergebnis/Ausblick Schlussendlich konnte mittels `Flasgger` eine unterteilte OpenAPI-Spezifikation erstellt werden, welche durch ein übersichtliches Benutzerinterface einfach zugreifbar und ausprobierbar ist. Alle Modelle können nachvollzogen werden und dienen auch für spätere Anbindungen von anderen Systemen. Hauptsächlich aber wurde es für eine gemeinsame Basis des Backends und Frontends verwendet. OpenAPI wird laufend angepasst und viele neue Funktionen werden hinzugefügt. Die Dokumentation von REST-APIs wird immer einfacher und eventuell müssen die Module zu einem späteren Zeitpunkt erneut evaluiert werden.

8.2.3.2 POIs, oder OpenStreetMap und das schwierige Verhältnis zu GeoJSON

OpenStreetMap setzt auf ein relationales Datenmodell mit den drei Hauptelementen Nodes, Ways und Relations. Nodes repräsentieren dabei einen Punkt auf der Erdoberfläche und werden mit Längen- und Breitengrad angegeben. Ways sind geordnete Listen mit 2 bis 2000 Nodes, welche eine Polyline definieren. Und Relations sind Mehrzweckelemente, die Beziehungen zwischen zwei oder mehreren Elementen (Nodes, Ways, Relations) definieren.

GeoJSON ist seit Juni 2008 ein RFC Standard und ist damit 4 Jahre jünger als OpenStreetMap. Es ist ein offenes Format um geographische Daten in einem JSON-Objekt zu repräsentieren. Dabei kennt GeoJSON drei einfache und drei mehrteilige Geometrien. Zu den einfachen Geometrien gehören Point, LineString und Polygon. Zu den mehrteiligen Geometrien gehören MultiPoint, MultiLineString und MultiPolygon. Points entsprechen einem Punkt (Position) in einem zweidimensionalen Raum. LineStrings und MultiLineStrings beinhalten eine Liste mit mehreren Positionen. Polygone beinhalten einen (ohne Löcher) oder mehrere (mit Löcher) lineare Ringe (LineString) um eine geschlossene Figur darzustellen. Multipolygone enthalten eine Vielzahl von solchen geschlossenen Figuren.[9]

Problem Da wir das Mapbox-Format verwenden und dieses GeoJSON benötigt um Punkte darstellen zu können, müssen wir die Antwort von OpenStreetMap zu GeoJSON konvertieren. Die Standards sind dabei nicht gänzlich kompatibel. So dass z.B. ein Node einem Point entspricht, aber ein Way entweder einem LineString (Fluss) oder einem Polygon (Rasenfläche auf Feld). Kompliziert wird es bei einer Relation. Eine Relation kann eigentlich eine von allen möglichen GeoJSON-Elementen sein. Da die Formate so unterschiedlich sind ist eine komplette Konvertierung sehr schwierig.

A way is considered a Polygon if

1. It forms a closed loop, and
2. it is not tagged `area=no`, and
3. at least one of the following conditions is true:
   - there is a `area=*` tag;
   - there is a `area:highway=*` tag and its value is not: `no`;
   - there is a `aeroway=*` tag and its value is not any of: `no`, or `taxiway`;
   - there is a `amenity=*` tag and its value is not: `no`;
   - there is a `barrier=*` tag and its value is one of: `city_wall`, `ditch`, `hedge`, `retaining_wall`, `wall` or `spikes`;
   - there is a `boundary=*` tag and its value is not: `no`;
   - there is a `building:part=*` tag and its value is not: `no`;
   - there is a `building=*` tag and its value is not: `no`;
   - there is a `craft=*` tag and its value is not: `no`;
   - there is a `golf=*` tag and its value is not: `no`;
   - there is a `highway=*` tag and its value is one of: `services`, `rest_area`, `escape` or `elevator`;
   - there is a `historic=*` tag and its value is not: `no`;
   - there is a `indoor=*` tag and its value is not: `no`;
   - there is a `landuse=*` tag and its value is not: `no`;
   - there is a `leisure=*` tag and its value is not: `no`;
   - there is a `man_made=*` tag and its value is not any of: `no`, `cutline`, `embankment` nor `pipeline`;
   - there is a `natural=*` tag and its value is not any of: `no`, `coastline`, `cliff`, `ridge`, `arete` nor `tree_row`;
   - there is a `office=*` tag and its value is not: `no`;
   - there is a `place=*` tag and its value is not: `no`;
   - there is a `power=*` tag and its value is one of: `plant`, `substation`, `generator` or `transformer`;
   - there is a `public_transport=*` tag and its value is not: `no`;
   - there is a `railway=*` tag and its value is one of: `station`, `turntable`, `roundhouse` or `platform`;
   - there is a `ruins=*` tag and its value is not: `no`;
   - there is a `shop=*` tag and its value is not: `no`;
   - there is a `tourism=*` tag and its value is not: `no`;
   - there is a `waterway=*` tag and its value is one of: `riverbank`, `dock`, `boatyard` or `dam`;

Abbildung 71: Wann ist ein Way ein Polygon ist[10]

Lösungsansätze Es gibt verschiedenen Libraries und Programme, welche die Konvertierung dieser zwei Formate übernehmen könnnen. `Ogr2ogr` ist eine C++ Applikation, die Dateien im .osm-Format lesen und als GeoJSON ausgeben kann. Unterstützt werden dabei Points, LineStrings, MultiLineStrings und Multipolygone. Eine andere Library wäre `osmtogeojson`. Sie basiert auf Node.js und man kann damit auch .osm-Dateien in GeoJSON wandeln. Unterstützt werden dabei Points und Multipolygone.

Umsetzung Da wir uns dafür entschieden haben Overpass einzusetzen und daher keine .osm-Dateien haben, sind die obigen Tools nicht sinnvoll nutzbar. Wir setzen auf eine eigene Konvertierung der Overpass-Daten mittels Python und dem GeoJSON Modul. Dabei haben wir uns dazu entschieden Relations von OpenStreetMap nicht zu unterstützen, da die Konvertierung relativ komplex wäre und für die Darstellung von einfachen POIs genügen meistens Nodes und Ways.

```
1 if type == 'node':
2     geometry = geojson.Point((element["lon"], element["lat"]))
```

Abbildung 72: Node zu GeoJSON

Wir iterieren über die erhaltene Liste von Elementen und filtern deren Typ. Die Nodes können dabei relativ einfach in GeoJSON-Points umgewandelt werden. Das temporäre Objekt wird dazu zu einem Point-Objekt.

```
elif type == 'way':
    coords: list = element['geometry']
    geometry = geojson.LineString([(coord['lon'], coord['lat'])
                                   for coord in coords])
    if len(coords) > 0 and \
            coords[-1]['lat'] == coords[0]['lat'] and \
            coords[-1]['lon'] == coords[0]['lon']:

        # is closed and not highway or barrier
        if ('highway' not in element['tags'] and 'barrier' not
            in element['tags']) or ('area' in element['tags'] and
                                    element['tags']['area'] == 'yes'):

            geometry = geojson.Polygon([[(coord['lon'], coord['lat'])
                                         for coord in coords]])
```

Abbildung 73: Way zu GeoJSON

Bei einem Way wird es schon komplizierter. Zuerst wird das temporäre Objekt zu einem LineString, weil die meisten Ways LineStrings sind. Danach wird geprüft ob der erste und letzte Node die gleichen Koordinaten haben um einen geschlossenen Ring darzustellen. Danach muss man noch überprüfen ob der Way die Tags `"highway=*"` oder `"barrier=*"` beinhalten, denn dann sind es meistens keine Polygone. Wenn dies nicht der Fall ist, wird das temporäre Objekt ein Polygon.

```
elif type == 'relation':
    center: dict = element['center']
    geometry = geojson.Point((center['lon'], center['lat']))
```

Abbildung 74: Relation zu GeoJSON

In der letzten Unterscheidung werden einfach die Mittelpunkte der Relations als Points erstellt. Damit muss keine Unterscheidung zwischen den vielen verschiedenen Arten von Relations gemacht werden.

```
features.append(geojson.Feature(
            id=element['id'],
            geometry=geometry,
            properties={'description': 'This is a test description',
                        'tags': element['tags']}
        ))

return geojson.dumps(geojson.FeatureCollection(features),
                     indent=4, ensure_ascii=False).encode('utf8')
```

Abbildung 75: Rückgabe `FeatureCollection`

Zum Schluss wird das temporäre Objekt zu einer Liste hinzugefügt und am Ende der Iteration werden die verschiedenen Elemente als `FeatureCollection` zurückgegeben. Dabei wird noch ein Zusatzattribut `"description"` angefügt um personalisierte Beschreibung der POIs im Frontend zu ermöglichen.

Ergebnis/Ausblick Die Umsetzung mit dem GeoJSON-Modul klappte problemlos. Wir haben für uns eine gute Variante gewählt, die es uns auch ermöglicht Anpassungen vorzunehmen, wenn wir dies benötigen. Die Umwandlung gestaltet sich als relativ komplexes Thema und ist noch nicht ganz geklärt.[10] Falls OpenStreetMap irgendwann auf ein anderes Datenmodell umsteigt kann man den Code anpassen. Allerdings muss man zuerst auf Overpass warten, die als Zwischenvermittler fungieren. Wir setzen momentan auf Overpass, da sie eine elegante Abfragesprache besitzen und wir nicht alles selber hosten müssen um die benötigten Daten zu bekommen.

8.2.3.3 OAuth mit OpenStreetMap

OpenStreetMap setzt auf OAuth 1.0 und 1.0a, wobei 1.0 nicht mehr eingesetzt werden soll. Die folgenden Links sind die wichtigen Punkte um eine gültige OAuth 1.0a Verbindung aufzusetzen:

- **Request Token URL:** https://www.openstreetmap.org/oauth/request_token
- **Access Token URL:** https://www.openstreetmap.org/oauth/access_token
- **Authorize URL:** https://www.openstreetmap.org/oauth/authorize

Man registriert eine neue Applikation unter
https://www.openstreetmap.org/user/username/oauth_clients/new
und kann dort alle Angaben zur Applikation angeben und die benötigten Rechte einstellen, die vom Benutzer gebraucht werden. Anschliessend erhält man einen Consumer-Key und ein Consumer-Secret zugeteilt, welche für die Verifikation gebraucht wird.

Problem Das Problem mit OAuth von OpenStreetMap ist, dass einmalig vergebene Access-Tokens ewig halten und dass bei jeder Anfrage eines Access-Tokens ein neuer Token generiert wird, anstatt auf schon vorhandene zu überprüfen, wie das z.B. Google mit OAuth 2.0 macht. Dadurch entsteht bei jedem Login eines Benutzers ein neues Access-Token und man kann mit diesem alleine nicht auf den Benutzer schliessen. Man muss zuerst noch die API von OpenStreetMap nach den Nutzerdaten (ID, Name) abfragen um den Nutzer identifizieren zu können. Dadurch entstehen viele Access-Tokens die unendlich lange gültig sind und in den Oauth-Settings des Benutzers angezeigt werden.

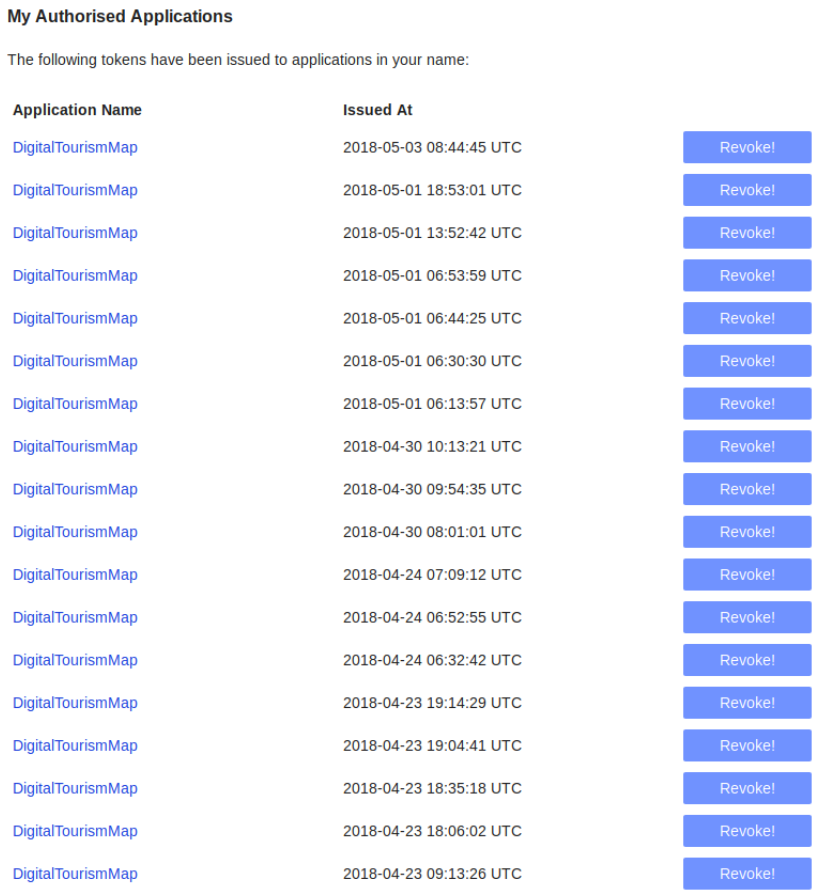

Abbildung 76: Mehrere Access Tokens

Lösungsansätze Grundsätzlich gibt es zwei Möglichkeiten. Man könnte ein eigenes Login-System einsetzen und müsste dann eigentlich nur OAuth von OpenStreetMap benutzen, wenn der Benutzer etwas hinzufügen möchte. Also wenn wir das Feature mit dem Einpflegen der vorhandenen Daten aus Excel o.Ä. in OpenStreetMap brauchen, könnte man es so umsetzen. Ansonsten würde man mit dem eigenen Login-System kein OAuth brauchen. Das ist aber kontraproduktiv für den Benutzer, welcher sich noch ein neues Passwort merken muss, was wir ihm eigentlich ersparen wollten. Zudem wäre unsere ganze Applikation komplizierter, da es ein eigenes Login-System bräuchte.

Die Variante ohne eigenem Login-System braucht OAuth für die Identifizierung der Nutzer. Man muss mit den verschiedenen Access-Tokens leben und der Benutzer müsste selbstständig die Tokens wieder entfernen. Sieht unschön aus, kann unter Umständen auch sicherheitsrelevant sein, aber ist am komfortabelsten für den Benutzer.

Wir haben uns für die zweite Variante entschieden, da eigentlich schon von Anfang an fest stand, dass sich der Benutzer kein neues Login merken muss und wir nehmen die negativen Auswirkungen dieser Entscheidung damit in Kauf.

Umsetzung mit RAuth `RAuth` ist eine Library für Python, welche auf `Requests` aufbaut. Sie ermöglicht eine einfache Consumer-Integration von OAuth 1.0/a und 2.0.

Zuerst muss ein Service-Container-Objekt erstellt werden:

```
1  from rauth import OAuth1Service
2
3  osm: OAuth1Service = OAuth1Service(
4      name='OSM',
5      base_url=' https://api.openstreetmap.org/',
6      request_token_url='https://www.openstreetmap.org/oauth/request_token',
7      access_token_url='https://www.openstreetmap.org/oauth/access_token',
8      authorize_url='https://www.openstreetmap.org/oauth/authorize',
9      consumer_key='XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX',
10     consumer_secret='XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX')
```

Abbildung 77: Service-Container-Objekt

Danach braucht man ein Request-Token. Dabei wird die Callback-URL dynamisch durch Flask erstellt und zeigt auf den Request Handler `user_oauth`.

```
1  request_token: Tuple[str, str] = osm.get_request_token(
2      params={'oauth_callback': url_for('user.user_oauth', _external=True)})
```

Abbildung 78: Request-Token

In der gleichen Methode werden das Access-Token und das Secret in das Session-Objekt von Flask gepackt, damit man diese nach der Weiterleitung wieder abfragen kann (Man benötigt die Request-Token um Access Token generieren zu können). Des weiteren wird der Benutzer an die entsprechende OpenStreetMap Adresse weitergeleitet.

```
session['request_token'] = request_token[0]
session['request_token_secret'] = request_token[1]
authorize_url: str = osm.get_authorize_url(request_token[0])

return redirect(authorize_url)
```

Abbildung 79: Authorize-URL

Nachdem der Benutzer die Autorisierung für die Applikation bestätigt hat, wird er durch OpenStreetMap auf unsere Callback-URL (`user/oauth`) mit dem OAuth-Verifier weitergeleitet. Diesen lesen wir mit dem Request-Token aus der Session aus und erstellen ein Auth-Session-Objekt, welches die Access-Tokens enthält.

```
verifier: str = request.args.get('oauth_verifier')
request_token: str = session.get('request_token')
request_token_secret: str = session.get('request_token_secret')

oauth_session = osm.get_auth_session(
    request_token,
    request_token_secret=request_token_secret,
    data={'oauth_verifier': verifier})
```

Abbildung 80: Access-Token

Durch dieses Session-Objekt kann man nun Requests an OpenStreetMap senden. Wir fragen den Benutzernamen und die ID des Benutzers ab um den Benutzer verifizieren zu können. Die XML-Antwort wird mit Hilfe von `ElementTree` geparst.

```
root: ElementTree = ElementTree.fromstring(
    session.get('https://api.openstreetmap.org/api/0.6/user/details').content)

id: int = root[0].attrib['id']
name: str = root[0].attrib['display_name']
```

Abbildung 81: User-Details von OSM

Aus den Benutzerdaten wird ein JSON-Web-Token generiert und zum Schluss wird man an eine Frontend-Route weitergeleitet, um dieses JWT an das Frontend zu liefern.

```
1 jwt = userservice.get_jwt(osm_id=osm_id, username=username,
2                           access_token=access_token,
3                           access_token_secret=access_token_secret)
4
5     splitted_url: List[str] = request.url_root.rsplit(':', 1)
6     return redirect(splitted_url[0] + ':' +
7                     current_app.config['FRONTEND_PORT'] +
8                     "/authsuccess?jwt=" + jwt)
```

Abbildung 82: JWT und Redirect zum Frontend

Ergebnis/Ausblick Wir haben OAuth 1.0a von OpenStreetMap erfolgreich in unserer Applikation nutzen können und versuchen es dem Benutzer so einfach wie möglich zu machen. Falls OpenStreetMap irgendwann auf ein neues System mit Wiedererkennung der Request-Token umsteigen würde, hätte man die Nachteile dieser Umsetzung nicht mehr. Allerdings kann man davon ausgehen, dass dies nicht so einfach umgesetzt wird.[11]

8.2.3.4 HTTP Authentifizierung

Problem Für die Authentifizierung der Benutzer im Backend muss man einen Mechanismus implementieren, da HTTP-Aufrufe stateless sind und der Server deshalb nicht weiss, wer jetzt diesen Aufruf macht. Damit Niemand, ausser der berechtigten Person, ihre Daten einsehen kann, muss die Identität des Aufrufers geprüft werden.

Lösung Für die Authentifizierung der Benutzer werden standardmässig serverseitige Sessions verwendet. Diese sind in der Datenbank des Servers gespeichert und in einem Cookie o.Ä. wird ein Session-Token mitgeschickt, welches dann für das Auslesen der Identität aus der Datenbank verwendet werden kann. Eine andere Möglichkeit bieten JSON-Web-Token (JWT). Ein geheimer Schlüssel wird verwendet um JSON-Data zu signieren/verschlüsseln und deshalb vor unerlaubter Veränderung zu schützen. Dieses JSON-Objekt beinhaltet meistens die Benutzer-ID und der Nutzer kann dann mittels einer Datenbankabfrage ermittelt werden. Eine neue Möglichkeit wäre auch Platform-Agnostic-Security-Token (Paseto), das viele Kritikpunkte an JWT verbessert.[5]

Umsetzung Bei der Umsetzung setzen wir auf JWT durch die Library `Flask JWT Extended` und einer SHA-256 Signatur. Paseto ist noch zu neu und die Kritikpunkte an JWT treffen nicht ganz auf uns zu. Deshalb wurde entschieden das Modul `Flask JWT Extended` einzusetzen, welches bereits die Implementation für Flask mit JWT-Authentifizierung anbietet.

```
# JWT Authentication Keyword
app.config['JWT_SECRET_KEY'] = 'XXXXXXX'

# For Header without Bearer keyword
app.config['JWT_HEADER_TYPE'] = ''

JWTManager(app)
```

Abbildung 83: Globale JWT Einstellungen für `Flask JWT Extended`

Um generelle Einstellungen am Modul vorzunehmen, setzt `Flask JWT Extended` auf die Konfiguration bei Initialisierung der Applikation. Der geheime Schlüssel wird vermerkt und normalerweise würde es dann schon laufen. Allerdings benötigt das Modul standardmässig das Keyword `Bearer` bei einer Anfrage im Authorization-Header und wir wollten auf dieses Keyword verzichten. Deshalb setzen wir manuell den Header-Type auf einen leeren String.

```
create_access_token(identity={'id': str(user.id)})
```

Abbildung 84: Erstellung eines JWT

Für die Erstellung eines JWT wird ein Objekt mit der Benutzer-ID erstellt und mit der Hilfsmethode in ein gültiges signiertes JWT gewandelt. Da die ID vom Typ UUID ist, muss sie zuerst noch in einen String für die Konvertierung zu JSON gecastet werden.

```
@jwt_required
def groups_get() -> Tuple[any, int]:
    """
    Get a list of groups of the user through JWT
    :return: The json data
    :rtype flask.Response
    """
    user_id = get_jwt_identity()['id']
```

Abbildung 85: Verwendung eines JWT im Backend

Mittels des Decorators wird automatisch die Signatur des JWT geprüft und eine entsprechende Fehlermeldung generiert, falls es fehlt oder nicht gültig ist. Danach kann man ganz einfach auf das darunterliegende Objekt zugreifen und dessen Eigenschaften abfragen.

Ergebnis/Ausblick JWT wird erfolgreich als Authentifizierungsmethode eingesetzt. Die Einbindung in Flask mittels `Flask JWT Extended` funktioniert sehr gut und das Modul bietet noch viele Erweiterungs- und Anpassungsmöglichkeiten, falls diese gewünscht sind. In einer nahen Zukunft könnte die Umsetzung auch mittels Paseto gemacht werden, falls die verfügbaren Tools erweitert werden.

8.2.3.5 OR-Mapping

Problem Das Datenmodell einer Datenbank ist relational und viele Programmiersprachen setzen auf Objektorientierung. Das sind zwei grundsätzlich verschiedene Strukturen und können nicht so einfach ausgetauscht werden. Man möchte die Wiederverwendbarkeit und Abstraktion für den Entwickler von einfachen Datenbankabfragen gewährleisten und setzt deshalb auf einen OR-Mapper. Dieser regelt das Mapping von relationalem Datenmodell zu Objekten. Da ein solcher Mapper mit vielen verschiedenen Objekten und Verbindungen umgehen können muss, gibt es auch immer wieder Probleme wie z.B. schlechte Performance.[6]

Lösung Für die einfachen SQL-Queries, die wir verwenden, reicht ein OR-Mapper und wir sollten nicht in generelle Probleme laufen. Durch den Einsatz von Flask sind wir fast völlig frei in der Wahl eines OR-Mappers. `SQLAlchemy` ist ein OR-Mapper der in etwa mit Hibernate vergleichbar ist. Eine andere mögliche Alternative ist `PeeWee`. `PeeWee` ist ein kleiner schlanker OR-Mapper mit wenigen Features.

Umsetzung Da Flask eigentlich immer zusammen mit `SQLAlchemy` eingesetzt wird und man viele Informationen diesbezüglich findet, haben wir uns für eine Umsetzung mit `SQLAlchemy` entschieden.

```
1 class Group(db.Model):
2     __tablename__ = 'group'
3     id = db.Column(UUID(as_uuid=True), primary_key=True,
4                    server_default=text("uuid_generate_v4()"))
5     name = db.Column(db.String(50), nullable=False)
6     lon = db.Column(db.Float)
7     lat = db.Column(db.Float)
8     zoom = db.Column(db.Float)
9     image = db.Column(db.String(255))
```

Abbildung 86: Gruppenmodell mit `SQLAlchemy`-Mapping

Die neue Klasse erbt von der, von `SQLAlchemy`, zur Verfügung gestellter Klasse `db.Model`. Dabei werden die Mappings automatisch definiert und aufgelöst. Die ID ist hierbei eine UUID und wird automatisch mit einer Datenbankfunktion generiert. Dabei muss auf PostgreSQL das Modul `uuid-ossp` geladen werden. Die anderen Attribute sind normale Datenbankattribute mit den jeweiligen Datentypen.

```
templates = db.relationship('TemplateMap', backref='group',
                            cascade="all, delete-orphan", lazy=True)
sharedmaps = db.relationship('SharedMap', backref='group',
                             cascade="all, delete-orphan", lazy=True)
```

Abbildung 87: Relationship-Mapping n-Teil

`templates` und `sharedmaps` definieren eine 1-n Beziehung. Eine Gruppe kann beliebig viele Vorlagen oder gesharte Karten besitzen. Das erste Attribut gibt dabei den Klassennamen der Referenz bekannt. Mittels der `backref` kann man auch von den Vorlagen auf die jeweilige Gruppe unter dem Attribut `group` zugreifen. Beim Löschen einer Gruppe werden auch deren Vorlagen und gesharte Karten gelöscht. Und wenn man dieses Objekt aus der Datenbank lädt werden nicht automatisch alle Vorlagen auch mitgeladen um die Performance nicht zu stark zu beeinträchtigen.

```
class TemplateMap(db.Model):
    id = db.Column(UUID(as_uuid=True), primary_key=True,
                   server_default=text("uuid_generate_v4()"))
    group_id = db.Column(UUID(as_uuid=True),
                         db.ForeignKey('group.id'), nullable=False)
    pois = db.Column(db.JSON)
    name = db.Column(db.String(50), nullable=False)
    descr = db.Column(db.String(255), nullable=True)
```

Abbildung 88: Relationship-Mapping 1-Teil

Auf der anderen Seite steht die Vorlagenklasse. Mittels der Spalte `group_id` wird ein Foreign-Key in der Tabelle definiert, der automatisch von `SQLAlchemy` auf die Gruppe gemappt wird.

```
users = association_proxy('group_users', 'user',
                          creator=lambda user: GroupUser(user=user))
```

Abbildung 89: Proxy-Mapping

Eine weitere Besonderheit, die viel Zeit gekostet hat ist das Association-Proxy. Um eine n-n Beziehung aufzulösen werden traditionell Zwischentabellen eingeführt, so dass jede Seite eine 1-n Beziehung hat. Weil ein Benutzer mehrere Gruppen und eine Gruppe mehrere Benutzer haben kann, haben wir das dort eingesetzt. Normalerweise kann man dies mit `SQLAlchemy` mittels eines Tabellenobjekts und zwei Relationships lösen. Allerdings verwenden wir auf der Zwischentabelle zusätzlich die Rolle des Benutzers. Daher muss man auch auf die Tabelle direkt zugreifen können und nicht nur auf die Beziehungen.

```
class GroupUser(db.Model):
    __tablename__ = 'group_user'
    userid = db.Column('userid', UUID(as_uuid=True),
                       db.ForeignKey('user.id'), primary_key=True)
    groupid = db.Column('groupid', UUID(as_uuid=True),
                        db.ForeignKey('group.id'), primary_key=True)
    role = db.Column('role', db.Enum('owner', 'admin', 'user',
                                     'pending', name='roles'))

    group = db.relationship('Group', backref=db.backref(
      'group_users', cascade="all, delete-orphan"))
    user = db.relationship('User', backref=db.backref(
      'user_groups', cascade="all, delete-orphan"))
```

Abbildung 90: Zwischentabelle

Die Zwischentabelle wird als eigenes Objekt und nicht nur als Tabelle erstellt. Die Gruppe und der Benutzer sind Foreign-Keys, welche auch wirklich in die Datenbank geschrieben werden. Die Attribute `group` und `user` sind wieder die n-Seite einer 1-n Beziehung und verweisen auf das Proxy-Attribut der jeweiligen Objekte. Mit diesem Vorgehen ist es möglich, die Zwischentabelle unsichtbar dazwischen zu haben und direkt über das Proxy auf die entsprechenden Beziehungen zuzugreifen. Wenn man explizit auf die Zwischentabelle zugreifen oder Queries ausführen will, kann man dies als ganz normales `SQLAlchemy`-Objekt machen.

```
def get_all_groups(self, user_id: uuid) -> List[Group]:
    return self.session.query(Group)\
        .join(*Group.users.attr)\
        .filter(Group.users.any(id=user_id))\
        .all()
```

Abbildung 91: Query SQLAlchemy

Die Query eines Association-Proxies ist nur mit dem `join` insofern speziell, dass man mittels eines Pointers automatisch die richtigen Attribute untereinander mappt und dann ganz gewöhnlich zugreifen kann. Wie man sieht ist die Abfrage mittels `SQLAlchemy` ähnlich zu einer gewöhnlichen SQL-Query.

Ergebnis/Ausblick Zu `SQLAlchemy` findet man viele Informationen und der OR-Mapper hat auch viele verschiedene Funktionalitäten, welche als Neuling überwältigend sind. In einer früheren Entwicklungsphase sah unser Datenmodell noch anders aus und setzte auf PostGIS und dessen Geometrien. Die Abbildung in `SQLAlchemy` gestaltete sich schwieriger als gedacht. Die Lösung mittels `GeoAlchemy2`, GeoJSON und dem WKT-Format Datenmodelle zu generieren und abzuspeichern funktionierte schlussendlich doch. Nach Problemen und Bedenken bezüglich dem Updaten von vorhandenen Daten und der Zugriffstrennung haben wir uns doch für ein einfacheres Datenmodell mit Redundanz entschieden. Dabei konnte die Umsetzung mittels `SQLAlchemy` einfach geändert werden. `SQLAlchemy` verfügt über ein automatische Migrationsfunktion, welche bei Schema-Änderungen sehr hilfreich sein kann und deshalb wird auch in naher Zukunft dieser OR-Mapper noch relevant sein.

8.2.3.6 Validierung der Nutzerdaten

Problem Bei einkommenden Requests wird ein JSON-Objekt mitgeschickt, welches dann massgeblich für die weitere Verarbeitung, z.B. bei der Erstellung einer neuen Gruppe, ist. Dabei können die Anfragen auch von einem versierten Nutzer selber ausgeführt werden. Das mitgeschickte Objekt muss nicht zwingend mit den Annahmen übereinstimmen und sollte daher überprüft werden.

Lösung Bei der Anfrage mit JSON-Inhalten muss das Objekt geprüft werden. `Flasgger` bietet an sich die Möglichkeit für die Validierung der einkommenden Daten mittels der OpenAPI-Spezifikation. Dabei könnte man sichergehen, dass das einkommende Objekt immer der Dokumentation entspricht und ist gezwungen die Dokumentation auf aktuellem Stand zu halten. Eine andere Möglichkeit bietet `Marshmallow`. `Marshmallow` ist ein Python-Modul, das komplexe Datentypen (Objekte) von und zu nativen Python-Datentypen umwandeln kann. `Marshmallow-SQLAlchemy` ist dabei die `SQLAlchemy`-Integration dieses Moduls und erlaubt die automatische Attributkonfiguration durch `SQLAlchemy`-Klassen.

Umsetzung Wir hatten Probleme mit der Validierungsfunktion von `Flasgger`, welche nicht mit den importierten Dateien innerhalb der YAML-Dateien umgehen konnte. Deshalb setzen wir `Marshmallow-SQLAlchemy` für die Validierung ein. Zusätzlich bietet `Marshmallow` die Funktion zur Konvertierung von komplexen Objekten und wir haben uns dazu entschieden eine Abstraktionsebene mit Viewmodels zu schaffen, bei der `Marshmallow` die Konvertierung übernimmt.

```
1 class UserSchema(ModelSchema):
2     class Meta:
3         model = User
```

Abbildung 92: Einfaches Viewmodel-Mapping

Die Annotation des Mappings gestaltet sich sehr einfach wenn das Viewmodel genau dem Datenmodell entspricht. Die Übernahme der Attribute übernimmt dabei das Modul `Marshmallow-SQLAlchemy`, welches die `SQLAlchemy`-Modelle als Vorlage benutzt und dann das Schema generiert.

```
1 @post_dump
2 def enrich(self, data: Dict):
3     group = self.session.query(Group).filter(
4       Group.id == data.get('group')).first()
5
6     if group is not None:
7         data['zoom'] = group.zoom
8         data['lon'] = group.lon
9         data['lat'] = group.lat
```

Abbildung 93: Komplexeres Viewmodel-Mapping mit Decorator

Wenn das Viewmodel nicht genau dem Datenmodell entspricht, muss man die Änderungen selber verwalten. Dies geschieht mittels den Decorators wie z.B. `post_dump` bei denen man individuelle Anpassungen im Mapping vornehmen kann, wie in Abbildung 93 beschrieben.

```
1 detailed_schema = GroupSchema(session=db.session(),
2                               user_id=user_id,
3                               exclude=('sharedmaps', 'templates', 'group_users'))
4 return jsonify(detailed_schema.dump(group_detail).data), 200
```

Abbildung 94: `Marshmallow` als Konverter

Der Gebrauch von `Marshmallow` als Konverter ist sehr einfach. Man kann die nicht benötigten Attribute ausblenden und das Modul kümmert sich um den Rest. Auch die Fehlerbehandlung wird von `Marshmallow` direkt übernommen.

```
1 errors = GroupSchema().validate(group)
```

Abbildung 95: `Marshmallow` als Validator

Die Validierung ist ähnlich zur Konvertierung und es werden einfach nur die vorhandenen Attribute überprüft.

Ergebnis/Ausblick Die Umsetzung der Validierung und der Konvertierung durch `Marshmallow` ging gut voran. Allerdings wäre die Lösung mit der Validierung von `Flasgger` viel praktischer, da man dann gezwungen wäre die Dokumentation aktuell zu halten. Man sollte ein Auge auf die Issues und Merge-Requests des Repos halten und in Zukunft eventuell auf `Marshmallow` verzichten falls das Problem mit dem Import behoben wird.

8.2.4 Static-Map-Server

Da, wie in Kapitel 8.1.4 geschrieben, das Drucken immer noch ein grosses Thema ist und die Auflösung eines gewöhnlichen Screenshots nicht ausreicht, haben wir einen anderen Service erstellt. Um eine möglichst hohe Druckauflösung zu bekommen, haben wir uns schon von Anfang an entschieden auf Vektorkacheln zu setzen, die in einem Vektorformat geschrieben sind und somit theoretisch eine unendliche Auflösung besitzen. Daher ist der Lösungsansatz diese Vektorkacheln in entsprechende Rasterkacheln umzuwandeln und danach den Layer mit den POIs darüberzulegen. Im Backend soll dazu die Konvertierung der Vektorkacheln und Generierung eines statischen Bildes der Karte umgesetzt werden.

Es gibt `Mapbox GL native`, das die Konvertierung dieser Kacheln möglich macht, allerdings bieten sie nur einen Node.js-Wrapper um die darunterliegende C++-Library an und keinen Python-Wrapper. Deshalb haben wir uns dazu entschieden einen eigenen Node.js-Express-Server einzurichten, der auf dem Mapbox-Wrapper basiert und die statischen Karten in der gewählten Auflösung erstellt. Mit Hilfe des Beispiels von Klokantech[7] wurde ein einfacher Server mit einer einzigen Route erstellt. Dieser erlaubt die Erstellung von Karten mit grosser Auflösung und dies hat einen direkten Einfluss auf die Performance. Momentan wird vom Frontend eine Auflösung von 3072 Pixel * 2172 Pixel benötigt um ein A4 mit einer DPI von ca. 260 zu erstellen und dies benötigt ca. eine Sekunde.

8.2.5 Testing

Beim Testen der API wurde das Standardmodul `unittest` verwendet. Dieses scannt den Testordner nach den Testdateien und führt alle Tests darin aus. Insgesamt wurden 53 Tests geschrieben, die eine Vielzahl an möglichen API-Aufrufen simulieren. Vor jedem Test wird die Datenbank neu befüllt und nach jedem Test wieder abgeräumt, so dass wir sicher gehen können, dass nur die nötigen Daten in der Datenbank zur Verfügung stehen. Dabei wird in den normalen Test-Docker-Container zugegriffen, allerdings auf eine spezielle Testdatenbank um nicht die normalen oder development Daten zu löschen.

Damit man eine kleine Übersicht hat, wie die Testabdeckung im aktuellen Projekt ist, wurde das Modul `coverage` eingesetzt. Dieses lässt alle Tests laufen und analysiert wie welche Dateien durchlaufen worden sind. Bei unserem Projekt haben wir momentan eine Testabdeckung von ca. 76%, die nach unserer Ansicht gut ist.

```
----------------------------------------------------------------------
Ran 53 tests in 13.113s

OK
Coverage Summary:
Name                                    Stmts   Miss Branch BrPart  Cover
----------------------------------------------------------------------
project/api/group/groups.py               120     23     14      2    78%
project/api/group/schema.py                57      2     32      7    88%
project/api/group/service.py               75     23     16      5    65%
project/api/models.py                      14      1      0      0    93%
project/api/pois/pois.py                   55     15     34      6    63%
project/api/sharedMap/schema.py            24      0      4      2    93%
project/api/sharedMap/service.py            9      0      0      0   100%
project/api/sharedMap/sharedMap.py         32      3      4      1    89%
project/api/template/schema.py             28      1      6      2    91%
project/api/template/service.py            28      1      4      1    94%
project/api/template/templates.py         103     17     16      1    82%
project/api/user/schema.py                  5      0      0      0   100%
project/api/user/service.py                30     15      4      0    44%
project/api/user/user.py                   42     22      0      0    48%
----------------------------------------------------------------------
TOTAL                                     622    123    134     27    76%
```

Abbildung 96: Test-Coverage

8.3 Docker und CI/CD

Für den Aufbau und die Wiederverwendbarkeit der Applikation wurde Docker und zusätzlich Docker-Compose eingesetzt.

8.3.1 Genereller Aufbau mit Docker-Compose

Mittels zwei Docker-Compose-Dateien, eine für die Entwicklungsumgebung und eine für die Produktivumgebung, können alle Docker-Container und deren Abhängigkeiten verwaltet werden. Dabei haben wir je einen Container für das Backend, das Frontend, die Datenbank und den Static-Map-Server. In der Entwicklungskonfiguration werden alle Umgebungen im Development-Modus mit Debugging gestartet und, falls vorhanden, das Entwicklungs-Docker-File geladen.

8.3.2 Datenbank Docker

Der Container basiert auf dem Standard PostgreSQL-Image und führt zusätzlich bei Erstellung SQL-Befehle aus um 3 verschiedene Datenbanken zu generieren (dev, prod, test).

8.3.3 Static-Map-Server Docker

Der Container basiert auf dem Node.js-6-Image, da die Abhängigkeit `Mapbox GL Native` höchstens mit Node.js 6 läuft, auch wenn die aktuellste Version 10 wäre. Da wir bei der Erstellung Mapbox selber builden müssen und dies eine C++ Applikation ist, sind verschiedenste Pakete hinzugefügt worden. Man kann nicht einfach so Pakete aktualisieren und wir haben eine andere Paketliste geladen, damit wir wirklich die neusten Pakete installieren können.

8.3.4 Backend Docker

Der Container basiert auf dem Python-3.6.4-Image. Die benötigten Python-Module werden mit `pip` und dem Requirements-File automatisch installiert. Beim Start des Containers läuft ein Script, welches überprüft ob die Datenbank so weit aufgestartet ist, dass sie Verbindungen akzeptiert. Mit Docker-Compose kann man die Abhängigkeiten zwar angeben, aber man kann nicht überprüfen ob der Container auch wirklich schon bereit ist und dies kann zu unschönen Fehlermeldungen führen, deshalb wurde dieses Script geschrieben.

8.3.5 Frontend Docker

Der Container basiert auf dem Node.js-9.6-Image. Mittels `yarn` werden alle Abhängigkeiten installiert und dann mit dem `create-react-app` gestartet. Beim produktiven Container wird ein Builder-Docker-Pattern eingesetzt. Zuerst werden alle Dateien mittels `yarn` gebildet und statisch abgelegt. Danach wird ein Nginx-Image gestartet, das dann diese statischen Dateien ausliefern kann. Die Responsetime ist dadurch im produktiven Container sehr gut.

8.3.6 CI/CD mit GitLab

GitLab bietet die Möglichkeit für eine Continous-Integration. Mittels YAML-Datei wurde die CI-Konfiguration vorgenommen und bei jedem Push auf den Master-Branch wird ein vollständiges Deployment ausgelöst. Zuerst werden die Container mit produktiven Einstellungen von einem Runner, der ein eigener Server oder, wie in unserem Fall, ein Runner von GitLab selber sein kann, gebildet. Nach erfolgreicher Erstellung der Container werden diese in die GitLab-Registry gepusht. Dort steht ein Speicherplatz von 10GB für Docker-Images zur Verfügung und kann privat anstatt offen auf dem offiziellen Docker-Registry gelagert werden. Nach dem erfolgreichen ersten Schritt werden alle Tests durchgeführt und wenn auch dieser Schritt erfolgreich verlaufen ist, wird eine Script ausgeführt, welches sich mit SSH auf unseren produktiven Server einloggt und die neusten Images von der GitLab-Registry lädt.

8.4 Security

8.4.1 Kritische Assets

Unter kritische Assets werden Werte oder Daten gesehen, welche einen zusätzlichen Schutz benötigen und besonders schützenswert sind. In unserer Applikation betrifft das:

- **JWT**, für die Authentifizierung
- **JWT Signaturschlüssel**, für das Signieren von JWTs
- **Passwort**, für SSH Zugriff auf produktiven Server
- **Access-Token + Secret**, für den API-Zugriff auf OpenStreetMap
- **CI-Build-Token**, für Zugriff auf die private Docker-Registry
- **Benutzername und ID aus OpenStreetMap**, für die Benutzerabfrage

Nachfolgend soll aufgeführt werden, welche Funktionen diese Assets haben und welche Schutzmassnahmen dafür angewandt worden sind.

JWT	Das Token wird für die Authentifizierung der Benutzer im stateless-HTTP für den Server anstatt einer Session verwendet. Darin enthalten ist eine UUID, des entsprechenden Benutzers. Damit lässt sich der Benutzer eindeutig zuweisen. Wenn sich jemand Zugriff auf das JWT verschaffen kann, könnte er im Namen des Benutzers API-Abfragen tätigen und so an alle Daten (zugehörige Gruppen, gespeicherte Vorlagen, Benutzernamen und ID aus OpenStreetMap) gelangen. Das JWT ist nur ein Tag gültig und es soll so verhindert werden, dass diese in Umlauf geraten können.
JWT Signatur-schlüssel	Der Signaturschlüssel wird für das signieren von JWTs verwendet. Damit kann sichergestellt werden, dass das JWT nicht durch einen Man-In-The-Middle geändert worden ist. Sollte dieser Schlüssel in unerlaubte Hände fallen, könnte man beliebige JWTs generieren und alle Daten der API ausfragen. Dazu müsste man aber die richtigen Benutzer-UUIDs erraten. Dieser Schlüssel wird sicher in einer Konfigurationsdatei aufbewahrt, ist nicht direkt in den Code geschrieben.
SSH Passwort	Das Passwort wird für die Anmeldung am produktiven Server verwendet um z.B. die neuesten Docker-Images aus der Registry zu laden. Bei unerlaubter Aneignung wäre es möglich den Server unter eigene Kontrolle zu bringen und z.B. für ein Botnetz zu verwenden. Um dem entgegen zu wirken, wird das Passwort als Secret-Variable im CI hinterlegt und man kann deshalb nur dort deren Wert nachlesen. In allen Build-Files wird nur darauf verwiesen.

OAuth Access-Token

Das Access-Token wird für den API Zugriff auf OpenStreetMap verwendet. Dieses wird bei jedem erneuten Anmelden neu erstellt und das alte bleibt noch erhalten. Damit kann die eindeutige Benutzeridentifikation via Benutzername und ID aus OpenStreetMap gemacht werden. Wenn ein altes oder neues Token in die falschen Hände geriete, könnte man damit alle OpenStreetMap-relevanten Daten auslesen und z.B. neue Punkte auf den anderen Benutzernamen einfügen oder ändern. Das Access-Token wird nicht in der Datenbank gespeichert, da die momentane Funktionalität der Applikation dies nicht benötigt. Falls die Anforderungen in Zukunft geändert werden und man z.B. auch eigene Punkte auf OpenStreetMap einpflegen möchte, müsste man sich überlegen wie man dieses Access-Token sicher speichert.

CI-Build Token

Das CI-Token ist für den Zugriff auf die private Docker-Registry von Gitlab. Falls dieses Token in den Umlauf gerät, könnte man auf die Docker-Images auf dem Repository zugreifen und eventuell schädliche Images pushen. Damit der Zugriff erschwert ist, wird es, ähnlich zum SSH Passwort, in einer Umgebungsvariable der Gitlab-Runner gespeichert.

OSM Benutzername und ID

Der Benutzername sowie die ID aus OpenStreetMap werden lokal in der Datenbank gespeichert. Dies dient der eindeutigen Identifikation des Benutzers beim Login mit OAuth. Da OpenStreetMap immer wieder ein neues Access-Token generiert (Siehe 8.2.3.3) fragen wir die OSM-API nach den Nutzerdaten (Name und ID) ab und vergleichen das mit der Datenbank. Wenn diese Informationen in falsche Hände geraten, kann man lediglich nachverfolgen was der Benutzer auf OpenStreetMap gemacht hat und man kann keine eigenen API-Calls absetzen.

8.4.2 Authentisierung

Die Authentisierung geschieht durch einen Abgleich des Benutzernamens und der ID aus OpenStreetMap mit der lokalen Datenbank. Ein anonymer Besucher loggt sich via OAuth 1.0a auf OpenStreetMap ein und es wird ein Access-Token erstellt, welches einmalig für die API-Abfrage nach den Benutzerdetails gebraucht wird. Nach diesem Login wird ein JWT erstellt, welches eine UUID für den jeweiligen Benutzer enthält. Dieses Token wird signiert und an den Client weitergeschickt. Bei jeder Anfrage mit Ausnahme der Login-Route, der POI-Route und der SharedMap-Route wird dieses JWT überprüft und der Benutzer anhand der UUID authentisiert.

8.4.3 Autorisierung

Die Autorisierung geschieht mittels Überprüfung der jeweiligen Rollen der betroffenen Gruppe. In jeder Route wird mittels einer Servicemethode die Berechtigung für die aktuelle Aktion überprüft. Dabei wird auf eine Rollenliste zugegriffen bei der der Rang der Rolle im Index der Liste abzulesen ist. Das heisst, dass höherwertige Rollen automatisch auch die Aufgaben der niederwertigen Rollen übernehmen können. Dies wird basierend auf dem Listenindex ermittelt. Ist für die aktuelle Aufgabe z.B. mindestens die Rolle des Index 1 nötig, können dies auch automatisch Rollen die einen Index >1 haben. Somit sind höherwertige Rollen sozusagen Supersets von tieferen Rollen.

```
1 min_role: str = permissions_roles.get(permission)
2
3 if not min_role \
4         or user not in group.users \
5         or roles_list.index(role.role) < roles_list.index(min_role):
6     raise TouristMapError(204, 'Group access forbidden', 403)
```

Abbildung 97: Auszug der Autorisierungsmethode

8.4.4 Rollenkonzept und Berechtigungen

Das Rollenkonzept sieht folgende Rollen vor, die sich hauptsächlich auf den Gruppenzugriff konzentrieren.

Pending	Benutzer mit der Rolle `pending` sind von einem Administrator oder einem Owner in die Gruppe eingeladen worden und dürfen lediglich die Einladung annehmen oder ablehnen.
User	Ein normaler Mitarbeiter hat die Rolle eines Users. Dieser darf eigene Vorlagen erstellen, Karten sharen und die Gruppendetails ansehen.
Administrator	Ein Administrator kann neue Benutzer zu der Gruppe hinzufügen oder aus der Gruppe entfernen. Des weiteren darf er den Gruppennamen ändern und ein Gruppenbild hochladen. Er darf User zu Administratoren befördern oder Administratoren zu Usern degradieren.
Owner	Wenn ein Benutzer eine neue Gruppe erstellt, wird er automatisch zum Owner. Nur dieser hat die Berechtigung die Gruppe zu löschen. Es kann minimal einen Owner geben und er darf die Rolle von anderen Benutzern zu Owner ändern oder die Rolle Owner einem Benutzer entziehen (solange es noch mindestens einen Owner gibt).

9 Projektmanagement

9.1 Rollen und Verantwortlichkeiten

9.1.1 Luca Schneider

Luca Schneiders Hauptaufgabe war die Implementierung des Backends. Er hat Routen für die API implementiert, die Datenbank und serverseitige Architektur aufgebaut. Er hat sich zusätzlich um das Docker- und dem CI/CD-System gekümmert.

9.1.2 Leo Zurbriggen

Leo Zurbriggen war für das ganze Frontend mit React zuständig. Er hat die Datenverwaltung im Frontend organisiert, die Kommunikation mit der API und alle Komponenten implementiert und gestaltet.

9.2 Infrastruktur

9.2.1 Versionierung

Für die Versionierung der Codebase wurde Git mit Gitlab.com eingesetzt. Frontend und Backend wurden im selben Repository gehostet, was Releases massiv vereinfacht.

Gearbeitet wurde mit Feature-Branches. Für jedes Feature wurde ein eigener Branch angelegt. Bei Bedarf wurde dieser mit einem Rebase wieder auf aktuellen Master-Stand angeglichen, bis das Feature fertig implementiert war und nach Absprache mit dem Teamkollegen dann selber in den Master-Branch gemerged wurde.

9.2.2 Testsystem

Da die Digital Tourist Map noch nicht für Benutzer verfügbar sein muss, wurde das Testsystem auf einem internen HSR-Server gehostet. Auf dem virtuellen Linux-Server wurde Docker installiert und das CI/CD so konfiguriert, dass neue Versionen automatisch auf diesen Server deployed werden.

9.3 Prozessmodell

Da für das Projekt ein bestimmter Zeitrahmen und Umfang vorgegeben war, konnte kein agiler Prozess angewendet werden. Als alternative bot sich der Rational Unified Process (RUP) an. RUP teilt ein Prozess in mehrere Phasen ein. Zu Beginn gibt es eine Inception-Phase, in welchem Projektziel und -vision festgelegt werden. In der Elaboration werden dann Anforderungen modelliert, Risiken erfasst und die Architektur beschrieben. Darauf folgt die Construction-Phase. Meist ist diese am längsten, da hier das Projekt iterativ umgesetzt wird.
Zum Schluss läuft das Projekt in die Transition-Phase, in welcher dafür gesorgt wird, dass das Projekt erfolgreich eingesetzt werden kann. Dies kann vieles beinhalten, wie beispielsweise Testing, Anleitungen und Dokumentation.

9.4 Meilensteine

9.4.1 MS1 End of Inception

Umfang:

- Vision
- Anforderungen (Funktional und Nichtfunktional)
- Szenarien (Kontext und Personas)
- Use Cases
- Gantt Chart (Grobplanung)

9.4.2 MS2 End of Elaboration

Umfang:

- GUI Mocks
- Use-Case Diagramm
- Architekturbeschrieb/Diagramm
- Risikoeinschätzung und Matrix
- Prototyp (Nur Durchstich durch Schichten)
- User Stories (Arbeitspakete)
- Gantt Chart (Feinere Planung mit Arbeitspaketen)
- Aufgesetzte Umgebung (Docker/CI/CD)

9.4.3 MS3 Protoyp Basic

Umfang:

- Lauffähiger Protoyp
- Karte anzeigen
- POI hinzufügen/entfernen
- Vorlage speichern
- Benutzer erstellen und Login

9.4.4 MS4 Export

Umfang:

- Sharing via QR-Code/E-Mail o.Ä.
- Linkgenerierung für lesenden Zugriff auf Karte
- Kartenleser mit POIs

9.4.5 MS5 Personalisierung

Umfang:

- Eigener Standort setzen
- Eigenes Logo hinzufügen
- Beschreibung zu POIs hinzufügen

9.4.6 MS6 Import

Umfang:

- Eigene Daten importieren (Excel)
- Anzeigen von POIs als Favoriten
- Ein und Ausblenden von eigenen POIs

9.4.7 MS7 End of Transition

Umfang:

- Aufgeräumter Code
- Code Freeze
- Zeit nachgeführt

9.5 Zeitplan

Die grobe Zeitplanung gibt eine Übersicht über die Meilensteine des Projekts. Die meiste Zeit wird von der tatsächlichen Entwicklung in der Constructionphase beansprucht. Die Meilensteine zeigen auf, bis wann welche Features umgesetzt werden.

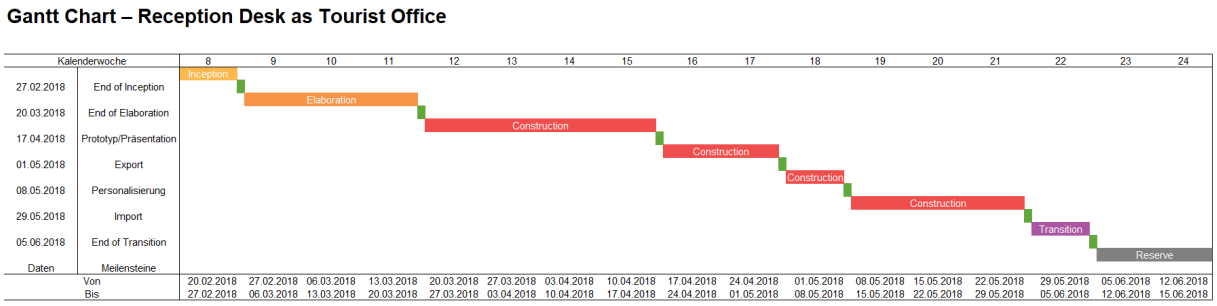

Abbildung 98: Zeitplanung grob

Zusätzlich zur Grobplanung wurde auch eine Feinplanung durchgeführt, in welcher die einzelnen Arbeitspakete innerhalb der Features aufgelistet wurde.

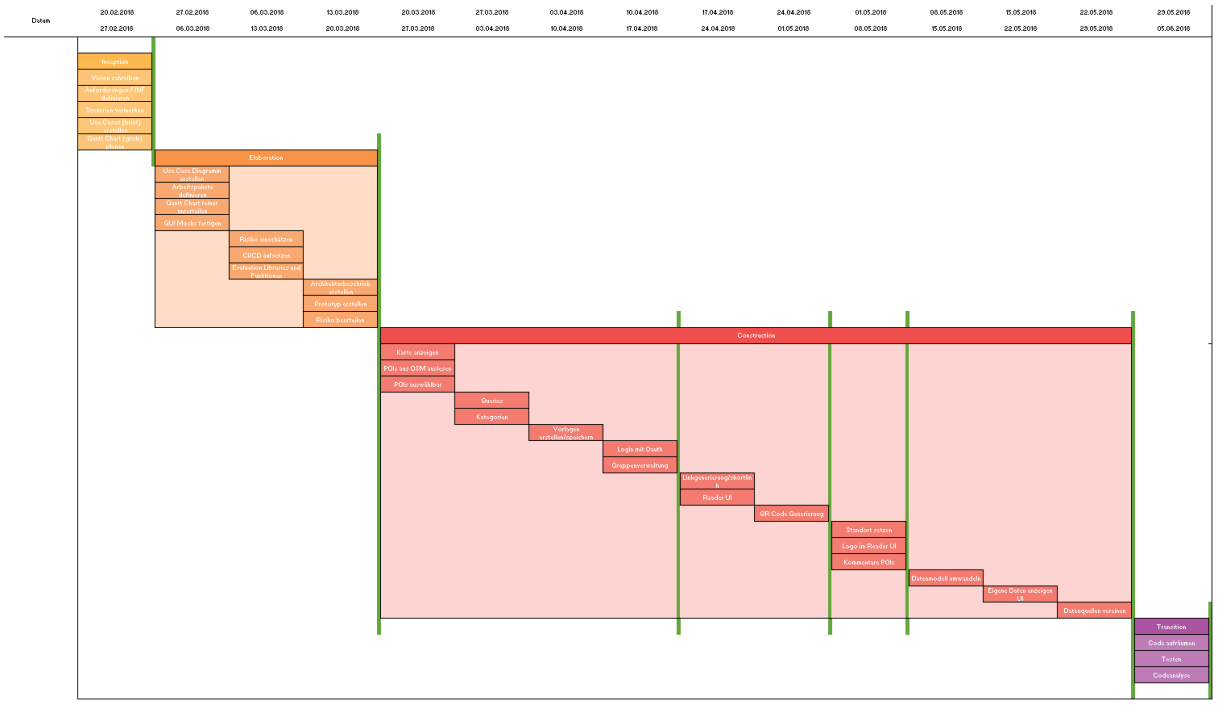

Abbildung 99: Zeitplanung fein

Die einzige grössere Änderung am Zeitplan, welche sich während dem Projekts ergeben hat, war der Austausch des Import-Features durch das Druck-Feature. Da dem Druck die gleiche Zeit zugeteilt werden konnte, musste der Zeitplan nicht angepasst werden. Ansonsten hat dieser Zeitplan sehr gut gepasst, was den gemachten Erfahrungen aus der Studienarbeit zu verdanken ist.

Die geplante Reserve wurde für administratives, Dokumentation und Anleitungen verwendet.

9.6 Risikoanalyse

Bei der Risikoanalyse wurden mögliche Risiken, die während dem Projekt auftreten könnten, identifiziert und bewertet. Die grössten Risiken waren hier technischer Natur und hätten bei Eintritt die Zeitplanung wesentlich beeinflussen können, weshalb die Risiken auch bei der Planung beachtet wurden.

Man muss sich an dieser Stelle immer bewusst sein, dass nie alle Risiken gefunden werden. Um relevante Risiken identifizieren zu können, ist viel Projekterfahrung nötig.

Nr.	Titel	Beschreibung	max. Schaden [h]	Eintritt %	Gewichteter Schaden [h]	Vorbeugung	Verhalten beim Eintreten
R1	Flask lernen	Einarbeitung und Umsetzung mit Flask benötigt mehr Zeit als gedacht.	20	40.00%	8	-Tutorials durcharbeiten - Kontinuierliche Weiterbildung	Der Zeitplan muss angepasst werden und zusätzliche Issues müssen auf Gitlab
R2	Infrastruktur aufbauen	Der Aufbau/die Anpassung der Infrastruktur benötigt mehr Zeit als gedacht	20	30.00%	6	-Tutorials durcharbeiten - Wissensaustausch mit Mitstudenten - Kontinuierliche	Der Zeitplan muss angepasst werden und zusätzliche Issues müssen auf Gitlab
R3	Gruppenverwaltung	Die Gruppenverwaltung ist nicht so einfach umsetzbar wie	10	10.00%	1	- Ähnliche Projekte durchlesen -Bereits in den ersten Phasen berücksichtigen	Der Zeitplan muss angepasst werden und zusätzliche Issues müssen auf Gitlab
R4	Basisfunktionalitäten Fr	Die Basisfunktionalitäten lassen sich nicht einfach mit den	15	5.00%	0.75	- Evaluation von Libraries - Prototyp erstellen	Der Zeitplan muss angepasst werden und zusätzliche Issues müssen auf Gitlab
R5	Basisfunktionalitäten Ba	Die Abfrage von OSM ist nicht so einfach oder qualitativ hochwertig wie	15	10.00%	1.5	- Prototyp erstellen	Der Zeitplan muss angepasst werden und zusätzliche Issues müssen auf Gitlab
R6	Import	Der Import muss an die Datei angepasst werden.	10	30.00%	3	- Den Aufbau so früh wie möglich berücksichtigen - Frühe Kontaktaufnahme mit Rapperswil Tourismus	Eventuell auf alle Extras verzichten. Neue Issues aufnehmen.
R7	Export	Der Export lässt sich nicht einfach umsetzen	15	30.00%	4.5	- Ähnliche Projekte studieren - Eventuell schon im Prototyp berücksichtigen	Eventuell auf alle Extras verzichten. Neue Issues aufnehmen.

Abbildung 100: Risikotabelle

Glücklicherweise konnten alle dieser Risiken vorgebeugt oder bereits im Zeitplan berücksichtigt und das Projekt planmässig umgesetzt werden. Risiko R6 (Import) konnte nicht eintreten, da das Feature durch den Druck ersetzt/nicht implementiert wurde.

9.7 Zeitauswertung

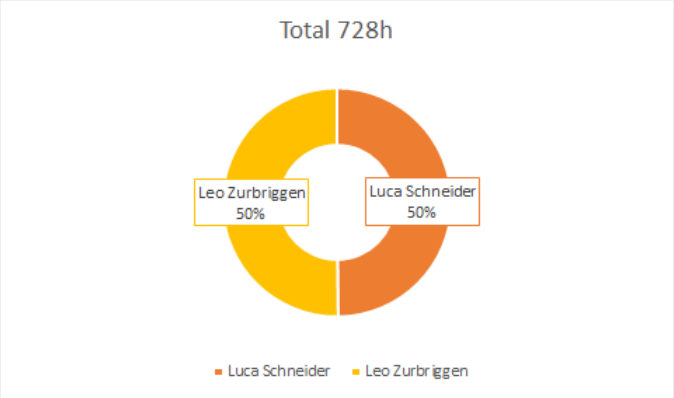

Abbildung 101: Zeitaufteilung nach Mitglieder

Wie man sieht, war die Zeit, welcher jeder von uns aufgewendet hat, ziemlich gleich. Es gibt einen kleinen Unterschied von vier Stunden, aber dieser hat prozentual fast keinen Einfluss. Diese gleiche Aufteilung kommt daher, dass wir viel zusammengearbeitet haben und einfach an den zwei bestimmten Tagen fast immer anwesend waren.

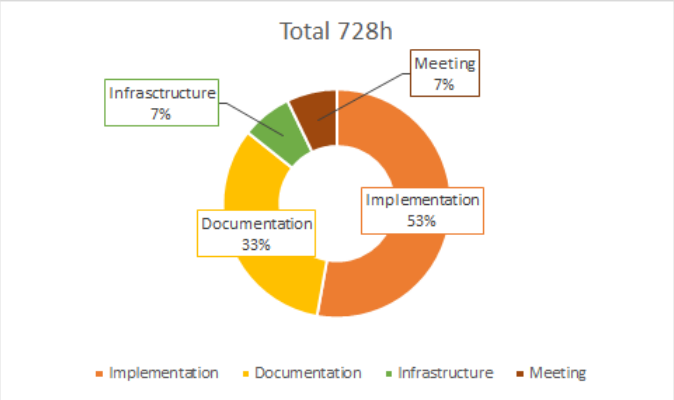

Abbildung 102: Zeitaufteilung nach Bereichen

Diese Zeitaufteilung zeigt, dass wir ca. die Hälfte der Zeit für das eigentliche Programmieren verwendet haben. Ein Drittel wurde für diese Dokumentation und alle daran beteiligten Texte im Wiki gebraucht. Für Infrastruktur (PC einrichten, Server aufsetzen, Docker) und Sitzungen haben wir jeweils 7% der Zeit, das heisst ca. 51 Stunden, benötigt.

10 Resultate und Ausblick

Das Projekt war ein Erfolg, da die in den Anforderungen erfassten Features und Ziele erreicht wurden. Eine Ausnahme stellt hier das geplante Datenimport-Feature dar, welches nach einem konstruktiven Gespräch mit Rapperswil Zürichsee Tourismus als kaum nutzbringend eingestuft und durch das betont wichtige Drucken ersetzt wurde. Nebst den Grundfeatures konnten zusätzlich noch weitere Features, wie beispielsweise die Internationalisierung implementiert werden.

Verbesserungs- und Erweiterungsmöglichkeiten finden sich unter anderem beim Druck und in der tatsächlichen Kartenzeichnungsansicht. So wären Tools denkbar, die es erlauben nebst einfachen Punkten auch Wege und Bereiche einzuzeichnen und mit Kommentaren zu versehen. Weiterhin wäre auch die Integration von externen Services denkbar. So könnten beispielsweise Fahrpläne und Eventkalender ihre Informationen direkt in der Digital Tourist Map darstellen und würden den Nutzern zusätzlich Arbeit sparen und Abläufe vereinfachen.

Da das Projekt ohne direkten Industriepartner umgesetzt wurde, gilt es nun ein Unternehmen zu finden, welches sich der Digital Tourist Map annimmt und für Hosting, Vermarktung und Weiterentwicklung sorgt.

Literaturverzeichnis

[1] [Online] Eran Hammer, Explaining the OAuth Session Fixation Attack, Available: https://hueniverse.com/explaining-the-oauth-session-fixation-attack-aa759250a0e7 [Accessed: 03-May-2018]

[2] [Online] Eaton Lists, Available: http://wiki.oauth.net/w/page/12238555/Signed_Callback_URLs [Accessed: 03-May-2018]

[3] [Online] The OAuth Bible, Available: http://oauthbible.com/ [Accessed: 03-May-2018]

[4] [Online] Eran Hammer, OAuth 2.0 and the Road to Hell, Available: https://hueniverse.com/oauth-2-0-and-the-road-to-hell-8eec45921529 [Accessed: 03-May-2018]

[5] [Online] Scott Arciszewski, No Way, JOSE! Javascript Object Signing and Encryption is a Bad Standard That Everyone Should Avoid, Available: https://paragonie.com/blog/2017/03/jwt-json-web-tokens-is-bad-standard-that-everyone-should-avoid [Accessed: 12-May-2018]

[6] [Online] Martin Fowler, OrmHate: https://martinfowler.com/bliki/OrmHate.html [Accessed: 12-May-2018]

[7] [Online] Klokan Technologies GmbH, TileServer GL: https://github.com/klokantech/tileserver-gl [Accessed: 25-May-2018]

[8] [Online] Martin Fowler, Richardson Maturity Model, Available: https://martinfowler.com/articles/richardsonMaturityModel.html [Accessed: 31-May-2018]

[9] [Online] RFC7946, The GeoJSON Format, Available: https://tools.ietf.org/html/rfc7946 [Accessed: 04-June-2018]

[10] [Online] OpenStreetMapWiki, Overpass turbo/Polygon Features, Available: https://wiki.openstreetmap.org/wiki/Overpass_turbo/Polygon_Features [Accessed: 04-June-2018]

[11] [Online] Do not ask for application OAuth permissions repeatedly, Available: https://github.com/openstreetmap/openstreetmap-website/issues/1455 [Accessed: 12-June-2018]

Abbildungsverzeichnis

Glossar

API Application Programming Interface, Eine Schnittstelle um Applikationen miteinander zu verbinden oder mit Programmbibliotheken zu interagieren.

Decorator Pythonkonstrukt für die Möglichkeit Funktionalität einer Funktion, Methode oder Klasse dynamisch zu ändern ohne Subklassen zu nutzen.

Deployment Bezeichnet den Vorgang um die Applikation einsatzfähig zu machen, normalerweise beinhaltet dies eine Installation auf einem oder mehreren Servers..

Digital Tourist Map Eine Applikation für Tourismusunternehmen um digitale Karten schnell und einfach für den Gast bereitzustellen..

Django Fullstack-Webframework für Python.

Docker Open Source Software zur Isolierung von Anwendungen mit Containervirtualisierung.

Docker-Compose Tool um Applikationen, welche mehrere Docker-Container benötigen, zu betreiben.

Docstring Python-Begriff für 3x" um einen speziellen Kommentar für das Dokumentieren zu erstellen.

Errorhandler Methode, die aufgerufen wird, falls irgendwo im Webframework ein Fehler geworfen wird.

Express Webframework basierend auf Node.js.

Flasgger Python Modul für Flask-Applikationen um OpenAPI-Spezifikationen zu erstellen.

Flask Microwebframework auf Basis von Python.

Flask JWT Extended Ein Flask-Modul für die Unterstützung von JSON-Web-Token mit zusätzlichen Funktionalitäten..

Flow Ein statischer Typchecker für Javascript.

GeoJSON Offenes Format für die Spezifikation von Geodaten.

gesharte Karte Karte, welche mit dem Kunden geteilt worden ist (via QR-Code oder Link).

GitLab Webanwendung zur Versionsverwaltung von Sofware auf Basis von git mit diversen Funktionalitäten.

Gruppe Zusammenschluss von Benutzern, meistens in der gleichen Organisation. Spezielle Gruppenzugriffs-Richtlinien regeln die Autorisierung.

Gunicorn Ein Python WSGI HTTP Server für Unix.

hot–Reloading Injecten von neuem Code während die Applikation läuft um die Änderungen mit erhaltenem State zu sehen.

IETF Internet Engineering Task Force, Organisation für die technische Weiterentwicklung des Internets.

JWT JSON-Web-Token, Access-Token basierend auf JSON.

Mapbox Open Source Mapping-Plattform mit vielen Features.

Marshmallow Python Modul um komplexe Datentypen in und von nativen Python-Objekten zu erstellen.

MIME–Type Multipurpose Internet Mail Extension gibt an welche Art von Daten gesendet werden.

Node.js Serverseitige Plattform für Softwareentwicklung auf Basis von JavaScript.

OAuth Open Authorization, Protokoll für sichere API Autorisierung.

OpenAPI Neue Benennung von Swagger.

Openpoiservice API für das querien von OpenStreetMap-Daten, gruppiert nach den vordefinierten POI-Kategorien.

OpenStreetMap Freies Projekt, das frei nutzbare Geodaten sammelt und zur Verfügung stellt.

OR–Mapper Middleware, die Objekte in relationalen Datenbanken abbildet und Zugriff über einen Wrapper bietet.

OSM Kurzform für OpenStreetMap.

Overpass API für das querien von OpenStreetMap-Daten.

Paseto Platform-Agnostic-Security-Token, verbesserte Version von JWT.

POI Bezeichnet den Vorgang um die Applikation einsatzfähig zu machen, normalerweise beinhaltet dies eine Installation auf einem oder mehreren Servers..

Polyfill Code, welcher zum Einsatz kommt wenn ein Browser ein benötigtes Feature nicht

unterstützt..

PostgreSQL Freies objektrelationales Datenbankmanagementsystem.

Provider Im Zusammenhang mit OAuth, Anbieter eines OAuth-Dienstes.

Pyramid Minimales Webframework für Python.

QR-Code Ein quadratischer Code, welcher kleine Datenmengen wie etwa einen Link beinhalten und z.B. mit Smartphones gelesen werden kann..

Query Eine Anfrage welche Daten zurückliefert. Beispielsweise gerichtet an eine Datenbank oder eine Web-API.

React Eine von Facebook entwickelte JavaScript Library zum Entwickeln von Benutzeroberflächen..

Redux Redux ist eine JavaScript State-Verwaltungslibrary.

RESTful Representational state transfer, Programmierparadigma.

RUP Rational Unified Process, Vorgehensmodell zur Softwareentwicklung.

Session-Fixation-Attacke Attacke um eine valide Benutzersession zu hijacken.

SQAlchemy OR-Mapper für Python.

SQLAlchemy Python Modul, ORMapper um Datenbank-Modelle in Python-Objekte darzustellen.

Superset Im Kontext: Programmiersprachen, welche eine andere Sprache erweitern..

Swagger Framework für die Beschreibung von RESTful-APIs, Schema-Sprache in YAML-Dateien.

SwaggerUI Benutzeroberfläche zu einer Swagger-Spezifikation.

Transpiler Ähnlich einem Compiler, das Resultat ist jedoch kein Maschinencode sondern Code in anderen Sprachen auf einem ähnlichen Abstraktionslevel..

UUID Universally Unique Identifier, Standard für Identifikatoren von Ressourcen.

View Eine View ist eine einzelne Ansicht oder Seite im User-Interface.

Vorlage Kartenvorlage. Ermöglicht schnellen Zugriff auf verschiedene POIs um dem Kunden effizient eine personalisierte Karte anzubieten.

WSGI Web Server Gateway Interface, Schnittstellen-Spezifikation für Python zwischen Webservern und Webframeworks.

YAML Superset von JSON, Ziel soll die einfache Lesbarkeit für Menschen sein.

Anhang

Benutzeranleitung

Vorraussetzungen

- Docker
- Docker-Compose
- Node
- Yarn
- Python 3

Entwicklungsumgebung

Vor dem Start

Beim Start der Development-Container werden die lokal installierten NPM-Packages in die Container kopiert, deshalb müssen diese zuerst installiert werden:

```
1 cd services/frontend
2 yarn install
```

Abbildung 103: Installieren der Node Modules

Starten des Projekts

```
1 docker-compose up -d
```

Abbildung 104: Starten der Umgebung im Entwicklungsmodus

- Frontend erreichbar unter `http://localhost:3000`
- Backend erreichbar unter `http://localhost:40001`
- Static-Map-Server erreichbar unter `http://localhost:40002`
- Swagger erreichbar unter `http://localhost:40001/apidocs`

CLI Commands

Während die Container laufen:

```
# Initialisiert die DB mit den Models
docker-compose run backend python manage.py recreate_db

# Schreibt default values in die DB
docker-compose run backend python manage.py seed_db

# Lässt die Unit-Tests laufen
docker-compose run backend python manage.py test

# Lässt einen Coverage Report laufen
docker-compose run backend python manage.py cov
```

Abbildung 105: CLI Commands welche mit Python zur Verfügung stehen

Produktionsumgebung

Adressen-Anpassung

Im Dockerfile `services/frontend/Dockerfile-prod` müssen die Environment-Variablen je nach Host angepasst werden. Läuft der produktive Server lokal:

```
ENV REACT_APP_API_URL=http://localhost:40001
ENV REACT_APP_STATIC_MAP_API_URL=http://localhost:40002
```

Abbildung 106: Anpassung der Umgebungsvariablen

Starten des Projekts

```
1 docker-compose -f docker-compose-prod.yml up -d
```

Abbildung 107: Starten der Umgebung im Produktivmodus

- Frontend erreichbar unter `http://localhost:80`
- Backend erreichbar unter `http://localhost:40001`
- Static-Map-Server erreichbar unter `http://localhost:40002`
- Swagger erreichbar unter `http://localhost:40001/apidocs`

Nutzwertanalyse Maplibraries

		Nutzwertanalyse											
Bewertungskriterium	**Gewichtung in %**	**Libraries**											
		Leaflet			**react-map-gl**			**react-mapbox-gl**			**react-geo**		
		Bew.	**Nutzw.**	**Kommentar**	**Bew.**	**Nutzw.**	**Kommentar**	**Bew.**	**Nutzw.**	**Kommentar**			
Implementationsaufwand	30%	60	18	Für die Nutzung von Vector Tiles müsste ggF. ein eigener Wrapper geschrieben werden	100	30	Angenehme API mit auf React ausgerichteter Funktionalität	80	24	Änderungen an der Karte (Move, Zoom, ...) müssten manuell über mapbox-gl-js Events implementiert werden	50	15	Hoch. Viele externe Abhängigkeiten. Weitere Hürden und Abhängigkeiten für Anzeige von Vektordaten. Nicht 100% idiomatisches React.
Verbreitung und Community	20%	100	20	Mit Abstand am verbreitetsten, bekanntes Produkt und aktive Community	75	15	Populärster React-Wrapper, Entwicklung von Uber	60	12	Weniger populär, dennoch aktive Entwicklung	25	5	Kaum verbreitet, jedoch aktive Entwicklung.
Lizenz	20%	80	16	2-Clause BSD License. Freie kommerzielle Nutzung	80	16	MIT + 3-Clause BSD. Freie kommerzielle Nutzung	80	16	MIT + 3-Clause BSD. Freie kommerzielle Nutzung	80	16	2-Clause BSD License. Freie kommerzielle Nutzung
Dokumentation	30%	100	30	Umfangreich, sauber und aktuell	100	30	Übersichtlich da kleinerer Funktionsumfang, aktuell	80	24	Im Markdown, eher unübersichtlich aber klar und aktuell	80	24	Saubere Dokumentation und Einleitung zum Einstieg, jedoch wenig aussagekräftige Beispiele
Nutzwertsumme			**84**			**91**			**76**			**60**	