Wlog

„Wlog – World Log, erstelle deinen individuellen Reiseblog und teile diesen in einer interaktiven Karte mit anderen Nutzern. Verfolge die Reisen anderer und lasse dich von ihren Erlebnissen inspirieren.“

Software-Projekt Medieninformatik (MI7)
Teammitglieder:

Name	Kürzel	Matrikel-Nr.
Agil Homar Franquesa	ah210	37478
Christian Heinz	ch148	37562
Jonas Leitner	jl121	37427
Peter Orbok	po012	37495
Sascha Schörnick	ss486	37423
Salome Wecks	sw193	37495

Inhaltsverzeichnis

1. Projektbeschreibung

Unser Projekt soll Benutzern eine Blog-Plattform bieten, um Bilder und Berichte ihrer Reisen zu teilen und auf einer interaktiven Karte zu referenzieren. Die Reisen werden als Trips kategorisiert und Benutzer können besuchte Orte und Länder auf ihren Trips angeben und weitere Berichte bzw. Posts mit Bildern zu den Orten hochladen.
Nach der Registrierung kann ein User sich einloggen und wird auf sein Profil geleitet. Das angezeigte Profilbild und die Profilbeschreibung können nach Belieben über “Edit” verändert werden. Unter dem Usernamen befindet sich eine Anzahl der Claps, welche eine Art “Like-Funktion” im Bezug auf die Trips darstellen. Weiter unten werden entweder bisher erstellte Trips mit ihren Thumbnails als Übersicht dargestellt, oder die Posts eines Trips auf einer interaktiven Karte verlinkt. Hierüber können die jeweiligen Trips aufgerufen werden. Eine Trippage zeigt den Titel und eine Kurzbeschreibung des Trips, sowie eine Karte mit den zugehörigen Blog Posts. Mit einem Klick auf das Map-Icon eines Posts in der Karte öffnet sich die Gesamtbeschreibung mit Bild und Text von diesem weiter unten. Rechts oben befindet sich die Like-Funktion, dargestellt als “Claps”.
Um einen solchen Trip mit Post oder nur einen Post zu einem zuvor kreierten Trip zu erstellen, wählt ein User entweder den Menüpunkt “Create” oder den gleichnamigen Button auf seinem Profil. In einem Pop-Up-Fenster kann der User nun eine der beiden Möglichkeiten auswählen. Anschließend müssen in den Input-Feldern alle benötigten Daten angeben werden bevor diese gespeichert und angezeigt werden können. Bei einem Trip den Titel, Kurzbeschreibung, Bild und Land. Bei einem Post die Caption, ausführliche Reisebeschreibung und Ort.
Andere User können über den Menüpunkt “Explore” über die dort implementierte Suchfunktion gefunden werden. Außerdem werden hier die zuletzt erstellten Trips in einem Slider angezeigt und auf einer Community Map alle Posts gekennzeichnet. Über Thumbnail bzw. Map-Icon können Trips wie oben beschrieben wieder aufgerufen werden.

2. Projektstrukturierung

Zu Beginn haben wir uns überlegt, wie wir unser Projekt am besten planen, managen und umsetzen können. Dabei sind wir schnell zu dem Schluss gekommen, dass wir zwei feste Termine pro Woche benötigen, an denen Besprechungen stattfinden. Zuerst hatten wir diese nicht, was jede Woche zu neuen Diskussionen nur über den Termin führte. Da zur Versionierung das hausinterne HdM GitLab verwendet wird, entschieden wir uns das GitLab Issue Board zur Aufteilung der Aufgaben und

Organisation zu verwenden. Hierdurch konnten wir direkt zum Projekt optimal ToDo's und die Aufgabenteilung managen, ohne die Plattform wechseln zu müssen.
Nachdem wir einige ToDo's gesammelt haben und einen Überblick hatten was zu tun ist, haben wir uns dafür entschieden eine Aufteilung in zwei Subteams, in Frontend und Backend, vorzunehmen. Da die gewünschten Technologien die wir zur Entwicklung verwenden wollten für jeden aus dem Projekt neu waren, half dies sich konkreter auf den gewünschten Bereich zu spezialisieren und diesen tiefer zu erlernen. Um mit der Umsetzung erster Aufgaben zu Beginnen, war es wichtig die gleichen Voraussetzungen für jedes Teammitglied zu schaffen. Deshalb wurde vorerst, unter gegenseitiger Hilfestellung im Team, die Entwicklungsumgebung bei jedem Teammitglied eingerichtet und die entsprechende Software und Frameworks installiert und in Betrieb genommen.
Damit jedes Teammitglied an individuellen Stellen und Problemen des Projektes arbeiten kann, und es dennoch immer einen funktionierenden Master Branch gibt, haben wir uns dazu entschieden, dass jeder seinen eigenen Branch mit dem dazugehörigen ToDo erstellt. Wir wählten das Format 'mitglied#01' wobei die ID des ToDos im GitLab Board verwendet wurde. Nach erfolgreichem Bearbeiten wurde ein Merge Request erstellt, welcher von den anderen Mitgliedern überprüft, bestätigt bzw. "approved", und letztendlich in den Master Branche gemerged wurde.
Da ein Projekt ohne Dokumentation und Austausch von Informationen nicht funktionieren kann, entschieden wir uns zudem das GitLab-Wiki zu verwenden. Mit diesem war es uns möglich Informationen, wie beispielsweise die Endpunkte zwischen Backend und Frontend, möglichst simpel auszutauschen bzw. zu dokumentieren.

2.1 Übernommene Aufgaben

Agil Homar	CI/CD, Video für MediaNight, Backend (Tests)
Christian Heinz	Frontend (Design, Implementierung, Features)
Jonas Leitner	Projekt Setup mit Docker, Backend (Api, Tests)
Peter Orbok	Präsentation am MI-Präsentationstag
Sascha Schörnick	Frontend (u. a. Tests, Dokumentation)
Salome Wecks	Frontend (Validierungen, Tests, Dokumentation)

3. Projekt Setup

Um die plattformübergreifende bzw. betriebssystemunabhängige Entwicklung und Nutzung bei unserem Projekt zu gewährleisten, verwenden wir Container-Virtualisierung mittels der freien Software Docker.

3.1 Frontend

Für das Frontend verwenden wir React. Die React Applikation wurde mit der Create React App aufgesetzt. Hierdurch kann der Development Server verwendet werden. Dieser ermöglicht eine simple Entwicklung, da jede getätigte Änderung direkt angezeigt bzw. direkt kompiliert wird. Damit dies möglich ist, wird ein Volume zwischen lokalem Rechner und Container verwendet. Die verwendeten Module werden über npm verwaltet und beim Build automatisch installiert.
Image: node:14-alpine
Port: 3000:3000

3.2 Backend

Das Backend, die REST-API ist in Python implementiert und verwendet das Webframework Flask. Die Flask Environment ist ebenfalls auf Development gesetzt, um ausführliche Fehlermeldungen zu erhalten, damit die Änderungen an Dateien sofort erkannt werden und der Server automatisch neu gestartet wird. Es wird ebenfalls ein Volume zwischen lokalem Rechner und Container verwendet. Benötigte Pakete stehen im *requirements.txt* File und werden mithilfe von pip beim Build automatisch installiert.
Image: python:3.7-alpine
Port: 5000:5000

3.2.1 Datenbank

Die Api verwendet als Datenbank eine MariaDB. Eine relationale Datenbank bietet sich für unser Datenmodell am besten an. Um die Daten zu Persistieren wird ein Docker Volume verwendet. Somit hat jeder User Zugriff auf seine individuelle und persistente Datenbank.
Image: mariadb:10.4

Um die Verwaltung der Datenbank effektiv und simpel zu gestalten, verwenden wir zudem PhpMyAdmin.
Image: phpmyadmin/phpmyadmin:latest
Port: 8000:80

Die vier verschiedenen Container agieren mithilfe des Docker Compose Tools als zusammengehörende Applikation. Die genannten Konfigurationen stehen im *docker-compose.yml* File. Für das Frontend und die Api gibt es zudem noch eigene Dockerfiles, die sich in den jeweiligen Unterordnern **frontend** und **backend** befinden. Mit Hilfe von diesen werden die Abhängigkeiten über die Paketmanager installiert.

3.3 Continuous Integration

Unsere Pipeline zur Continuous Integration haben wir mit Hilfe von GitLab CI realisiert. Dazu haben wir zwei Stages in *.gitlab-ci.yml* definiert. Für das Backend wird das Image *python:3.7-alpine* verwendet. Im Frontend ist es das Image *node:14*, da hier die *alpine* Version nicht ausreichend war.

build
Im building Stage werden für das Backend und das Frontend jeweils alle zugehörigen Dependencies und Packages mit Hilfe von *pip* bzw. *npm* installiert. Es soll überprüft werden, ob die reine Installation möglich ist.

validate
Hier werden die Dateien im Front- und Backend mit Hilfe von Lint-Tools auf korrekte Validierung geprüft. Für das Frontend nutzen wir *ESLint* für alle Dateien mit der Endung *.js* oder *.jsx* im src-Directory. Die Regeln zur Validierung sind in der Konfigurationsdatei *.eslintrc.json* aufgeführt.
Für das Backend nutzen wir *Flake8* zur Validierung aller Python-Dateien. Hier gibt es unter anderem die Konfigurationsdatei *setup.cfg* im Ordner */backend*.

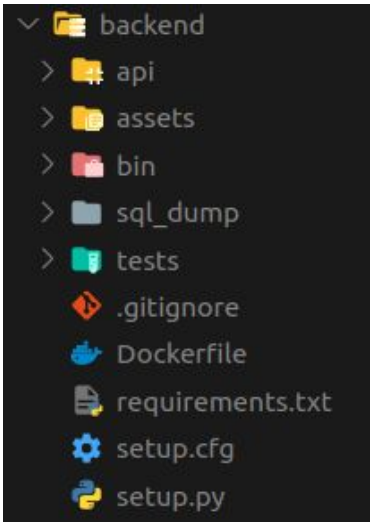
Eine Test Stage ist auch angedacht, in der die Unit Tests für das Frontend und Backend ausgeführt werden. Hierfür müsste allerdings docker-compose im Runner ausführbar sein. Da wir die Shared Runner der HdM verwenden und dies aus Sicherheitsgründen deaktiviert ist, müssen die Tests momentan weiterhin lokal ausgeführt werden.

4. Backend

4.1 Allgemeines

Das Backend ist eine REST API, welche in Python implementiert ist und das Webframework Flask verwendet. Die API bildet unser Backbone, welches mithilfe einer MariaDB die komplette Geschäftslogik abbildet. Über die Endpunkte ist es möglich sich zu registrieren, und nach erfolgreichem Authentifizieren das eigene Profil zu bearbeiten oder auch Trips zu erstellen. Zudem können diese von anderen Nutzern angesehen werden. Um die API auf ihre Funktionalität zu Testen gibt es zudem mit pytest implementierte Unit Tests.

Ordnerstruktur:
Das Backend hat einen **api** Ordner, in diesem befindet sich die eigentliche Applikation. Im **assets** Ordner werden die von Benutzern hochgeladenen Bilder gespeichert. **bin** beinhaltet Skripte zum simplen Konfigurieren, Validieren und Testen der Applikation. Die SQL-Skripte für die Datenbank, welche zum Aufsetzen oder auch zum Testen der Api verwendet werden, befinden sich im Ordner **sql_dump.** Die Unit Tests der API sind unter **tests** zu finden. Das **requirements.txt** File beinhaltet alle zusätzlichen Abhängigkeiten die für die Applikation benötigt werden. Das **setup.py** File ermöglicht es, die API als Python Package zu installieren.



4.2 Daten Schema

Um unsere Logik abzubilden, und die dritte Normalform einzuhalten, verwenden wir vier verschiedene Tabellen in unserer relationalen Datenbank. Jede Tabelle enthält eine ID welche als Primary Key dient. Die ID wird automatisch inkrementiert. Zudem beinhaltet jede Relation ein *created_at* Attribut, um nachzuvollziehen wann der Eintrag erstellt wurde.

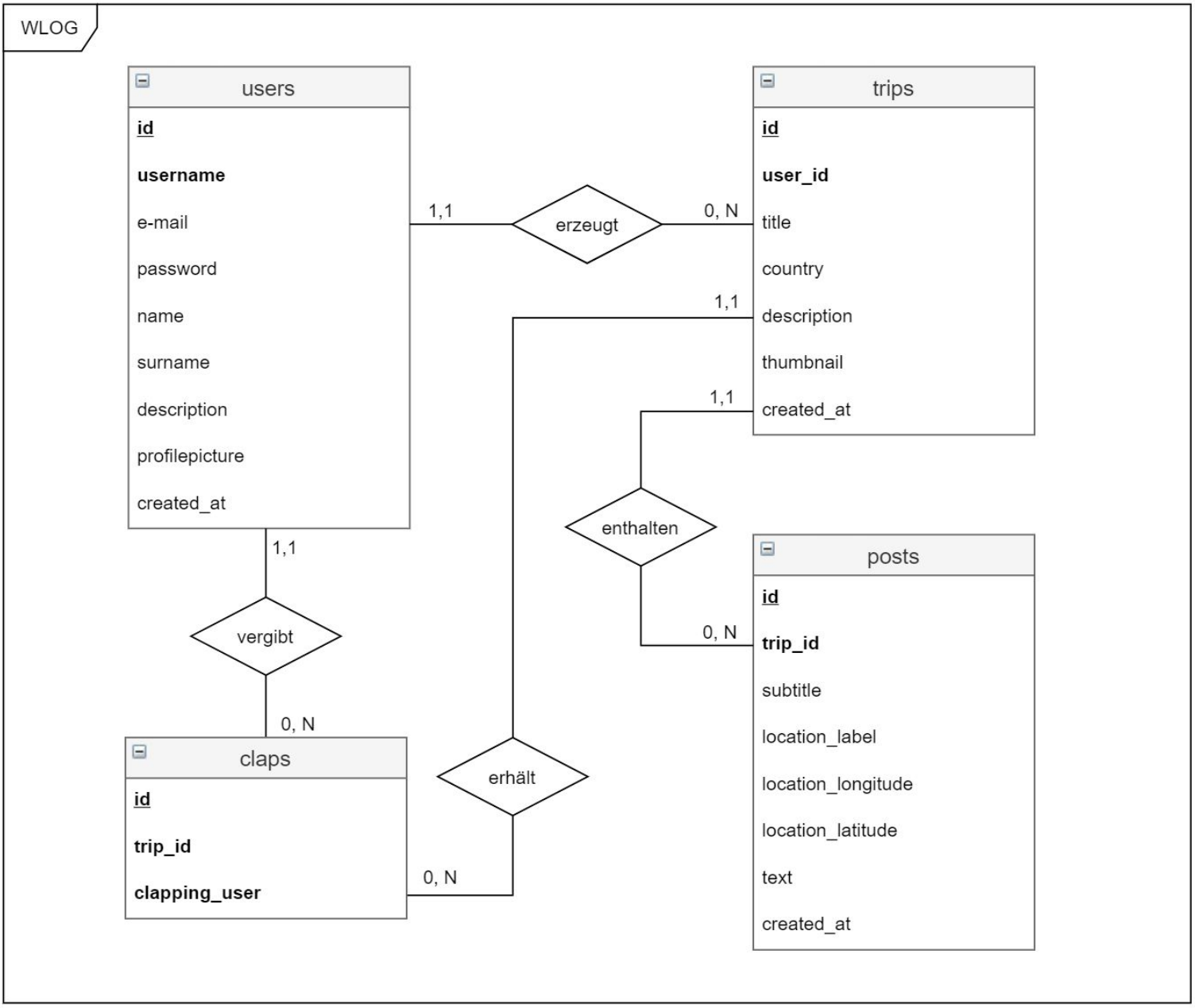
Tabelle: users
Benutzer können sich registrieren, authentifizieren und zudem ihr eigenes Profil verwalten und gestalten. Es wird unter anderem ein Username benötigt, welcher einzigartig sein muss und auf welchem ein Index erstellt wurde, um die häufigen Anfragen auf den Username zu optimieren.

Tabelle: trips
Jeder User hat die Möglichkeit eigene Trips zu erstellen. Ein Trip ist ein Zusammenschluss von verschiedenen Posts. Jeder Trip besitzt unter anderem eine *user_id*, welche ein Foreign Key auf den User ist, der den Trip erstellt hat.

Tabelle: posts
Jeder erstellte Post ist wiederum einem Trip zugeordnet und besitzt deshalb eine *trip_id* mit Foreign Key auf die id des Trips. Ein Post kann einem bestimmten Ort zugewiesen werden und besitzt deshalb auch u. a. eine *location_longitude* und *location_latitude*.

Tabelle: claps
Als Feedbackfunktion ist es möglich einem Trip zu “applaudieren”, also einen clap zu geben. Jeder Benutzer kann einem Trip genau einmal applaudieren. Jeder Eintrag besitzt eine *trip_id* und eine *user_id*, welche in Kombination unique sein müssen.

4.2.1 ER-Diagramm



4.3 REST - API

4.3.1 Struktur

Um eine klare Struktur und Ordnung zu behalten, wurden die Logik, die Models, die Endpunkte und die Controller voneinander getrennt. Die einzelnen Model Klassen, bei der typischerweise eine Instanz eine Tabellenzeile abbildet, befinden sich im *db* Ordner. Die dazugehörigen Controller, welche die Endpunkte verwalten mit denen ein Client kommunizieren kann, befinden sich im Ordner *controller*. In Flask werden diese über sogenannte Blueprints ermöglicht. Hiermit ist es möglich, die Endpunkte in sinnvolle und modulare Untergruppen aufzuteilen.

4.3.2 Models und Datenbankverwaltung

Die Model Klassen sind in diesem Projekt selber implementiert worden. Es gibt eine abstrakte Model Klasse mit den Properties *id* und *created_at*. Zudem gibt es die abstrakten Methoden *insert*, *update* und *delete*, die jedes Model selber implementieren muss. Zum Speichern wurde eine *save* Methode implementiert. Hierdurch wird die Instanz automatisch in die Datenbank gespeichert, indem je nach Bedarf die *insert* oder *update* Methode ausgeführt wird.
Jedes Model erbt von der abstrakten Klasse und kann zudem seine eigenen individuellen Properties und Methoden implementieren.

Connection Pooling
Um mit der MariaDB zu kommunizieren, wird der **mysql.connector** für Python verwendet. Damit nicht für jeden SQL-Befehl eine neue Datenbankverbindung auf- und wieder abgebaut werden muss, verwenden wir einen MySQL Connection Pool. Hiermit ist es möglich die Poolgröße je nach Bedarf anzupassen. Es werden zu Beginn die gewünschten Verbindungen aufgebaut, aus welchen beispielsweise ein Model nach Bedarf eine verwenden kann, welche es nach Abschluss der Transaktion wieder freigibt.

Escapen
Der mysql connector escaped bereits automatisch um unter anderem MySQL Injections und Cross Site Scripting zu verhindern. Somit ist der Developer auf der sicheren Seite und kann dies nicht vergessen.
Damit Blogeinträge sehr individuell erstellt werden können, sendet das Frontend über den TinyMCE einen über HTML Formatierten Post Eintrag. Damit dieser wieder als HTML interpretiert wird, unescaped das Post Model dieses Text Attribut wieder, bevor es an den Client versendet wird. Das öffnet allerdings wieder die Türen für XSS. Dies ist uns bewusst und in der nächsten Version würden wir dieses Konzept abändern.

Passwort Sicherheit
Um die Passwörter der Benutzer sicher zu speichern und zu verwalten, wird das Python Package **passlib** verwendet. Mit diesem wird das User Passwort mit Hinzunahme eines Salts gehashed. Dies wird im *user* Model bei einer Registrierung durchgeführt. Bei einem Authentifikationsversuch wird über **passlib** eine Verifikation durchgeführt.

Instance Cache
Da die Models bei einem Update Command alle Attribute aktualisieren, und dafür die Instanz zuvor aus der Datenbank laden, kann es passieren, dass zwei Requests gleichzeitig den selben Eintrag ändern wollen und dadurch sich gegenseitig überschreiben. Für diesen Fall wurde ein Instanz Cache entwickelt. Dieser speichert die Instanzen, damit beim nächsten Request die Instanz nicht erneut aus der Datenbank geholt wird und zwei verschiedene Instanzen vom eigentlich selben Objekt bestehen, sondern in diesem Fall die Referenz auf das bestehende Objekt zurückgeliefert wird. Somit arbeiten beide mit derselben Instanz. Dieses Verfahren hat in unserem Fall das Problem gelöst, ist aber auf keinen Fall eine optimale Lösung. Hierzu mehr im Kapitel 4.5 Probleme.

4.3.3 Controller

Um sich möglichst an die REST Architektur zu halten, wurden die Controller nach Ihren Endpunkten aufgeteilt. Somit gibt es beispielsweise einen Controller *users.py*, der zum Endpoint users gehört. In diesem ist es unter anderem möglich über '/users/[id]' mit einem GET Request das Profil eines bestimmten Nutzers zu bekommen, oder über einen GET Request auf '/users/[id]/trips' alle zu einem User gehörigen Trips anzufragen. Auch ein PATCH Request auf 'users/[id]' ist möglich, um das Profil des Benutzers zu verändern, und natürlich einige mehr. Eine vollständige Liste befindet sich unter dem Abschnitt 4.3.5 Endpoints.

Sessions
Um Informationen zwischen Requests zu speichern bietet Flask ein *session* Objekt an. Dieses speichert basierend auf Cookies beim Clienten verschlüsselte Daten, die nur von der API gelesen und verändert werden sollen.
Hiermit ist es möglich, den Login des Clienten zu realisieren. Dabei wird in der Session seine *User id* gespeichert. Beim Ausloggen wird die komplette Session wieder bereinigt. Bei jedem Request werden die Cookies mitgesendet, und die API kann überprüfen ob der Client sich bereits eingeloggt hat, und in diesem Fall den Request einem bestimmten Nutzer zuordnen. Dies ist notwendig wenn zum Beispiel der User sein eigenes Profil ändern möchte, oder einen Trip erstellt, welcher ihm zugeordnet werden muss.

Statusmeldungen
Die Kommunikation zwischen Client und der API findet abgesehen von dem Fileupload nur JSON-codiert statt. Im Falle, dass ein nicht JSON-codierter Request kommt, wird ein der HTML-Statuscode 400 - “Bad Request” zurück gesendet. Für den Fall, dass eine Authentifikation vor dem Request stattgefunden haben muss und dies nicht der Fall ist, wird der HTML-Statuscode 401 - “Unauthorized” zurück gesendet.
Wenn die Kommunikation ansonsten valide verläuft, müssen für den Fall angepasste individuelle Statusmeldungen versendet werden. In diesem Fall wird ein Status Code und eine Statusmeldung als Text zurück gesendet: *{"statusCode": int, "status": string}*. Wie üblich wird der statusCode 0 als ‘Success’ verwendet und bei anderen Problemen wird eine andere individuelle positive Zahl verwendet. Die möglichen Ereignisse befinden sich ebenfalls unter dem Abschnitt 4.3.5 Endpoints.

4.3.4 Bilder Management

Es ist möglich ein Thumbnail für seinen Trip hochzuladen, oder Bilder direkt im Post, sowie sein eigenes persönliches Profilbild.
Um diesen Upload zu realisieren muss man an den Endpoint */images* einen ‘POST’ Request mit einer *multiform/data-form* mit dem gewünschten File senden. Zudem muss man für den Request bereits eingeloggt sein. Im Fall eines Profilbildes muss das Bild mit dem Key *‘profileImg’* gesendet werden und im Fall des Thumbnails mit dem Key *‘thumbnail’*. Um die Bilder gut verwalten zu können, gibt es einen helper mit dem Namen ‘imageHandler’. Mit diesem wird nun zuerst überprüft, ob es sich bei dem File um ein valides Bild handelt, indem die Extension auf die erlaubten Bildformate überprüft wird. Ist dies der Fall, wird das Bild in einem *‘tmp’* Ordner unter den *‘assets’* temporär gespeichert und eine dazugehörige ‘uuid’ zurück gegeben. Beim Thumbnail wird diese der Session hinzugefügt. Im nächsten Request des Clients, welcher ein Request zum erstellen eines Trips sein sollte, wird überprüft ob diese uuid vorhanden ist und einem temporären Bild, welches in diesem Fall als Thumbnail dient, zugeordnet wurde. In diesem Fall wird das Bild über den imageHandler gespeichert und der Filename, welcher über einen Timestamp einmalig ist, zurückgegeben, um den Trip erfolgreich in der Datenbank anzulegen.
Im Fall des Profilbildes wird geprüft ob der angemeldete Nutzer tatsächlich noch valide ist und das neue Profilbild ebenfalls mit einem einzigartigen Namen im ‘assets’ Ordner abgespeichert wurde. Erst dann wird der User Eintrag mit dem neuen Profilbild aktualisiert.
Bei den Post Bildern, musste das Verfahren dem TinyMCE angepasst werden. Dieser sendet jedes Bild einzeln und erwartet einen JSON mit einem ‘location’ Attribut, welches die Adresse des gespeicherten Bildes beinhaltet. Der ImageHandler behandelt diesen Fall separat. Hier ist es leider nicht möglich, das Bild temporär zu speichern und zu warten bis die Metadaten kommen. Das Bild wird direkt gespeichert

und der Filename zurück gegeben. Zudem werden die Filenames nicht direkt in der Datenbank abgelegt, sondern nur im vom TinyMCE empfangenen HTML codiert gespeichert. Somit ist es nicht möglich, die Bilder mit dem dazugehörigen Post zu löschen. Dieses Problem ist uns bewusst, und in der nächsten Version würden wir vom TinyMCE auf ein simpleres Konzept wie eine Galerie wechseln.
Zudem muss zusätzlich noch eine Funktion implementiert werden, die in einem gewissen Abstand, über einen Cronjob, die temporären Bilder überprüft und gegebenenfalls nicht referenzierte löscht.

4.3.5 Endpoints

Folgende Endpoints können bei der API angesprochen werden:

"/register" Method = POST
Zum Registrieren werden folgende genannte Attribute erwartet. Der Username muss einzigartig sein, ansonsten wird der Status Code 1 gesendet.

Erwartetes JSON	Statusmeldung
`{ "username": string, "email": string, password": string, "name": string, "surname": string }`	`{"statusCode": int, "status": string, <"error": string> }`

statusCode	Bedeutung
0	successfully registered
1	username not available
2	other error
3	could not register user

"/login" Method = POST
Zum Einloggen wird ein Username und ein Passwort erwartet.

Erwartetes JSON	Statusmeldung
`{ "username": string, "password": string }`	`{"statusCode": int, "status": string, <"user_id": int> }`

statusCode	Bedeutung
0	successfully logged in
1	invalid username or password

"/logout" Method = GET

Statusmeldung

```
{"statusCode": int, "status": string}
```

statusCode	Bedeutung
0	successfully logged out

"/users/{int:id}" Method = GET

Es wird ein JSON mit den nachfolgenden Attributen eines bestehenden Profils zurückgegeben. Falls das Profil nicht existiert wird ein leeres JSON zurückgegeben.

Zurückgegebenes JSON

```
{
   "id": int,
   "username": string,
   "name": string,
   "surname": string,
   "description": string,
   "profilepicture": string
}
```

oder `{}`

"/users/{int:id}/trips" Method = GET

Zurückgegeben werden alle zu einem User gehörenden Trips.

Zurückgegebenes JSON

```
[
 {
   "id": int,
   "user_id": int,
   "author": string,
   "title": string,
   "country": string,
   "description": string,
   "thumbnail": string
 },
 ...
]
```

"/users/{int:id}/posts" Method = GET
Zurückgegeben werden alle zu einem User gehörenden Posts.

Zurückgegebenes JSON

```
[
 {
  "trip_id": int,
  "subtitle": string,
  "location_label": string,
  "location_longitude": double,
  "location_latitude": double,
  "text": string
  },
  ...
]
```

"/users/{int:id}/claps" Method = GET
Zurückgegeben werden alle zu einem User gehörenden Claps.

Zu erwartender JSON

```
{'claps': int}
```

"/trips/{int:id}" Method = GET
Es wird ein JSON mit den nachfolgenden Attributen eines bestehenden Trips zurückgegeben. Falls der Trip nicht existiert wird ein leeres JSON zurückgegeben.

Zurückgegebenes JSON

```
{
   "id": int,
   "user_id": int,
   "author": string,
   "title": string,
   "country": string,
   "description": string,
   "thumbnail": string,
   "user_clapped": bool,
   "posts": [{postData}, ..]
}
```
oder `{}`

"/trips/{int:id}/claps" Method = GET
Zurückgegeben werden alle zu einem Trip gehörenden Claps.

Zu erwartender JSON

```
{'claps': int}
```

"/trips/{int:id}/claps" Method = POST
Um ein “Clap” zu betätigen ist der Login die Voraussetzung.

Statusmeldung

```
{"statusCode": int, "status": string}}
```

statusCode	Bedeutung
0	successfully clapped
1	no valid trip found
2	already clapped

"/trips/{int:id}/claps" Method = DELETE
Um ein “Clap” zurückzunehmen ist der Login die Voraussetzung.

Statusmeldung

```
{"statusCode": int, "status": string}}
```

statusCode	Bedeutung
0	successfully unclapped
1	no valid trip found
2	could not delete clap

"/trips" Method = GET
Sendet die 50 neuesten Trips mit dem Neusten zuerst.

Zurückgegebenes JSON

```
[
 {
   "id": int,
   "user_id": int,
   "author": string,
   "title": string,
   "country": string,
   "description": string,
   "thumbnail": string,
 },
 ...
]
```

"/trips" Method = POST
Login benötigt.
Um einen Trip zu erstellen wird ein thumbnail benötigt. Dieses wird zuvor an den Endpoint **/uploadImg** gesendet. Zum Zuordnen erhält man eine *file_upload_uid* in der Session.

Erwartete JSON	Statusmeldung

`{`
`  "title": string,`
`  "country": string,`
`  "description": string`
`}`
	`{"statusCode": int, "status": string, "trip_id": int}`
	statusCode: 0 – trip successfully created; 1 – invalid request, attributes missing; 2 – thumbnail missing; 3 – could not save trip

"/posts" Method = POST
Login benötigt.
Um einen neuen Post einem Trip hinzuzufügen, wird eine trip_id benötigt sowie der Post Eintrag. Zudem wurden die Bilder über den TinyMCE an den Endpoint **/uploadImg** gesendet.

Erwartete JSON	Statusmeldung

`{`
`  "trip_id": int,`
`  "post": {`
`  "subtitle": string,`
`  "location_label": string,`
`  "location_longitude": double,`
`  "location_latitude": double,`
`  "text": string`
`  }`
`}`
	`{"statusCode": int, "status": string}`
	statusCode: 0 – post successfully created; 1 – invalid request, attributes missing; 2 – no valid trip found; 3 – could not save post

"/images" Method = POST
Login benötigt.

Um ein Bild hochzuladen, muss genau ein Image an eine FormData angehängt werden.

- Für den Fall der Post Images mit dem Namen: 'postImg'
- Zum erstellen eines Trips, hochladen des Thumbnails mit dem Namen: 'thumbnail'
- Um das Profilbild zu aktualisieren mit dem Namen: 'profileImg'

Statusmeldung

```
{"statusCode": int, "status": string}
```

statusCode	Bedeutung
0	file temporarily saved
1	invalid upload
2	file not allowed
3	profileImg successfully saved

"/images/{string:filename}" Method = GET
Um ein gespeichertes Bild abzurufen

"/users/search?pattern=" Method = GET
Der Patternparamter ist in der URL codiert.
Bei Erfolg liefert der Endpunkt eine Liste mit übereinstimmenden Usern zurück:

Zurückgegebenes JSON

```
[
  {
  "id": int,
  "username":string
  },
  ...
]
```

Statusmeldung

```
{"statusCode": int, "status":
string}
```

statusCode	Bedeutung
1	argument 'pattern' missing
2	invalid pattern needs at least 3 characters

4.4 Tests

Damit die einzelnen Models und Controller auf ihre Funktionstüchtigkeit getestet werden können, wurden mit pytest Unit Tests implementiert. Hierfür gibt es einen **tests** Ordner, in dem dieselbe Struktur wie auch in der API angelegt wurde. Für jedes Model oder jeden Controller gibt es eine zugehörige Test Klasse.
Um die API sinnvoll testen zu können wird eine Datenbank benötigt. Da natürlich nicht auf der 'echten' Datenbank getestet werden kann, wird hierfür eine neue Datenbank angelegt, die die identische Struktur besitzt und zudem in bestimmten Fällen bereits mit Testdaten befüllt wurde.
Pytest ermöglicht dies mit sogenannten fixtures. Diese führen Code vor und nach den Tests aus. Der Scope ist wählbar. In unserem Fall hat sich der Scope 'Class' angeboten. Das bedeutet, dass bevor beispielsweise das komplette User Model ausgeführt wird die Datenbank aufgesetzt wird und die einzelnen Test-Funktionen aufeinander aufbauen können. Zum Schluss werden alle Daten gelöscht, damit die nächste Test-Klasse wieder mit einer neu aufgesetzten Datenbank beginnt.
Um die Controller mit ihren Endpunkten testen zu können, bietet Flask einen *test_client*. Mit diesem können Requests versendet und die Response empfangen werden.
Fixtures bieten zudem die Möglichkeit der Testfunktion einen Parameter zu übergeben. Der Test Client von Flask wird den Testfunktionen der Controller Test Klassen mit übergeben, damit diese diesen Nutzen können.
Die Fixtures werden bei pytest in einem sogenannten *conftest.py* File definiert. Für unseren Fall wurden zwei verschiedene definiert. Zum einen ein Fixture, welcher eine leere Datenbank nur mit der Tabellenstruktur bietet. Dieser wird für die Models verwendet. Der andere bietet den Test Client von Flask an und setzt die Datenbank direkt mit bereits vorhandenen Testdaten auf. Dies ermöglicht den Controller Test Klassen eine flexible Testumgebung.

4.5 Probleme

4.5.1 Update Problem

Eines der größten Probleme, sind die selbst implementierten Model Klassen. Diese waren für das Projekt erstmal sehr lehrreich, vor allem auch um sich mit bestimmten Problemen real vertraut zu machen. Allerdings merkte man schnell, dass diese nicht so flexibel sind, wie wenn man ein professionelles ORM-Framework verwenden würde. Hier sind wir vor allem in das Problem hinein gelaufen, dass es nur ein UPDATE Command gibt, mit dem immer alle Attribute aktualisiert werden. Somit musste das Objekt immer zuerst aus der Datenbank geladen werden, dann die gewünschten Attribute angepasst und das komplette Objekt mit allen Attributen, egal welche sich verändert haben, geupdated werden. Das führte zu einem Problem,

wenn parallel verschiedene Änderungen an dem selben Objekt durchgeführt werden sollten. Als Beispiel in unserem Fall war es möglich das eigene Profil zu verändern. Man konnte ein Profilbild hochladen und seine Beschreibung verändern. Es wurde ein Request an '/images' und einer an 'profile/[id]' versendet. Beide haben sich dieselbe User Instanz aus der Datenbank geladen, und unterschiedliche Attribute verändert. Beim Update wurde meistens eine der Änderungen durch die andere wieder überschrieben.
Als Lösung entschieden wir uns für einen sogenannten Instance Cache. Dieser speichert die Objekte global sobald diese das erste mal aus der Datenbank geladen wurden. Beim nächsten get() auf ein Objekt erhält man dann die Referenz auf dasselbe Objekt. Dadurch haben wir das Problem erstmal 'gelöst'. Dies ist dennoch keine optimale Lösung, da die Applikation nicht skalierbar ist. Sobald die Applikation mit mehreren Kernen arbeitet geraten wir wieder in dasselbe Problem hinein.
Der nächste Schritt an dem Projekt ist das umschreiben der API, damit ein ORM Framework verwendet wird. Hierfür bietet sich SQLAlchemy an. Hiermit ist es möglich auch wirklich nur einzelne Attribute eines Objektes zu manipulieren und zudem würde es vor allem auch allgemein mehr Flexibilität bieten.

4.5.2 TinyMCE

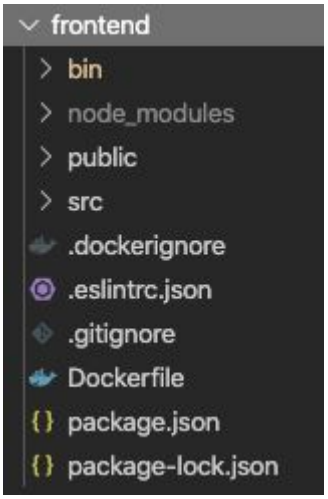
Die Verwendung des TinyMCE Editors im Frontend bietet für den Nutzer einen sehr angenehmen und flexiblen Editor mit vielen Möglichkeiten. Es ist möglich den Text nach belieben zu formatieren und auch Bilder direkt mit einzubinden. Der Output ist allerdings in HTML codiert. Das bietet wie bereits erwähnt ein großes Einfallstor für Cross Site Scripting, da das HTML direkt an das Frontend gesendet und eingebunden wird. Ein weiteres Problem, vor allem für das Backend, ist dass die Bilder hochgeladen werden und der Filename nur im HTML codiert gespeichert, jedoch nicht separat in der Datenbank gespeichert wird. Hierdurch ist es sehr schwer die Bilder zu verwalten, da diese nicht explizit zugeordnet sind. Wenn der Post Eintrag verändert oder gelöscht wird ist es schwierig diese wieder zu entfernen.
In Zukunft würden wir auf ein simpleres Konzept zurückgreifen, bei dem reiner Text oder höchstens in Markdown codierter Text für einen Post Eintrag geschrieben werden kann und zudem eine Galerie existiert für die Bilder mit Zunahme einer Caption hochgeladen werden können.

5. Frontend

5.1 Allgemeines

Für das Frontend verwenden wir HTML, CSS und die Javascript Library React mit JSX. Außerdem haben wir jQuery installiert, da einige der verwendeten Libraries davon abhängig sind und sich Code an manchen Stellen kompakter schreiben lässt.
Für die Kommunikation mit der API nutzen wir den HTTP-Client axios, um Daten vom Backend zu holen, zu speichern oder zu bearbeiten.
Wir haben uns dazu entschieden das Routing zwischen den Seiten ausschließlich im Frontend umzusetzen. Dazu nutzen wir BrowserRouter aus dem Modul *react-router-dom*, um den Routen einen Pfad und eine Komponente, welche gerendert werden soll, zuzuweisen. Um zwischen den Routen zu wechseln nutzen wir die Komponenten *Redirect* oder *withRouter* aus dem Module *react-router-dom.* Bestimmte Routen sollten dabei nur nach erfolgreicher Authentifizierung erreichbar sein. Dazu haben wir die Komponente *ProtectedRoute.jsx* geschrieben, welche ihre Komponente nur rendert, wenn eine Session in dem Session Storage des Browsers aktiv ist.
Die Session wird beim Login im Backend gesetzt. Ist dies erfolgreich, so wird die User Id im Session Storage des Browsers gespeichert. Dadurch lassen sich Funktionen abhängig vom Login-Zustand dynamisch gestalten. So wird bspw. der Menüpunkt *Register* in der Navigationsleiste zu *Profile,* wenn ein User eingeloggt ist.

Ordnerstruktur



Das Frontend hat einen **bin** Ordner mit Skripten zum validieren und testen der Komponenten. Im **public** Ordner befinden sich die statischen Bilder und das favicon der Anwendung. Außerdem liegt hier die index.html, wo die Anwendung im div mit der id *root* gerendert wird. Hier werden auch Stylesheets, Fonts und Libraries global integriert. Im Ordner **src** befinden sich die React Komponenten und die dazugehörigen Unit Tests im Unterordner tests. Die einzelnen Komponenten haben wir in drei Kategorien untergliedert:
Layout Komponenten, welche übergreifend verwendet werden um UI Elemente zu erzeugen, wie zum Beispiel Navigationsleiste oder Modals. Die Komponenten selber als Bausteine für die jeweiligen Seiten und Funktionalitäten
Page Komponenten, in denen die einzelnen Komponenten zu einer funktionierenden Seite zusammengefügt werden z.B. die Homepage, welche unter anderem eine

Login Komponente beinhaltet.
Im *package.json* stehen u. a. die Dependencies.

5.2 Prototyping

Um uns auf einen einheitlichen Stil zu einigen, haben wir zu Beginn damit angefangen für die wichtigsten Seiten Home, Login, Profil und Create (Blog Post) Prototypen zu erstellen.
Mit Hilfe von groben Skizzen konnten wir das Layout definieren und abwägen, welche Elemente wichtiger sind und wie diese positioniert werden sollen. Es war aber auch wichtig High Fidelity Prototypen in Form von Mockups zu erstellen, um sich für ein einheitliches Look & Feel der Webseite zu einigen. In einem weiteren Schritt haben wir uns zusammengesetzt und uns gegenseitig Feedback für die Entwürfe gegeben, so konnten wir uns auf ein einheitliches Design einigen und fehlende bzw. überflüssige Elemente anpassen.

5.3 Design

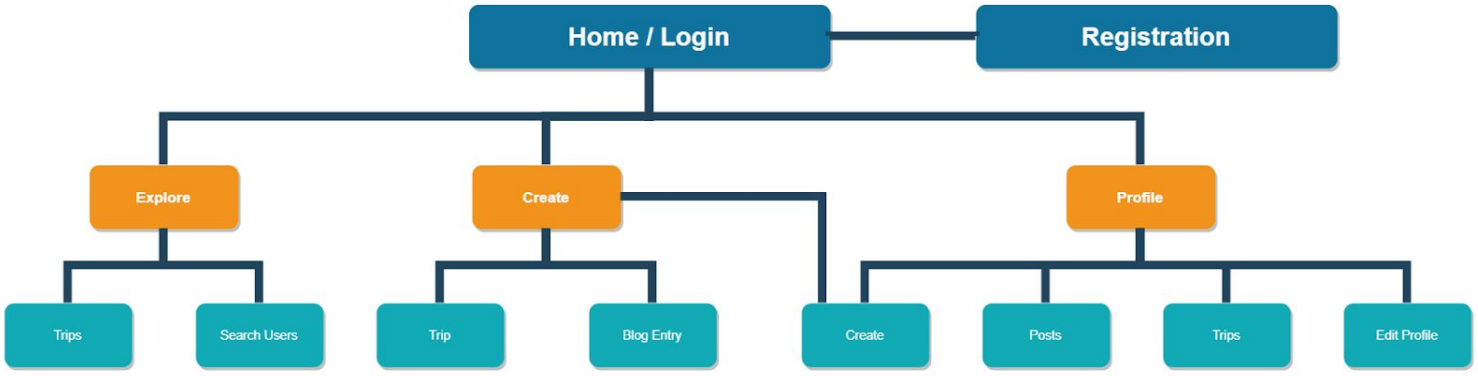
Um ein konsistentes Design zu erreichen, haben wir beim Prototyping einen Styleguide definiert. Dieser beinhaltet ein Farbschema mit vier Farbtönen und ein Schriftsystem mit der Serifenschrift Libre Baskerville, welche über einen Google Fonts Link eingebunden wird. Sie wird vor allem für Überschriften oder Titel verwendet und bietet die Basis für unser Logo. Als Standardschrift für Texte und Formulare wird vom Browser automatisch eine sans-serif Systemschrift gewählt, welche betriebssystemabhängig ist. Das Farbschema beinhaltet eine Grundfarbe für den Hintergrund, eine Primär- / Sekundärfarbe und eine Farbe für wichtige Aktionen wie zum Beispiel das Erstellen eines Posts. Nach diesem Farbschema haben wir auch unsere *React-Bootstrap* Komponenten, wie Tabs oder Buttons angepasst. Unsere Icons haben wir über das *Font Awesome* Kit integriert.
Ebenso haben wir eine Spinner-Komponente mit Font Awesome animiert, um Ladezeiten beim Data Fetching anzudeuten.
Um die vier Haupt-Funktionalitäten unserer Seite zu veranschaulichen haben wir auf der Startseite Illustrationen verwendeten, um die genannten Funktionalitäten metaphorisch zu visualisieren.
Die Illustrationen stammen von der Seite *freepik.com* und wurden für unseren Anwendungsfall modifiziert und koloriert.

5.3.1 Styling

Für das Styling der einzelnen Elemente haben wir uns anfangs auf ein Inline Styling mit JavaScript Objects geeinigt, bis wir wenig später noch das globale Stylesheet

App.css dazu genommen haben, um ein einheitliches und anwendungsübergreifendes Styling zu ermöglichen. Und vor allem, um dynamische Events durch Selektoren wie *:hover* hinzuzufügen, da dies in den Objects nicht möglich war. Außerdem haben wir das CSS-Framework Bootstrap 4 verwendet für die Positionierung in dem 12 Spalten Grid und für die Abstände zwischen den einzelnen Elementen.

5.4 Aufbau



5.4.1 Beschreibung der Components

Die Components werden im Folgenden der Einfachheit halber anhand des visuellen Aufbaus der Webseite beschrieben, also so, wie wenn man die Seiten bzw. die Menüpunkte beim Besuch aufrufen würde. Oberpunkte bilden somit alle sich in **pages** befindenden JavaScript XML - Files, welche in der *App.js* gerendert werden.

Register.jsx - Registrierung */register*

Die Klassenkomponente *Register.jsx* fragt in einem Formular nach allen benötigten Informationen für die Registrierung und Accounterstellung eines Users. Es handelt sich hierbei um Username, E-Mail, Passwort, Bestätigung des Passworts, Name und Nachname. Die Daten werden noch bei Eingabe auf folgende Notwendigkeiten geprüft:

Validierung

Validierung der Register Form:
username: 3-20 Zeichen, [a-zA-Z0-9_-.]
email: Regex: "^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+.[a-zA-Z0-9-.]+$"
password: >6 Zeichen
name: 2-50 Zeichen
surname: 2-50 Zeichen

Bei Verstoß erscheint ein rot gekennzeichneter Hinweis unter dem Input Feld und der Submit kann erst erfolgen, wenn alle Punkte korrekt angegeben wurden. Ein Alert erscheint, wenn nach Absenden der Daten trotzdem etwas schief ging. Beispielsweise wenn der Username schon vorhanden ist, da dies erst mit dem Absenden und Überprüfen der in der Datenbank bereits bestehenden Usernamen

abgeglichen wird. Bei Erfolg bekommt man einen Success-Alert und wird auf die Startseite zum Login geleitet.

Home.jsx - Startseite /

Die *Home.jsx* befindet sich im **src**-Ordner unter **pages** und ruft zum einen die Klasse *Login.jsx*, also auch die Funktionskomponente *InfoContent.jsx* auf.

Login.jsx bietet zwei Inputfelder für das Einloggen eines Users mit Username und Passwort. Falls eines dieser beiden nicht stimmt, wird eine Fehlermeldung geliefert, bei Erfolg wird ein User jedoch weitergeleitet zu seinem Profil. Außerdem befindet sich hier ein Link über den man zur Registrierung geleitet wird, falls noch kein User-Konto vorhanden ist. Wichtig ist, dass *Login.jsx* bei erfolgreichem Einloggen einen State auf “authenticated” setzt, durch welchen ein eingeloggter User auch die in *ProtectedRoute.jsx* definierten Seiten aufrufen kann. Nach dem Einloggen wird der Menüpunkt “”Register” ersetzt durch den Menüpunkt “Profile”.

InfoContent.jsx zeigt eine Anleitung zur Bedienung und den Funktionalitäten von Wlog sowohl visuell als auch schriftlich auf.

ProfilePage.jsx - Profilseite *./users/:id*

Die Profilseite besteht aus einem Profilbild, dem Usernamen und vollen Namen, einer Kurzbeschreibung, sowie einer Übersicht aller erstellter Trips durch die Klassenkomponente *TripGrid.jsx* bzw. *TripImage.jsx* und der Betrachtung aller Posts auf der Karte mit Hilfe von *LeafletMap.jsx*. Zudem finden sich hier Buttons für das Editieren der Profilseite, dem Kreieren eines Trips mit Posts und ein Logout-Button.

Zu den Buttons:

<u>Create:</u>

CreateModal.jsx befindet sich im **layout**-Ordner als Klassenkomponente und sie erscheint beim Aufruf des Buttons “Create” auf der Profilseite als Dialog mit dem User bzw. als Modal/Pop Up Window. Dem User werden von hier aus zwei Weiterführende Möglichkeiten geboten, je nachdem, was er erstellen möchte. Mit der Auswahl “new trip” kann er einen ganz neuen Trip erstellen und einen Blogpost. Mit “blog entry” kann er nur einen Post zu einem bereits vorhandenen Trip kreieren.

Trips und Posts können aber auch ganz allgemein über den Menüpunkt “Create” in der Navigationsleiste erstellt werden, von welchem sich ebenfalls zuerst das genannte Modal öffnet. Das Erstellen von Trips und Posts wird ausführlich in einem anderen Abschnitt erklärt.

<u>Logout:</u>

Der Logout-Button meldet den User ab und er wird auf die Startseite geleitet. Ein Success-Alert bestätigt zudem erfolgreiches Ausloggen.

<u>Edit:</u>

Bei einem Klick auf “Edit” kann der User sein Profil bearbeiten. Er kann sein Profilbild, sowie seine Profilbeschreibung ändern. *EditProfile.jsx* wird im folgenden Abschnitt genauer erläutert.

EditProfile.jsx - Profilbearbeitung */edit*

Über den “Edit” Button auf der Profilseite wird die Klassenkomponente *EditPage.jsx* im **pages**-Ordner aufgerufen. Ein User hat die Möglichkeit seine Profilbeschreibung oder sein Profilbild zu ändern. Mit dem Button “Save”, um seine Änderungen zu speichern, öffnet sich ein Pop Up, genauer Modal, implementiert durch *SaveChangesModal.jsx* und der User wird bei erfolgreichem Speichern zurück auf sein Profil geleitet.

Das Editieren einer Profilseite ist natürlich nur bei eingeloggten Usern möglich, welches durch den Klassenkomponenten *ProtectedRoute.jsx,* gerendert in der *App.js*, gewährleistet wird.

CreatePost.jsx - Erstellen eines Trips mit Post */create*

Wählt man nach dem Klick auf den Navigationsleistenpunkt “Create” bei dem oben beschriebenen Modal das “new trip”- Feld aus, handelt es sich um den Klassen Component *CreatePost.jsx.* Dieser setzt sich folgendermaßen zusammen:

Der Klassenkomponent *TripForm.jsx* bietet in einer FormGroup vier Inputfelder um einen Trip zu erstellen. Ein User kann einen Titel wählen, das Land von dem er erzählen möchte, eine Kurzbeschreibung und ein Thumbnail hochladen. Letzteres wird direkt daneben als Vorschau durch den in *CreatePost.jsx* gerenderten Component *TripImages.jsx* angezeigt. Die Thumbnails werden, wie oben erwähnt, auf der Profilseite für eine Übersicht und Auswahl der Trips gebraucht. Alle Felder müssen hierbei ausgefüllt werden, bei Missachten bekommt das Input-Feld einen roten Border und bei dem Versuch zu speichern taucht ein Error-Alert auf, da keine Daten gesendet werden.

Des Weiteren rendert *CreatePost.jsx* auch den Component *PostForm.jsx*, da jeder Trip auch einen Post benötigt. Zu dem Trip kann man also direkt einen Blogeintrag/Post schreiben. In *PostForm.jsx* sind in einer FormGroup zunächst drei Input-Felder implementiert: Eine Überschrift (Caption) und eine genaue Ortsangabe. Im “Logging”-Feld werden dann das Land aus dem Trip mit dieser automatisch zusammengefügt. Auch diese Felder sind Pflichtangaben und werden beim Leer lassen rot gekennzeichnet und können nicht abgeschickt werden. Außerdem rendert dieser Component den WYSIWYG-Editor TinyMCE, mit welchem ein User seine Reise ausführlich mit Text und Bildern gestalten und beschreiben kann.

Sind alle benötigten Felder ausgefüllt, wird mit dem Button “Save” ein Trip mit Post erstellt. Hierfür muss ein User dies über ein Modal, implementiert durch *SaveChangesModal.jsx*, dies noch einmal bestätigen bzw. abbrechen und wird somit auf sein Profil bzw. zum Erstellen zurückgeleitet.

Auf der Profilseite ist nun der erstellte Trip in der Übersicht, sowie der Post dazu auf der Karte zu sehen und kann angeklickt werden, um in seiner Gesamtbeschreibung betrachtet zu werden. Dazu mehr unter dem Punkt *TripPage.jsx*.

Zu beachten ist, dass das Erstellen eines Trips mit einem Post nur nach einem Login

möglich ist, welches durch den Klassenkomponenten *ProtectedRoute.jsx,* gerendert in der *App.js*, gewährleistet wird.

AddPost.jsx - Erstellen eines Posts */add*
Wählt man nach dem Klick auf den Navigationsleistenpunkt “Create” bei dem oben beschriebenen Modal das “blog entry”- Feld aus, handelt es sich um den Klassen Component *AddPost.jsx.* Hierbei wird nur der wie oben beschriebene Component *PostForm.jsx* gerendert. Zu erwähnen ist, dass weil nur ein Post an sich erstellt wird und nicht ein Trip mit Post, der User auswählen muss, zu welchem Trip der Post nun gehört.
Auch hier hat der User beim “Save” die Wahl den Post nun zu speichern, oder doch noch einmal Änderungen durch den Component *SaveChangesModal.jsx* vorzunehmen.
Außerdem gilt hierbei wie beim **/create**, dass ein User eingeloggt sein muss, um einen Post für einen Trip zu erstellen, gewährleistet durch den Komponenten *ProtectedRoute.jsx*.

TripPage.jsx - Blogeinträge anzeigen */trips/:id*
Der Component *TripPage.jsx* ermöglicht das Anzeigen von einem Trip mit seinen Posts, sowie der Karte durch *LeafletMap.jsx*. Zudem befindet sich hier eine Art Like-Funktion, bei der andere User die Trips mit einem “Clap” bewerten können.
Ein Trip wird geöffnet durch das Klicken auf sein Thumbnail auf der Profilseite (*ProfilePage.jsx)* oder bei der Explore Seite (*Explore.jsx)*. Direkt daneben wird die Karte mit allen Posts durch einzelne Icon-Marker angezeigt. Klickt man auf ein Icon öffnet sich die Beschreibung des jeweiligen Posts weiter unten.

Explore.jsx - Usersuche und Newsfeed */explore*
Der Punkt “Explore” in der Menübar bietet zunächst eine Nutzersuche, durch welche man das Profil des gesuchten Users nach Eingabe in ein Suchfeld durch der gerenderten *SearchBar.jsx* aufrufen kann.
Außerdem werden hier in *Explore.jsx* auch die zuletzt erstellten Trips aller User mit dem Component *SlickGrid.jsx* in einem Slider und mit Hilfe von *TripImage.jsx* mit ihren Thumbnails abgebildet und kann einen Trip mit Klick hierauf aufrufen.
Des Weiteren werden die Trips auch auf einer “Community Map” über *LeafletMap.jsx* angezeigt. Bei einem Klick auf das Map-Icon eines Posts gelangt man über sein Pop Up wieder auf diesen. *Spinner.jsx* dient hierbei als “Loading-Indikator”, damit ein User sich zurecht findet.

Jeder dieser acht Klassenkomponenten aus dem **pages**-Ordner wird in der *App.js* je nach Userauswahl (Switch), jedoch immer mit dem Component *Header.jsx*, also der Navigationsleiste, gerendert.

5.5 Libraries & Modules

In diesem Abschnitt geht es darum, welche Libraries wir für unser Projekt genutzt haben und auf welche Weise.

TinyMCE
Für das Erstellen von Blogeinträgen nutzen wir den WYSIWYG-Editor TinyMCE, der es den Nutzern ermöglicht ohne jegliche HTML-Kenntnisse Texte frei zu erstellen und zu formatieren. Man kann Tabellen, Bilder und Links einbinden und anpassen.
Der Editor selber ist frei modifizierbar und es lassen sich Plugins in der Konfiguration angeben, die dann genutzt werden können.
Es musste ein Account eingerichtet werden, bei welchem man einen API-Key erhält, um den Editor nutzen zu können. Der Inhalt wird als HTML-Code an das Backend gesendet und dort gespeichert, was eine mögliche Sicherheitslücke darstellt (siehe 4.5.2 TinyMCE).

Leaftlet-Geosearch
Jedem Blogeintrag wird ein genauer Standort mit exakten Koordinaten zugeordnet. Dazu nutzen wir das Module *leaftlet-geosearch* mit dem *OpenStreetMap* Provider, welcher eine API bietet, die für Queries einen Array mit passenden Resultaten zurückgibt. Die Resultate sind JS Objects und beinhalten ein Label, eine x-Koordinate (Länge) und eine y-Koordinate (Breite). So kann der Nutzer in der Input Form einen Standort oder eine Adresse angeben und sieht, sobald er die Input Form wieder verlässt in einem Readonly-Feld die geloggte Adresse. Der geloggte Standort ist dabei immer das erste Object im Array, da dieses am genauesten auf die Query zutrifft.

React-Leaflet
Um die einzelnen Posts als Marker auf einer Karte anzuzeigen, nutzen wir *React-Leaftlet* mit Tiles von *OpenStreetMap.* Die Marker werden über eine Polyline miteinander verbunden, um den Reiseverlauf anzudeuten. Klickt man auf einen der Marker, so wird dieser als aktiver Marker gesetzt und ein PopUp mit zusätzlichen Informationen erscheint. Der Zoom und der Mittelpunkt richten sich immer nach dem aktuell aktiven Post. Das Icon für die Marker haben wir durch eine Vektor Datei ersetzt, um einen individuellen Stil zu erzeugen.

React-Testing-Library
Für das Testen wird die React-Testing-Library verwendet. Die Tests befinden sich in **frontend/src/tests** und werden im Kapitel 5.6 Tests genauer erläutert.

User-Interface Elemente

Für das User Interface nutzen wir zum einen das Frontend-Framework *React-Bootstrap* beziehungsweise dessen Komponenten. Wir nutzen die Komponenten Alerts, Accordion, Buttons, Forms, Modals und Tabs. Diese haben wir teilweise individuell auf unser Farbschema angepasst wie in 5.3.1 Styling beschrieben.

Um die verschiedenen Trips auf der Explore Page anzuzeigen nutzen wir die *jQuery* Library *React Slick*, die es ermöglicht mehrere Elemente in einem konfigurierbaren horizontalen Karussell abzubilden. Dabei lässt sich mit Hilfe von Slidern durch die Elemente navigieren.

Ein Nutzer sollte sein Profilbild ändern können, wobei dieses in einem kreisrunden Ausschnitt dargestellt wird. Hierbei ist es schwierig den relevanten Ausschnitt aus einem hochgeladenen Bild auszuwählen. Dafür kommt das Module *react-avatar-edit* zum Einsatz, welches es ermöglicht ein Bild hochzuladen und dieses für ein rundes Profilbild zurechtzuschneiden.

5.6 Tests

Für das Testing im Frontend verwenden wir die *React Testing Library* in Kombination mit dem Testframework *Jest*. Wir implementierten u. a. Tests zur Prüfung der erfolgreichen Validierung bei Registrierung eines Benutzers, und um zu prüfen ob Komponenten erfolgreich gerendert wurden.

Das *App.test.js* File testet unsere *App.js*. Hierfür war eine Konfiguration des *setupTests.js* Files nötig, um Fehlermeldungen mit dem Klassenkomponenten *SlickGrid.jsx* zu vermeiden.

In *Register.test.js* wird die Validierung aller Input Felder zunächst auf richtige Eingabe geprüft. Anschließend wird getestet, ob eine falsche Eingabe auch eine Fehlermeldung wirft.

5.7 Probleme

Navigation Bar

Für die Navigation Bar nutzen wir das Module *NavLink* aus dem Module *react-router-dom*, welches es ermöglicht für aktive Navigationspunkte im Header einen *activeClassName* zu verwenden, um den User zu zeigen, wo er sich gerade befindet. Das heißt, sobald der Nutzer auf dem Pfad */explore* ist, wird der Menüpunkt Explore farbig hervorgehoben. Bei dem Menüpunkt Create sieht es etwas komplizierter aus. Hier gibt es zwei mögliche Pfade */add* und */create*, die beide über das *CreateModal.jsx* aufgerufen werden können. Wenn einer der beiden Pfade aktiv ist, ändert sich der Classname von *nav-link* zu *nav-link-active*.

Ist der Nutzer nicht eingeloggt, so erscheint ihm das *CreateModal* als Weiterleitung zum Login oder zur Registrierung. Da sich hierbei nicht die Route ändern soll, sondern lediglich das Modal gerendert wird, wird anstatt von *NavLink* ein simpler *div* container mit onClick-Event verwendet. Dadurch wird die Accessibility der Seite in der ausschließlichen Nutzung mit der Tastatur eingeschränkt, da der Container aufgrund des fehlenden *href*-Attributes nicht über die Tastatursteuerung (Tabben) zugänglich ist.

File Handling

Eine weitere Hürde war das Senden von Bildern an die API. Wir hatten uns darauf geeinigt, dass Bilder ausschließlich als *File-Interface* in einem *FormData* Objekt geschickt werden dürfen (siehe 4.3.4 Bilder Managment). Da der *TinyMCE* Editor Bilder generell als *Binary Large Object BLOB* speichert, waren wir gezwungen dessen Funktion *images_upload_handler* umzuschreiben. Dabei werden die Bilder einzeln als *FormData* über einen *XML HTTP Request* an den Endpoint */images* gesendet.

Bei dem Module *react-avatar-edit* war es ähnlich. Hier wird das ausgeschnittene Vorschaubild für das Profilbild als Base64 codiert gespeichert. Daher ist es notwendig dieses in der Funktion *dataUriToBlob* als *BLOB* zu konvertieren bevor man das *BLOB* in einem *File* als *FormData* an die API sendet.

Nutzer Feedback

Um den Nutzern Feedback für ausgeführte Aktionen zu geben, nutzen wir Bootstrap Alerts und Validierungen. Bei Fehlermeldungen wird diese in der aktiven Komponente direkt gerendert und bleibt solange bestehen bis die Aktion erneut ausgeführt wird z. B. Submit der Registrierungs Form. Anders ist es bei Erfolgsmeldungen, welche oft erst gerendert werden nachdem auf eine neue Seite weitergeleitet wurde. Daher werden diese nicht im Kontext einer Page Komponente aufgerufen, sondern global in *App.js* und damit unabhängig von der aktuellen Route.

Das ermöglicht es das Feedback nach einer Weiterleitung auf eine andere Route global für drei Sekunden einblenden zu lassen. Die Function *onShowAlert* in *App.js* wird als *props* an die Komponenten der jeweiligen Routen weitergegeben.

Dazu wird anstatt dem Attribut *component* das Attribut *render* verwendet, um die Funktion als *props* zu übergeben.

6. Lessons Learned

- zentralen Projektmanager bestimmen
- Im Team virtuell arbeiten und Ideen diskutieren
- GitLab Issue Boards zur Taskverwaltung
- Versionierungsverwaltung mit Branches für einzelne Tasks und Merge Requests
- Docker-Environment mit Container-Struktur aufsetzen
- korrekte REST-Architektur und Datenaustausch
- Data Model verwenden für Backend
- automatisierte Skripte zur Konfiguration und Überprüfung
- Umgang und Speicherung von Files / Bildern und deren Kodierungen
- User und Sessions Management mit Cookies und Session Storage
- verschiedene Frameworks und Libraries in die Anwendung integrieren
- dynamische Webentwicklung mit React (State Management)
- wiederverwendbaren Code mit Komponenten strukturieren
- automatisiertes Testing / Validierung in einer Pipeline (CI/CD)
- Auf korrekte Installation weiterer Frameworks im Container achten
- Organisation im gesamten Team noch mehr optimieren