Hochschule
für Technik
Stuttgart

STUDIENGANG INFORMATIK
UBIQUITOUS COMPUTING
SOMMERSEMESTER 2020

Geo Timetracking

Timetracking auf Standortbasis

VON

Simon Kellner	72kesi1bif@hft-stuttgart.de
Marcel Schwarz	72scma1bif@hft-stuttgart.de
Tobias Wieck	72wito1bif@hft-stuttgart.de
Tim Zieger	72ziti1bif@hft-stuttgart.de

Betreuender Professor
Prof. Dr. Stefan KNAUTH

24. Juli 2023

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Im Rahmen der Vorlesung Ubiquitous Computing ist eine Projektarbeit mit dem Thema der Allgegenwärtigkeit von Computern vorgesehen.

Dazu fanden wir uns als Gruppe zusammen und sammelten unsere Ideen für Projektthemen. Letztendlich setzte sich der Timetracker mit Geofence gegen eine Wetterstation und eine Augmented Reality App durch.

Die Idee der Timetrack-Anwendung ist, dass ein Arbeitnehmer erst dann mit der Aufzeichnung seiner Arbeitszeit beginnen kann, wenn er sich am Arbeitsort befindet. Wenn er diesen verlässt, wird seine Aufzeichnung beendet. Dabei hat er mehrere Zeitkonten zur Auswahl, auf die er seine Arbeitszeit verbuchen kann.

Das Projekt ist in drei Teile unterteilt: Backend, Web-Frontend und Android-App. Entsprechend dem Aufwand arbeiteten am Web-Frontend zwei Studenten, am Backend und der Android-App jeweils ein Student. Die Android-App soll lediglich die Basisfunktionalitäten bieten, wie Starten und Stoppen der Aufzeichnung für einen gewählten Account, wenn man sich am Arbeitsort befindet, sowie log in und log out. Im Web-Frontend sollen verschiedenste Statistiken angezeigt werden, sowie Adminfunktionaltäten, um Accounts zu editieren, neue Einträge hinzufügen, Geodaten für den Arbeitnehmer setzen und Benutzer löschen. Das Backend kommuniziert mit der Datenbank, sichert die Authentifizierung der Benutzer und stellt den Oberflächen Endpoints zur Verfügung.

2 Projektplanung

2.1 Ziel des Projekts

Es sollte ein Projekt gebaut werden, das es ermöglicht, die Arbeitszeit über eine App zu tracken. Dies sollte aber nur möglich sein, wenn man sich in einem definierten Umkreis zu seinem Arbeitsplatz befindet. Des Weiteren sollte es eine Website zur Verwaltung geben.

2.2 Definition des Workflows

2.2.1 Kommunikation

Zur Kommunikation haben wir für kurze Fragen, sowie das Vereinbaren von Treffen, WhatsApp genutzt. Für Besprechungen haben wir TeamSpeak verwendet.

2.2.2 Sprints

Wir haben die Projektzeit in fünf zweiwöchige Arbeitssprints und einen einwöchigen Vorbereitungssprint aufgeteilt. Die Enddaten der Sprints waren dabei an die Treffen mit Professor Knauth angepasst.

2.2.3 Code-Owners

In unserem Git-Repository haben wir mit Code-Ownership gearbeitet. Dazu haben wir drei Ownerships eingeführt. Marcel Schwarz war Code-Owner für das Backend, Tobias Wieck für die Android App und Simon Kellner, sowie Tim Zieger für das Frontend. Wenn eine Änderung im jeweiligen Gebiet gemacht wurde, musste immer mindestens ein Code-Owner diese genehmigen.

2.3 Sprintziele

2.3.1 Iteration 1

Das Ziel des ersten Sprints war die Erlernung der notwendigen Technologien und die Schnittstellendefinition.

2.3.2 Iteration 2

Im zweiten Sprint sollten die Designgrundlagen und Feature Scopes besprochen werden. Des Weiteren sollte im Backend die Verbindung zwischen dem Backend und der Datenbank hergestellt werden. Im Frontend war geplant, weiter am Grundgerüst der Seite zu arbeiten und bei der App wurde die Einarbeitung weitergeführt.

2.3.3 Iteration 3

Im Frontend sollte im dritten Sprint ein neues Designframework eingeführt werden und die Kommunikation mit dem Backend getestet werden. Der Plan für das Backend war die Erstellung der REST-Controller. Bei der Android App sollte die Login-Funktionalität, sowie das Einlesen der Geo-Informationen realisiert werden.

2.3.4 Iteration 4

Für den vierten Sprint war im Backend geplant, die letzten Controller und Endpoints zu entwickeln. Im Frontend sollten die restlichen Seiten mitsamt Datenabholung aus dem Backend entwickelt werden. Für die App sollte der Geofence entwickelt und die Kommunikation mit dem Backend aufgebaut werden.

2.3.5 Iteration 5

Das Ziel des letzten Sprint war die Fertigstellung des Projekts, die Erstellung der Dokumentation und die Vorbereitung der Präsentation.

3 Entwicklungsumgebung

Da wir uns für eine Full-Stack Application entschieden haben, war es wichtig, gleich zu Beginn die Entwicklungsumgebung so robust wie möglich zu gestalten. Weiter sollte das ganze Setup einfach unter Versionskontrolle gestellt werden können, um überall reproduzierbar zu sein.

3.1 Versionsverwaltung

Für die Versionsverwaltung haben wir das aktuell am weitesten verbreitete Tool Git benutzt. Dies war notwendig, damit wir unabhängig voneinander arbeiten können. Das Repository ist öffentlich auf der Plattform GitLab einsehbar. Die Adresse lautet `https://gitlab.com/marcel.schwarz/2020ss-qbc-geofence-timetracking`.

3.1.1 GitLab

Die Entscheidung für GitLab fiel aber nicht ohne Grund. GitLab bietet auch sehr ausgeprägte Projektplanungsmöglichkeiten, die die Kollaboration sehr vereinfachen. Dazu zählen:

- Issues. In den Issues werden alle Aufgaben für das Projekt abgelegt, die noch erledigt werden müssen oder eine weitere Betrachtung benötigen. Auch Bugfixes werden dort angelegt.
- Issues werden in Merge Requests bearbeitet. Diese Merge Requests können genutzt werden, um über Code zu diskutieren und gegebenenfalls zu verbessern.
- Code-Ownership. Da wir die Teile der Anwendung nach Personen aufgeteilt haben, gibt es für jeden Teil der Anwendung mindestens ein Teammitglied, welches sich besonders gut mit diesen Themen auskennt. Diese Teammitglieder haben dadurch auch die Code-Ownership für diesen Teil des Codes.
- Merge Request Approval. Wenn ein Issue mehrere Teile der Anwendung ändert, muss der jeweilige Codeowner dieses Teils dem Merge Request ebenfalls zustimmen. Ein Beispiel wäre hier die Implementation: „ein Datum im Frontend ändern“, was aber zusätzlich die Anpassung des Datumsformates im Backend erfordert. Bei diesem Merge Request muss dann sowohl ein Codeowner des Frontends als auch des Backends zustimmen. Diese zusätzliche Sicherheitsschicht dient der Stabilität des Master-Branches und der allgemeinen Codequalität.

3.1.2 Umgang mit Issues

Die Issues sind die komplette Dokumentation der erledigten Aufgaben während des Projekts. In ihnen können alle Informationen abgelegt werden, die relevant sind. Beispiele sind hier Bugfixes, Features aber auch Definitionen von Design und Farbschema, die die Zustimmung mehrerer Gruppenmitglieder benötigen. Auch API-Definitionen und Alignment-Meetings gehören bei uns dazu.
Der Lebenszyklus eines Issues sieht wie folgt aus:

1. Issue wird angelegt. Entweder während eines anderen Sprints aus einer Idee heraus, oder bei der Sprintplanung.
2. Zuweisung und Schätzung. Das Issue wird spätestens beim nächsten Planning einem Sprint zugewiesen und erhält Labels die zeigen, welche Teile der Anwendung von diesem Issue betroffen sind. Zuletzt wird dem Issue noch ein Gewicht zugewiesen, welches die geschätzte Dauer in Stunden darstellt.
3. Assignment. Das Issue wird der Person zugewiesen, die es erledigen wird.
4. Merge Request. Es wird ein Merge Request zum Issue angelegt, welcher, wenn er akzeptiert wird, das Issue automatisch schließt.
5. Coding.
6. Review und Ausführung der Continuous Integration Pipeline.
7. Merge des Merge Requests und Schließen des Issues.

3.1.3 GitLab Wiki

Das GitLab Wiki ist eine, auf der Markdown Sprache basierende, Möglichkeit, genauere Dokumentationen zu erstellen. Sollten beispielsweise Entscheidungen getroffen werden, die das ganze Team betreffen, werden diese hier abgelegt.

3.1.4 Continuous Integration

GitLab bietet die Möglichkeit Pipelines für Continuous Integration anzulegen. Diese Pipelines werden dann auf geteilten Maschinen (sog. Runners) der GitLab eigenen Infrastruktur ausgeführt. Die Pipeline baut im Kontext eines Merge Requests die Codeteile, in denen Änderungen erfolgt sind, und validiert damit die Funktionalität auf einem sauberen System. Auch Unit- oder Integrationtests sowie Deployments könnten an dieser Stelle ausgeführt werden.

3.2 Docker

Um die Umgebung unserer Anwendung so homogen wie möglich zu halten, wurde schon von Anfang an auf Docker als Runntime gesetzt. Jeder Teil der Anwendung besitzt eine Dockerfile, die den Code zu einem Executable zusammenbaut. Alle Teile außer Android besitzen zusätzlich noch eine Runtime-Umgebung in Docker. Die Dockerfiles sehen wie folgt aus:

Listing 3.1: Dockerfile Backend

```
FROM gradle:jdk11 AS build
WORKDIR /root
COPY . .
RUN ["gradle", "bootJar"]

FROM openjdk:11-jre-slim
WORKDIR /root
ENV TZ=Europe/Berlin
```

```
COPY --from=build /root/build/libs/*.jar app.jar
EXPOSE 5000
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Listing 3.2: Dockerfile Frontend

```
# build stage
FROM node:14 as build-stage
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# production stage
FROM nginx:stable-alpine as production-stage
COPY --from=build-stage /app/dist /usr/share/nginx/html
COPY ./nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Listing 3.3: Dockerfile Android

```
FROM gradle:jdk8
USER root

ENV SDK_URL="https://dl.google.com/android/repository/commandlinetools-linux-6200805_latest.zip" \
    ANDROID_HOME="/usr/local/android-sdk" \
    ANDROID_VERSION=29 \
    ANDROID_BUILD_TOOLS_VERSION=29.0.3

# Download Android SDK
RUN mkdir "$ANDROID_HOME" .android && cd "$ANDROID_HOME" \
    && curl -o sdk.zip $SDK_URL && unzip sdk.zip && rm sdk.zip \
    && mkdir "$ANDROID_HOME/licenses" || true \
    && echo "24333f8a63b6825ea9c5514f83c2829b004d1fee" > "$ANDROID_HOME/licenses/android-sdk-license"
#    && yes | $ANDROID_HOME/tools/bin/sdkmanager --licenses

# Install Android Build Tool and Libraries
RUN $ANDROID_HOME/tools/bin/sdkmanager --sdk_root=$ANDROID_HOME "tools" > /dev/null
RUN $ANDROID_HOME/tools/bin/sdkmanager --update > /dev/null
RUN $ANDROID_HOME/tools/bin/sdkmanager "build-tools;${ANDROID_BUILD_TOOLS_VERSION}"
    "platforms;android-${ANDROID_VERSION}" "platform-tools" > /dev/null

# Install Build Essentials
RUN apt-get update \
   && apt-get install build-essential -y \
   && apt-get install file -y \
   && apt-get install apt-utils -y

COPY . .

RUN gradle assembleDebug
```

Zu erwähnen sind noch einige Besonderheiten. Da wir ein Programm entwickeln, was sich mit Zeiterfassung beschäftigt, ist es wichtig im Backend die Zeitzone des Images zu setzen, da sonst immer die UTC Zeitzone als Standard angenommen wird. Selbiges gilt auch für die SQL Datenbank, welche mit MariaDB umgesetzt ist.

3.3 Docker-Compose

Um nun die ganze Applikation auszuführen, kommt ein Docker-Compose Setup zum Einsatz.

Listing 3.4: Docker-Compose

```
version: '3.7'

services:
  frontend:
    container_name: qbc_frontend
    build:
      context: ./frontend
    ports:
      - "8080:80"
    depends_on:
      - backend

  backend:
    container_name: qbc_backend
    restart: always
    build:
      context: ./backend
    ports:
      - "5000:5000"
    depends_on:
      - db

  db:
    container_name: qbc_database
    restart: always
    build:
      context: ./sql
    volumes:
      - "qbc-db-data:/var/lib/mysql"
    environment:
      MYSQL_DATABASE: geotime
      MYSQL_ROOT_PASSWORD: supersecure

volumes:
  qbc-db-data:
```

Es werden drei Services definiert: frontend, backend und db. Diese Dienste arbeiten zusammen und warten auf ihre jeweiligen Abhängigkeiten. Zudem wird das Frontend auf dem Port 8080 an die lokale Maschine gebunden und das Backend an Port 5000. Android ist hier nicht vertreten, da

dies eine alleinstehende Anwendung ist. Dieses Setup funktioniert sowohl zu Entwicklungszwecken, als auch in unserem kleinen production deployment. Um den Stack bereitzustellen, muss nur der Konsolenbefehl

```
docker-compose up --build -d
```

im Root-Verzeichnis der Anwendung ausgeführt werden. Es sei noch zu erwähnen, dass beim allerersten Start die Datenbankinitialisierung etwas länger brauchen kann und deshalb das backend mehrere Versuche braucht, bis die Verbindung aufgebaut werden kann.

3.4 Infrastruktur

Die Infrastruktur, auf dem die Anwendung zur Zeit bereitgestellt ist, ist ein kleiner Linux Server bei Strato. Dieser Server hat ebenfalls eine Docker-Compose Installation und läuft auf Ubuntu 20.04 LTS. Gestartet wird es dann exakt gleich, wie im obigen Abschnitt erklärt. Natürlich ist Docker-Compose kein Deployment, welches in Produktion verwendet werden sollte, aber es reicht aktuell für unsere Zwecke aus. Nichts desto trotz ist die Anwendung aber auf eine sehr viel größere Skalierung bestens vorbereitet. Durch die Containerisierung ist unsere Anwendung komplett entkoppelt und könnte somit unabhängig skalieren. Einzig die SQL Datenbank müsste als Container entfernt werden und in ein eigenes Deployment verschoben werden.

4 Backend

Das Backend ist das Herzstück einer jeden Anwendung. Es muss hochverfügbar und enorm fehlertolerant sein. Aus diesem Grund haben wir uns für Technologien entschieden, die Open-Source-Software sind und eine entsprechend große Verbreitung haben. Weiter war es von Anfang an wichtig, trotz der großen Abhängigkeit zum Backend die Entwicklung der anderen Teile nicht zu blockieren. Es wurden daher frühzeitig Modelle und Protokolle erstellt, die bereits vor der Fertigstellung gemockt werden konnten.

4.1 Technologiebeschreibung

4.1.1 Spring Boot

Für die Implementierung des REST-Backends haben wir auf das Spring Framework gesetzt. Genauer gesagt, das Spring *Boot* Framework. Das Wort „Boot“ steht hierbei sinngemäß für „bootstrap“, was uns viel Konfigurationsarbeit abgenommen hat. Alle Standard Beans und Factories waren bereits initialisiert und konnten ohne weitere Konfiguration genutzt werden.
Es wurden im Projektverlauf auch noch diverse Erweiterungen des Frameworks genutzt.

- **org.springframework.boot:spring-boot-starter-web** bringt einen integrierten Tomcat Application Server mit und ermöglicht das Verarbeiten von Webanfragen.
- **org.springframework.boot:spring-boot-starter-actuator** wird genutzt, um Endpoints für Diagnose freizuschalten.
- **org.springframework.boot:spring-boot-starter-data-jpa** bringt alle nötigen Abhängigkeiten, um mit der Java Persistence API Daten in einer Datenbank abzulegen.
- **org.springframework.boot:spring-boot-starter-data-rest** bietet Möglichkeiten, sehr leicht Datenbank Entitäten als HTTP REST Ressourcen bereitzustellen.
- **org.springframework.boot:spring-boot-starter-security** wird später zusammen mit der Authentifizierung über JWT genutzt.
- **org.springframework.boot:spring-boot-starter-test** bringt Möglichkeiten, leichtgewichtig Unit Tests für Webanwendungen zu schreiben.

Zur weiteren Reduktion des „Boilerplate Codes“ wurde zusätzlich noch das Lombok Framework[1] genutzt. Es bietet die Möglichkeit, Getter und Setter sowie diverse Konstruktoren für Datenklassen zu generieren. Dadurch konnten die Datenklassen um etwa 80% in der Große reduziert werden, dies fördert die Lesbarkeit und vermeidet auch Leichtsinnsfehler.

[1] `https://projectlombok.org/`

4.1.2 MariaDB

Als Datenbank wurde MariaDB eingesetzt. MariaDB ist die quelloffene Entwicklung der MySQL Datenbank und nimmt deshalb alle Befehle an, die auch MySQL annimmt. Als Alternative stand noch Postgres zur Auswahl, da wir aber keine der erweiterten Funktionen von Postgres brauchten, fiel die Auswahl auf MariaDB. MariaDB musste auf keinem Entwicklungsrechner installiert werden, da immer das offizielle Dockerimage[2] genutzt wurde.

4.1.3 Weitere Open Source Software

Eine weitere Bibliothek, die für die Authentifizierung benutzt wird, ist die Java-JWT Implementation von Auth0. Sowie die H2 In-Memory Datenbank. Diese zweite Datenbank wird während der Entwicklung genutzt, um schnell homogene Beispieldaten zu laden und Tests auf diesen durchzuführen.

4.1.4 Spezielles Setup

Um produktiv zu arbeiten, mussten noch weitere Tools genutzt werden. Dazu zählt primär die IntelliJ IDEA Ultimate Entwicklungsumgebung[3]. Diese IDE hat sehr viele Integrationen für das Spring Framework, als auch mit unseren Docker-Setup. Es wird dadurch möglich, ausschließlich in der IDE zu arbeiten, ohne weitere Kommandozeilenfenster.
Das zweite wichtige Programm war der REST-Client Insomnia REST[4], welcher alle Möglichkeiten bietet, um REST APIs zu testen und Testabfragen auszuführen.

[2] `https://hub.docker.com/_/mariadb`
[3] `https://www.jetbrains.com/de-de/idea/`
[4] `https://insomnia.rest/`

4.2 Umsetzung

4.2.1 Spring Entities

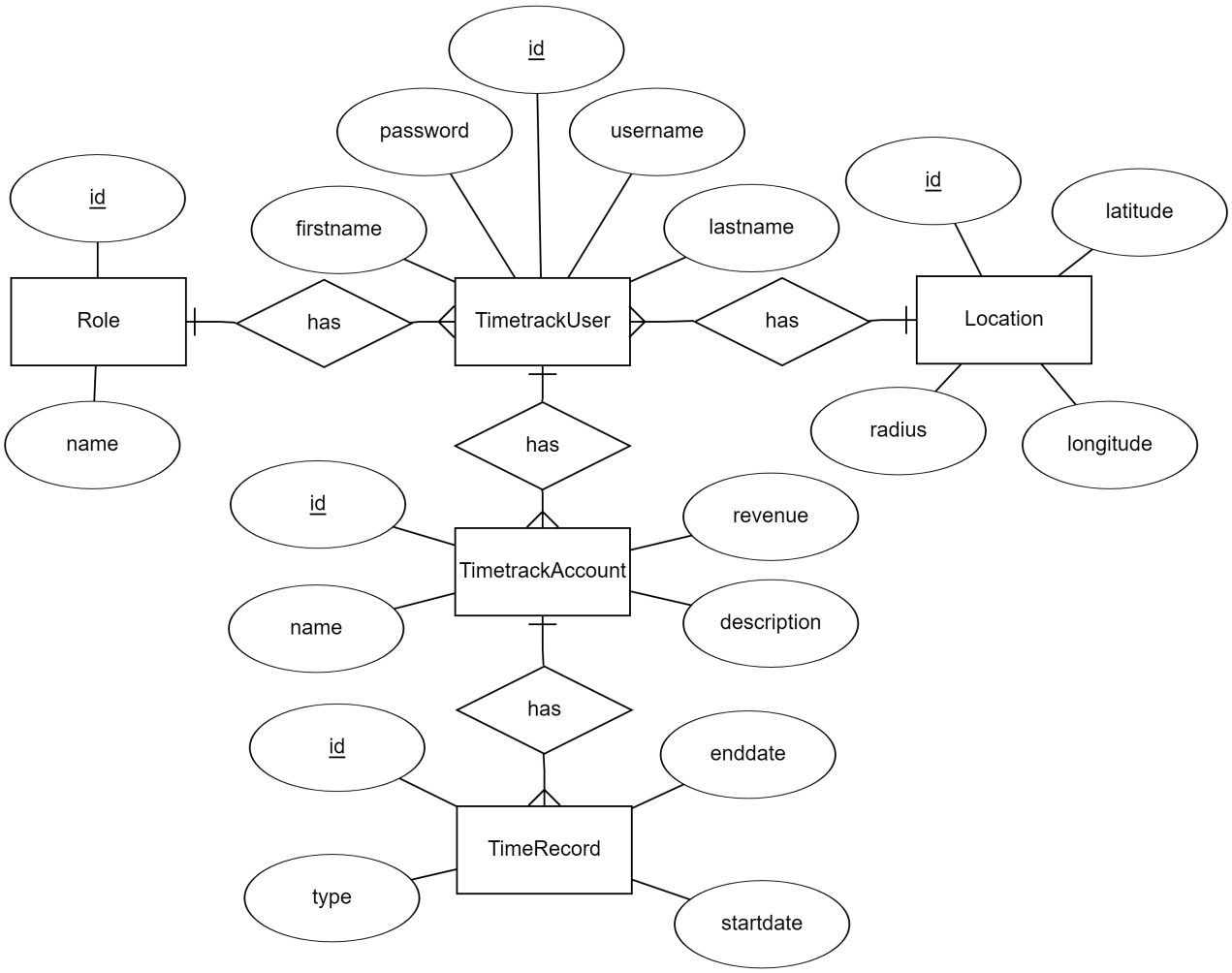


Abbildung 4.1: ER-Modell

Das ER-Modell in Abbildung 4.1 zeigt die komplette Hierarchie, wie sie unserem Konzept entspricht. Wir legen diese Definition aber nicht selbst in SQL an, sondern lassen Java Hibernate dies für uns tun. Die Grundstruktur der gespeicherten Daten ist wie folgt zu verstehen:

- Der **TimetrackUser** ist die Grundstruktur, die alle anderen Daten des Users zusammenhält. Sie speichert allgemeine Nutzerdaten und hält Referenzen auf die **Role** des Nutzers, seine **Location** und alle ihm gehörenden **TimetrackAccounts**.
- Die **Role** sollte ursprünglich erlauben, zwischen einem Admins und einem normalen Nutzers zu unterscheiden, aus Zeitgründen wurde dies aber weggelassen. Die Grundstruktur ist dennoch implementiert, allerdings so, dass jeder Nutzer automatisch Administrator ist.
- Die **Location** Entität speichert den Geofence des Nutzers. Diese Daten werden ausschließlich von der Android App genutzt, um beim Einloggen den Geofence zu setzen.
- Der **TimetrackAccount** ist die zweite große Struktur, die alle **TimeRecords** des Nutzers verwaltet. Jeder Nutzer kann mehrere **TimetrackAccounts** besitzen, aber jeder Account kann nur

einem Nutzer gehören.

- Jede getrackte Zeitspanne wird in einem **TimeRecord** abgespeichert. Dieser Record speichert einen Typ sowie das Start- und Enddatum. Der Typ kann entweder „PAID“ oder „BREAK“ sein. Jeder Record gehört zu genau einem TimetrackAccount.

Die Umsetzung in Java wird nun am Beispiel des TimetrackUsers und des TimetrackAccounts gezeigt.

Listing 4.1: TimetrackUser

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class TimetrackUser {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(unique = true)
    private String username;

    @JsonIgnore
    private String password;

    private String firstname;

    private String lastname;

    @ManyToOne(fetch = FetchType.EAGER)
    private Role role;

    @ManyToOne
    private Location location;

}
```

Die komplette Klasse ist durch die Lombok Integration sehr klein gehalten. Alles weitere wird durch Annotationen geregelt, einige Beispiele sind hier:

@Entity markiert die Klasse als speicherbar in der Datenbank.

@ManyToOne markiert das Attribut als Fremdschlüssenrelation aus einer anderen Tabelle.

@Id zeichnet den Primärschlüssel der Tabelle aus.

@Column setzt spezielle Attribute für die Spalte in der Datenbank.

Die TimetrackAccounts haben zusätzlich noch die Eigenschaft, dass sie sich selbst rekursiv löschen, wenn der zugehörige User gelöscht wird. Selbiges gilt auch für die Records, wenn der zugehörige Account gelöscht wird.

Listing 4.2: TimetrackAccount

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class TimetrackAccount {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private double revenue;
    private String name;
    private String description;

    @ManyToOne
    @OnDelete(action = OnDeleteAction.CASCADE)
    private TimetrackUser user;

}
```

4.2.2 Sicherheit durch JWT

Da wird die Web App im Laufe des Projekts auch öffentlich in Internet stellen mussten, war eine Art Authentifizierung so gut wie unumgänglich. Damit wir keine Probleme mit Session-Affinity haben, entschieden wir uns für eine Token-Based Authentifizierung. Bei der genauen Implementation handelt es sich hier um das JSON Web Token, kurz JWT.

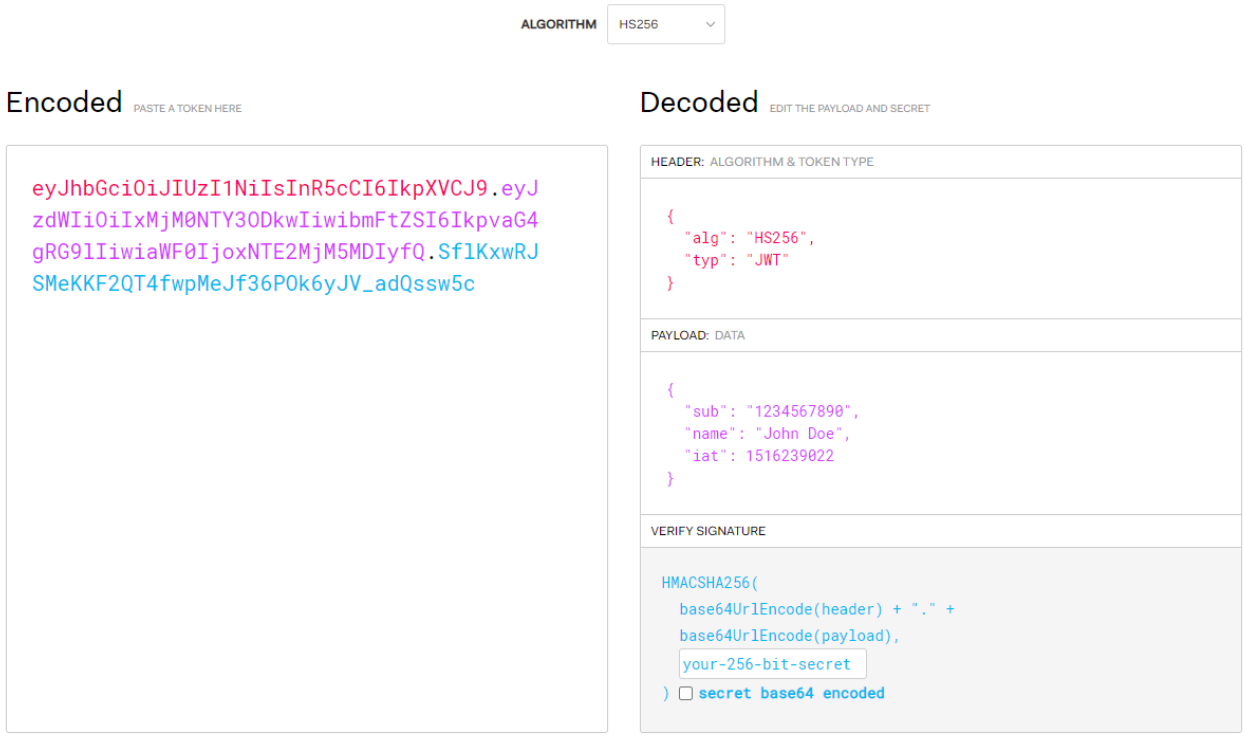


Abbildung 4.2: Aufbau eines JWT

In Abbildung 4.2 ist ein exemplarischer Aufbau eines JWT dargestellt. Das JWT besteht grundsätzlich aus drei Teilbereichen:

1. **Rot hinterlegt:** Bei diesem Teil handelt es sich um den Header, dieser beinhaltet den Typ des Tokens, als auch den Algorithmus, mit dem es verschlüsselt wurde.
2. **Lila hinterlegt:** In diesem Teil werden die eigentlichen Nutzdaten des Tokens abgelegt, dort können z.B. Nutzernamen oder Nutzer-Id sowie eine Rolle hinterlegt werden.
3. **Blau hinterlegt:** Der letzte Part ist dann noch die Signatur des Tokens.

Jeder dieser Teile ist durch einen Punkt im Token abgetrennt. Es ist daher nicht verwunderlich, dass alle Token das selbe Präfix haben werden und nur der Mittelteil, sowie die Signatur sich ändern.
Die Implementation in Spring Boot gelang in drei, vergleichsweise einfachen, Schritten. Zunächst mussten einige Konstanten definiert werden, zur einfacheren Handhabung wurde auch das Secret in den Code platziert. Dieses könnte aber sehr leicht über eine Umgebungsvariable überschrieben werden.

Listing 4.3: JWT Security Constants

```
package de.hft.geotime.security;

public class SecurityConstants {
    public static final String SECRET = "SecretKeyToGenJWTs";
    public static final long EXPIRATION_TIME = 864_000_000; // 10 days
    public static final String TOKEN_PREFIX = "Bearer ";
    public static final String HEADER_STRING = "Authorization";
    public static final String SIGN_UP_URL = "/sign-up";
}
```

Die Lebensdauer eines Tokens wurde mit 10 Tagen ebenfalls sehr hoch gewählt, um die Entwicklung zu vereinfachen. Auch muss dem Token zur erfolgreichen Nutzung in anderen Systemen das Prefix „Bearer “ vorangestellt werden.
Um nun die Tokens in Java zu erzeugen und Abzugleichen, musste die Filterkette von Spring Boot, welche bei jedem Request durchlaufen wird, bearbeitet werden. Jeder Endpunkt außer „/login“ und „/sign-up“ benötigte ab diesem Zeitpunkt eine autorisierte Anfrage.

Listing 4.4: JWT Authentication Filter

```
    @Override
    public Authentication attemptAuthentication(
            HttpServletRequest req,
            HttpServletResponse res) throws AuthenticationException {
        try {
            HashMap creds = new ObjectMapper().readValue(req.getInputStream(),
                HashMap.class);
            return authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(
                            creds.get("username"),
                            creds.get("password"),
                            new ArrayList<>()
                    )
```

```
            );
        } catch (IOException e) {
            logger.info("Unsuccessful login attempt: " + e.getMessage());
            res.setStatus(HttpServletResponse.SC_FORBIDDEN);
            return null;
        }
    }
```

In Listing 4.4 ist der Schritt zu sehen, der die ankommende Anfrage versucht, in eine Loginanfrage zu parsen. Diese Anfrage wird dann in der Filterkette weitergereicht. Bis sie zum UserDetailsService kommt, welcher den User in der Datenbank abfragt und auch das Password abgleicht. Sollte die interne Autorisation erfolgreich sein, wird dieses Objekt mit den Nutzerdaten wieder an die Filterkette zurückgegeben und landet schließlich bei Listing 4.5.

Listing 4.5: JWT Authentication Filter

```
    @Override
    protected void successfulAuthentication(
            HttpServletRequest req,
            HttpServletResponse res,
            FilterChain chain,
            Authentication auth) {
        res.setHeader("Access-Control-Expose-Headers", "Authorization");
        String token = JWT.create()
                .withSubject(((User) auth.getPrincipal()).getUsername())
                .withExpiresAt(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
                .sign(HMAC512(SECRET.getBytes()));
        res.addHeader(HEADER_STRING, TOKEN_PREFIX + token);
    }
```

Der letzte Schritt ist dann nur noch, das Token mit den erhaltenen Daten zu befüllen und dann den „Authorization“ Header der Antwort auf das soeben erstelle Token zu setzen.
Ab jetzt kann sich der Client, der das Token angefragt hat, für die nächsten 10 Tage damit authentifizieren. Dies läuft sehr ähnlich ab, deshalb hier nur sehr kurz dargestellt.

Listing 4.6: JWT Authorization Filter

```
    private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest
        request) {
        String token = request.getHeader(HEADER_STRING);
        if (token != null) {
            // parse the token.
            String user = JWT.require(Algorithm.HMAC512(SECRET.getBytes()))
                    .build()
                    .verify(token.replace(TOKEN_PREFIX, ""))
                    .getSubject();

            if (user != null) {
                return new UsernamePasswordAuthenticationToken(user, null, new
                    ArrayList<>());
            }
```

```
            return null;
        }
        return null;
    }
```

Der eingehende Request geht wieder durch die Filterkette und wenn er an dem Filter in Listing 4.6 ankommt, wird der User extrahiert und später im Security Manager als Autorisation für diesen Request gesetzt. Wichtig ist hier, dass keine weitere Prüfung auf die Existenz des Users durchgeführt wird, auch das Password wird nicht nochmal abgefragt. Der Grund hierfür ist, wenn es den User nicht geben würde, wie käme er dann an das Token?

4.2.3 Repositories

Nachdem der Nutzer authentifiziert ist, bekommt er Zugriff auf alle REST-Repositories. Für jede Ressource, die oben im ER-Modell definiert ist gibt es ein entsprechendes Repository. Dieses wird größtenteils automatisch vom Classpath Scan von Spring automatisch implementiert. Die normalen CRUD Operationen werden für jedes angelegte Repository komplett ohne zutun implementiert. Ein solches Repository ist beispielsweise das der Location.

Listing 4.7: LocationRepository

```
@RepositoryRestResource(
        path = "locations",
        itemResourceRel = "locations",
        collectionResourceRel = "locations"
)
public interface LocationRepository extends PagingAndSortingRepository<Location, Long> {
}
```

Das einzige, was dort getan werden muss, ist die Angabe des Typs, der hier behandelt wird, hier Location, und der Datentyp des Primärschlüssels, hier ein Long. Die Bedeutung der Klasse „PagingAndSortingRepository“ wird in einem späteren Kapitel genauer erläutert. Um den Link der Ressource anzupassen, werden die Parameter in der „RepositoryRestRessource“ Annotation genutzt. Der Pfad geht immer vom Rootpfad der Applikation aus.
Werden nun noch weitere Funktionalitäten in den Repositories benötigt, können diese entweder selbst implementiert werden, oder durch gut ausgewählte Funktionsdefinitionen im Interface der Ressource deklariert werden. Spring kann die Implementierung dann aus dem Namen und den Parametern der Funktion ableiten. Als unser Maximalbeispiel dient hier das RecordRepository.

Listing 4.8: RecordRepository

```
@RepositoryRestResource(
        path = "records",
        itemResourceRel = "records",
        collectionResourceRel = "records"
)
public interface RecordRepository extends PagingAndSortingRepository<TimeRecord, Long> {

    @RestResource(rel = "allBetween", path = "allBetween")
    Page<TimeRecord> findAllByStartdateBetween(
```

```
            @DateTimeFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss") LocalDateTime start,
            @DateTimeFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss") LocalDateTime end,
            Pageable pageable
    );

    @RestResource(rel = "allForUserAndAccount", path = "allForUserAndAccount")
    Page<TimeRecord> findAllByAccount_User_UsernameAndAccount_Name(String username,
        String account, Pageable pageable);

    @Query("SELECT record from TimeRecord record " +
            "where record.account.user.username = :#{principal} " +
            "AND record.enddate > current_date " +
            "AND record.enddate < current_date+1"
    )
    Page<TimeRecord> today(Pageable pageable);

    @RestResource(rel = "openEntries", path = "openEntries")
    Page<TimeRecord> findAllByEnddateIsNull(Pageable pageable);

}
```

In diesem Repository befinden sich diverse verschiedene Methoden, wie Datenoperationen, die definiert werden können, ohne das sie aktiv implementiert werden müssen. Es beginnt mit der Funktion „findAllByStartdateBetween“. Dieser Name kann direkt als Java Hibernate Statement interpretiert werden und nimmt als Parameter zwei Datumsangaben entgegen und eine Page. Die zwei Datumsangaben werden aus dem Schlüsselwort „Between“ abgeleitet. Damit es sich aber um echt vergleichbare Daten handelt, müssen diese nach einem bestimmten Schema geparsed werden. Dieses Schema ist in der „DateTimeFormat“ Annotation angegeben. Als Rückgabe liefert diese Funktion dann eine Menge aller Einträge zwischen diesen Daten.

Die nächste Funktion funktioniert nun ähnlich, nur dass dort über Eigenschaften mehrerer verlinkter Objekte gegangen werden kann. „findAllBy“ ist wieder das selbe wie oben und zeigt an, dass eine Liste von Ergebnissen zurückgeliefert wird, aber „Account_User_Username“ bedeutet nun folgendes: „Gehe zum Account des Records, dann zum User dieses Accounts und von diesem User dann den Username“. Der gefundene Username wird dann mit dem Parameter der Funktion verglichen und die Ergebnisse entsprechend gefiltert. Weiter zeigt das „And“ eine Verkettung eines weiteren Ausdrucks an. So können auch relativ komplexe Abfragen automatisch implementiert werden.

Reicht allerdings die obige Syntax nicht mehr aus, kann auch direkt eine Hibernate Abfrage über die „@Query“ Annotation angegeben werden. Der Name der Funktion ist dann nicht mehr relevant für die Implementation, sondern nur noch für den Pfad, unter dem die Funktion später zu erreichen ist. Die Query an der „today“ Funktion bietet nun die Möglichkeit, alle Einträge in der Records Tabelle für den aktuell anfragenden User zu bestimmen. Zusätzlich wird der Zeitraum noch auf den aktuellen Tag eingeschränkt, daher ergab sich auch der passende Name „today“ für die Funktion. Der Nutzer wird automatisch über die „principal“ Variable in der Abfrage eingefügt. Der Pricipal wird gesetzt, sobald der Authentication Filter den User erfolgreich eingeloggt hat. Weiter wird der aktuelle Tag über die Datenbankvariable „current_date“ abgefragt.

Zuletzt kann auch nach Standard SQL Werten wie „null“ oder „not null“ gefragt werden. Zu sehen ist dies in der zuletzt dargestellten Funktion.

Die Datei ist nicht vollständig abgedruckt, sondern nur ausschnittsweise, um die Grundkonzepte zu erläutern.

4.2.4 Projections

Projections bieten nun noch weitere Möglichkeiten, Daten vor der Rückgabe noch zu transformieren und gegebenenfalls mit Zusatzdaten anzureichern. Eine Projektion ist ebenfalls durch ein Interface definiert und bringt vor allem dann Vorteile, wenn mehrere Ressourcen gebündelt angefragt werden müssen, um beispielsweise eine Übersicht zu erstellen.

Listing 4.9: RecordOverviewProjection

```
@Projection(name = "overview", types = TimeRecord.class)
public interface RecordOverviewProjection {

    LocalDateTime getStartdate();

    LocalDateTime getEnddate();

    long getDuration();

    @Value("#{target.type.name()}")
    String getType();

    @Value("#{target.account.name}")
    String getAccount();

    @Value("#{target.account.user.username}")
    String getUsername();

}
```

Die „RecordOverviewProjection“ (Listing 4.9) reichert eine normale „Record“ Ressource noch zusätzlich mit dem Username und den Accountnamen an. Dadurch muss nicht für jeden Record erneut einzeln der Accountname nachgeschlagen werden. Zudem wird noch ein, bei jeder Anfrage neu berechnetes, zusätzliches Feld angefügt. Nämlich die Dauer des Records in Minuten.
Eine Projektion kann am Beispiel des Records gut verdeutlicht werden.

Listing 4.10: Einzelner Record ohne Projektion

```
{
   "startdate": "2020-05-30T18:00:00",
   "enddate": "2020-05-30T19:00:00",
   "type": "PAID",
   "duration": 60,
   "_links": {
```

```
      "self": {
         "href": "http://localhost:5000/records/27"
      },
      "records": {
         "href": "http://localhost:5000/records/27{?projection}",
         "templated": true
      },
      "account": {
         "href": "http://localhost:5000/records/27/account{?projection}",
         "templated": true
      }
   }
}
```


Listing 4.11: Einzelner Record mit Projektion „overview“


```
{
   "duration": 60,
   "username": "scma",
   "account": "TestAccount",
   "startdate": "2020-05-30T18:00:00",
   "enddate": "2020-05-30T19:00:00",
   "type": "PAID",
   "_links": {
      "self": {
         "href": "http://localhost:5000/records/27"
      },
      "records": {
         "href": "http://localhost:5000/records/27{?projection}",
         "templated": true
      },
      "account": {
         "href": "http://localhost:5000/records/27/account{?projection}",
         "templated": true
      }
   }
}
```


Es ist zu sehen, dass in Listing 4.10 die beiden Felder „account“ und „username“ fehlen, diese tauchen erst bei der Abfrage mit angewendeter, serverseitiger, Projektion auf (siehe Listing 4.11). Die Anfrage für Listing 4.10 lautet `http://localhost/records/27` und um nun die Projektion anzuhängen, wird die URL wie folgt erweitert: `http://localhost/records/27?projection=overview`. Projektionen können hierbei nicht nur auf einzelne Objekte einer Ressource angewendet werden, sondern auch auf eine Menge dieser.

4.3 Endpoints

Die vier Hauptendpoints sind sicherlich die unserer Hautptressourcen: locations, accounts, records und users. Unten gibt es noch den nicht implementierten Endpoint für die Rollen („roles“), dieser liefert aber kaum Informationen. Der „profile“ Endpoint wird erst im nächsten Kapitel erläutert. Um diesen Output zu bekommen, muss der Nutzer authentifiziert sein. Dies geschieht, wie oben schon erwähnt, über den „/login“ Endpoint. Da dieser aber keine Ausgaben außer dem Header mit dem Token liefert, wird er hier nicht weiter erläutert. Selbiges gilt auch für den „/sign-up“ Endpoint. Alle Anfragen, die ab jetzt ausgeführt werden, geschehen immer mit vorheriger Authentifizierung.

Listing 4.12: Zugriff auf die Route „/“ der API


```
{
   "_links": {
      "locations": {
         "href": "http://localhost:5000/locations{?page,size,sort}",
         "templated": true
      },
      "accounts": {
         "href": "http://localhost:5000/accounts{?page,size,sort,projection}",
         "templated": true
      },
      "records": {
         "href": "http://localhost:5000/records{?page,size,sort,projection}",
         "templated": true
      },
      "users": {
         "href": "http://localhost:5000/users{?page,size,sort,projection}",
         "templated": true
      },
      "roles": {
         "href": "http://localhost:5000/roles"
      },
      "profile": {
         "href": "http://localhost:5000/profile"
      }
   }
}
```


Listing 4.13: Zugriff auf die Route „/locations“ der API


```
{
   "_embedded": {
      "locations": [
         {
            "latitude": 1.0,
```

```
            "longitude": 2.0,
            "radius": 3,
            "_links": {
               "self": {
                  "href": "http://plesk.icaotix.de:5000/locations/1"
               },
               "locations": {
                  "href": "http://plesk.icaotix.de:5000/locations/1"
               }
            }
         }
      ]
   },
   "_links": {
      "self": {
         "href": "http://plesk.icaotix.de:5000/locations{?page,size,sort}",
         "templated": true
      },
      "profile": {
         "href": "http://plesk.icaotix.de:5000/profile/locations"
      }
   },
   "page": {
      "size": 20,
      "totalElements": 6,
      "totalPages": 1,
      "number": 0
   }
}
```


Aufgrund der massiven Größe der Ausgaben der API werden die weiteren Endpoints nur noch mit ihrem Link angegeben. Alle Ressourcen unterstützen zudem die CRUD Operationen auf einzelnen Ressourcen, als auch auf der Hauptressource, deshalb werden sie aus Platzgründen ebenfalls übergangen.

Endpoints für Ressourcen

- `/locations{?page,size,sort,projection}`
- `/locations/<nr>{?projection}`
- `/roles{?page,size,sort,projection}`
- `/accounts{?page,size,sort,projection}`
- `/accounts/<nr>{?projection}`
- `/accounts/<nr>/user{?projection}`

- /accounts/search/findByUsernameAndName{?username,account,projection}
- /accounts/search/findByUsername{?username,page,size,sort,projection}
- /users{?page,size,sort,projection}
- /users/<nr>{?projection}
- /users/<nr>/location{?projection}
- /users/<nr>/role{?projection}
- /users/search/byUsername{?username,projection}
- /records{?page,size,sort,projection}
- /records/<nr>{?projection}
- /records/<nr>/account{?projection}
- /records/search/allBetweenAndUser{?start,end,username,page,size,sort,projection}
- /records/search/openEntries{?page,size,sort,projection}
- /records/search/today{?page,size,sort,projection}
- /records/search/allForUser{?username,page,size,sort,projection}
- /records/search/allBetween{?start,end,page,size,sort,projection}
- /records/search/allFrom{?date,page,size,sort,projection}
- /records/search/allForUserAndAccount{?username,account,page,size,sort,projection}

Wenn Ressourcen aktualisiert werden müssen, müssen die Daten immer im JSON Format vorliegen. Die Felder des JSON Objekts müssen mit denen der Ressource übereinstimmen. Es müssen aber nicht alle Felder Werte beinhalten. Soll eine neue Ressource erstellt werden, werden die Daten als POST abgesendet, bei einer Aktualisierung als PATCH. Links zwischen Ressourcen können über den Self Link der Ressource hergestellt werden. Weiter gibt es noch zwei komplett selbst gebaute Endpoints.

Der „/whoami“ Endpoint

Dieser Endpoint dient dazu, um nach dem Login schnell die Startseite der App oder der Webseite mit den Nutzerdaten zu befüllen. Es sind Daten wie der Vor- und Nachname, sowie der Username enthalten. Zusätzlich wird noch die gesetzte Location des Nutzers mitgegeben.

Der „/track“ Endpoint

Beim „/track“ Endpoint handelt es sich um einen der wichtigsten Endpoints. Er erlaubt es, ein Recording zu erstellen, ohne Angabe des Nutzers. Lediglich der Name des Timetrack Accounts, auf den gebucht werden soll, muss angegeben werden. Der Endpoint entscheidet auf Serverseite, von welchem Nutzer die Anfrage ankam. Dazu wird der Nutzer aus dem JWT extrahiert und abhängig davon im Account des Nutzers geschaut, ob schon ein Tracking läuft oder nicht. Sollte noch kein Tracking laufen, wird ein neuer Eintrag mit der aktuellen Zeit erstellt und zurückgeliefert. Das Enddatum ist zu diesem Zeitpunkt noch leer und auch die Duration zeigt „0“ an. Sollte schon ein Tracking laufen, wird dieses mit der aktuellen Zeit als Endzeit beendet und ebenfalls zurückgeliefert. Sollte der Account nicht gefunden werden, oder ein anderer Fehler auftreten, wird ein entsprechender HTTP Statuscode zurückgeliefert.

Listing 4.14: Aufruf von „/track“ ohne laufendes Tracking

```
{
   "duration": 0,
   "username": "scma",
   "account": "Demo",
   "startdate": "2020-06-11T00:59:22",
   "enddate": null,
   "type": "PAID"
}
```

Listing 4.15: Aufruf von „/track“ mit laufendem Tracking

```
{
   "duration": 129,
   "username": "scma",
   "account": "Demo",
   "startdate": "2020-06-10T22:47:55",
   "enddate": "2020-06-11T00:57:41",
   "type": "PAID"
}
```

4.3.1 HAL, Paging und Sorting

Die Hypertext Application Language, kurz HAL, ist eine Spezifikation, mit der APIs automatisch erkundbar gemacht werden können. Sie bietet META-Elemente an, einige davon werden auch bei uns benutzt.

- **„_links“** zeigt weiterführende Links zu Ressourcen oder Informationen zum Paging an.
- **„_embedded“** enthält die Nutzdaten zur entsprechenden Ressource, aber auch weitere Einbettungen zu Sub-Ressourcen.

Zusätzlich dazu nutzt Spring bei der Generierung der Repositories auch Teile der „Hypermedia as the Engine of Application State“, kurz HATEOAS, Spezifikation. Das Listing 4.12 zeigt hierfür den

zusätzlichen Endpoint „profile“. Unter diesem sind viele Spezifikationen zu finden, wie alle anderen Routen auf bestimmte Daten reagieren und auch antworten.
Der „profile“ Endpoint zeigt zusätzlich noch alle Projektionen an, die auf eine bestimmte Ressource angewendet werden können. Der Name der Projektion wird dann durch den URL-Parameter „projection=“ angehängt.
Zuletzt gilt es noch zu erwähnen, dass alle Ressourcen Paging und Sorting unterstützen. Paging ist besonders bei Web APIs wichtig, da die Geschwindigkeit sehr stark von der Menge der übertragenen Daten abhängt. Wenn eine Ressource immer alle Daten zurückliefern würde, würde dies bei mehreren hundert Einträgen sicher noch funktionieren. Aber sobald die Zahl der Einträge deutlich höher wird, muss abgeschnitten und aufgeteilt werden. Unsere Standard Seitengröße ist auf 20 Einträge gesetzt. Weiter enthält die Antwort des Servers durch die HAL Integration immer Links zur aktuellen, nächsten und vorherigen Seite als Link. Sorting wird ebenfalls unterstützt. Es kann nach jedem Feld im zurückgegebenen JSON sortiert werden, auch die Richtung ist spezifizierbar.

4.4 Probleme und Lösungen

4.4.1 Einlesen in Spring

Spring ist ein sehr komplexes Framework, weshalb es manchmal wirklich sehr schwierig war, Fehler zu verstehen, und die Gründe dahinter zu verstehen. Solange man sich aber an viele der Best-Practices von Spring hält, ist es überhaupt nicht schwer, in relativ kurzer Zeit auch sehr komplexe APIs zu bauen. Durch die enorme Menge an Dokumentation und auch Hilfe aus der Community sowie Techtalks können viele Probleme leicht gelöst werden.

4.4.2 Änderungen an den Endpoints

Es mussten anfangs viele Endpoints immer wieder umdefiniert werden, da sie nicht Best-Practices entsprochen haben oder nicht performant funktioniert haben. Dies wurde später aber immer einfacher, wenn man sich an die Denkweise einer REST-API gewöhnt hat. Auch zwei Wege Links zwischen Ressourcen waren bei uns nicht möglich, da sie zu Endlosrekursionen führten. Später wurde aber auch klar, das dies überhaupt nicht gewünscht ist.

4.4.3 Probleme mit MariaDB

Zu Beginn haben wir für das Docker-Image der Datenbank den „latest“-Tag benutzt. Dies war möglich, da wir keinerlei eigene Konfiguration der Datenbank und deren Tabellen vorgenommen haben. Allerdings wurde Mitte April die neue LTS-Version von Ubuntu veröffentlicht und damit auch das Basisimage von MariaDB angepasst. Durch Änderungen in Ubuntu 20.04 funktionierten nun gewisse Datenbankfunktionen nicht mehr ordnungsgemäß. Als Lösung kam dann nur ein Downgrade auf eine ältere Version in Frage.

4.5 Deployment

Das Deployment des Backends findet immer gleichzeitig mit der Datenbank und dem Frontend statt. Die Daten bleiben dabei erhalten und werden, so fern nötig, von Spring automatisch migriert. Auch

beim Hinzufügen oder Entfernen von Feldern aus Entitäten aktualisiert Spring die Datenbanktabellen entsprechend den neuen Feldern. Sollten Felder wegfallen, werden diese einfach gelöscht. Kommen neue hinzu, wird entweder der Defaultwert gesetzt oder, wenn erlaubt, „Null“.

5 Web-Frontend

5.1 Technologiebeschreibung

5.1.1 Vue.js

Vue.js[1] ist ein Javascript Framework, welches den Aufbau von Frontend-Anwendungen erleichtert. Ein Hauptmerkmal hierbei ist die Kapselung der einzelnen Elemente in Komponenten, welche ihren eigenen HTML, Javascript und CSS Code enthalten. Eine Komponente kann mehrere andere Komponenten einbinden, sowie diesen Daten mitgeben. Eingebundene Komponenten können an die übergeordnete Komponente Daten senden.

5.1.2 Vuetify

Vuetify[2] ist ein Designframework für Vue.js, das viele Elemente wie Menüleisten, Buttons und Dialogfenster bereitstellt. Ein bekanntes äquivalentes Framework ist Bootstrap. Das Designschema von Vuetify ist an Googles Material Design angelehnt. Nach Installation können die Elemente sehr einfach eingebunden und verwendet werden.

5.2 Farbschema und Designsprache

Wir haben uns für die, von Google entwickelte, Designsprache „Material Design“ entschieden. Diese zeichnet sich durch ihre kartenartigen Flächen und dem Gestaltungsstil Flat Design aus. Verwendet werden auch viele Schatten, um die materialistisch-physikalische Darstellung zu erzeugen.
In unserem eigens entwickelten Farbschema haben wir uns für ein dunkles Thema mit Blau als Hauptfarbe entschieden.

Abbildung 5.1: Farbschema

Unser Logo wurde im abgestimmten Farbschema umgesetzt und stellt die Kombination zwischen einer Stoppuhr und einem Kompass dar. So verbindet das Logo Zeit und Ort, welche bei der Verwendung unseres Produkts eine wichtige Rolle spielen.

[1] https://vuejs.org/
[2] https://vuetifyjs.com/de-DE/

Abbildung 5.2: Logo unserer Anwendung

5.3 Umsetzung

5.3.1 Einarbeitung

Zur Einarbeitung haben wir den Vue JS Crash Course[3] von Traversy Media genutzt. Dieser ist kostenlos auf YouTube zu finden.

5.3.2 Arbeit mit Dummy-Daten

Zur Erstellung der Listen und Diagramme haben wir häufig Dummy-Daten verwendet, um die Funktionalität im Frontend unabhängig vom Backend zu entwickeln. Die Dummy-Daten haben wir im jeweiligen Vue Component wie folgt angelegt:

Listing 5.1: Dummy-Daten

```
<script>
...
export default {
    ...
  data() {
    return {
      timeRecords: [
        {
          id: 1,
          start: "25.04.2020 / 8:00",
          end: "25.04.2020 / 13:00",
          time: "5:00",
          type: "Paid"
        },
        {
          id: 2,
          start: "25.04.2020 / 13:00",
          end: "25.04.2020 / 14:00",
          time: "1:00",
          type: "Lunch"
```

[3] https://www.youtube.com/watch?v=Wy9q22isx3U

```
      },
      {
        id: 3,
        start: "25.04.2020 / 14:00 ",
        end: "25.04.2020 / 16:30",
        time: "2:30",
        type: "Paid"
      }
    ]
   }
  },
  ...
}
</script>
```

Durch Verwendung der Dummy-Daten war es ebenso möglich, Funktionsaufrufe zum Löschen oder Bearbeiten von Daten zu testen, ohne persistente Veränderungen an den Daten auszulösen. Durch neu laden der Seite sind die Dummy-Daten wiederhergestellt. Bei der Erstellung der Diagramme waren die Dummy-Daten ebenfalls wichtig, so konnten Formatierungsfunktionen für die Zeitanzeige getestet werden. Ebenfalls konnte so die optimale Größe und Anordnung der Diagramme bestimmt werden.

Durch die Verwendung von Dummy-Daten war der Umstieg auf die Livedaten nicht allzu schwer. Die Dummy-Daten konnten bei Anbindung an die Datenbank reibungslos durch Live-Daten aus der Datenbank ersetzt werden.

5.3.3 Authentifizierung

Wie schon im Backend beschrieben wurde, haben wir zur Authenfizierung JSON Web Token benutzt. Beim Login wurde das Token abgeholt und in den Sessionstorage geschrieben. Wir haben uns für den Sessionstorage entschieden, weil dieser beim Schließen des Browsertabs automatisch gelöscht wird. Der Logout Button entfernt ebenso das Token aus dem Storage.

5.3.4 Abrufen der Daten in Listen

Zum Abrufen der Daten nutzen wir „XMLHttpRequests“. Diese geben vom Backend ein JSON Objekt zurück. Dies ermöglicht es uns, die JSON Funktionen von Java Script zu nutzen.

Listing 5.2: Get Request

```
var xhttp = new XMLHttpRequest();
var today;
xhttp.onreadystatechange = function() {
   if (this.readyState == 4 && this.status == 200) {
      today = JSON.parse(xhttp.responseText);
      today = today._embedded.records;
   }
};
xhttp.open("GET", baseUri + "/records/search/today", false);
```

```
xhttp.setRequestHeader("Authorization", sessionStorage.getItem("jwt"));
xhttp.send(null);
```

5.3.5 Ändern und Hinzufügen von Daten

Zum Ändern und Hinzufügen von Daten haben wir ebenfalls „XMLHttpRequests“ genutzt. Zum Hinzufügen wurden Post Requests gesendet, zum Ändern Patch Requests.

Listing 5.3: Post Request

```
xhttp.open("Post", baseUri + path, false);

xhttp.setRequestHeader("Authorization", sessionStorage.getItem("jwt"));

xhttp.send(JSONData);
```

Listing 5.4: Patch Request

```
xhttp.open("PATCH", baseUri + path, false);

xhttp.setRequestHeader("Authorization", sessionStorage.getItem("jwt"));

xhttp.send(JSONData);
```

5.3.6 Auswertung in Diagrammen

Jeder Benutzer kann seine Daten in einer Übersicht zusammengefasst betrachten, hier verwenden wir folgende Diagramme:

- Kreisdiagramme
  - Verhältnis von Arbeitszeit zu Pausenzeit von allen Accounts des Benutzers.
  - Verhältnis der Arbeitszeit je Timetrack Account des Benutzers mit Angabe des Gesamtverdienstes.
  - Verhältnis des Verdienstes je Timetrack Account des Benutzers.
- Säulendiagramme
  - Übersicht über die Letzten 7 Tage mit Arbeits- und Pausenzeit.
  - Übersicht über die Letzten 30 Tage mit Arbeits- und Pausenzeit.

Um Diagramme verwenden zu können, haben wir das Framework Apexcharts[4] eingebunden, welches es ermöglicht, konfigurierbare Diagramme einzufügen. Die Konfiguration des Säulendiagramms für die Ansicht der letzten 30 Tage ist nachfolgend dargestellt.

Listing 5.5: Konfiguration Säulendiagramm

```
<script>
...
```

[4] https://apexcharts.com/

```
export default {
  ...
  data() {
    return {
      series: [
        {
          name: "Working Hours",
          data: []
        },
        {
          name: "Pause Hours",
          data: []
        }
      ],
      chartOptions: {
        chart: {
          type: "bar",
          stacked: true,
          background: "#202020",
          toolbar: {
            ...
          }
        },
        colors: ["#0096ff", "#e21d1f", "#546E7A", "#E91E63", "#FF9800"],
        ...
        plotOptions: {
          bar: {
            horizontal: false,
            columnWidth: "50%"
          }
        },
        xaxis: {
          type: "datetime",
          categories: []
        },
        yaxis: {
          labels: formatter: function(value) {
                    ...
                  return hours + ":" + minutes + ":" + seconds;
                }
        },
        ...
      }
    };
},
```

Für die Kreisdiagramme war es notwendig, alle Zeiteinträge abzuholen, die Zeiten zu addieren und in die Datenfelder des Diagramms zu schreiben. Bei dem Kreisdiagramm, das die Accounts darstellt, war es notwendig, die Timetrack-Accounts des Benutzers abzufragen und für jeden ein Feld der addierten Zeit anzulegen und den Lohn in einer Variablen abzulegen.

Listing 5.6: Zuordnung der Zeit zu den Timetrack Accounts

```
for (let index = 0; index < records.length; index++){
   var record = records[index];
   for (let indexAccs = 0; indexAccs < this.chartOptions.labels.length; indexAccs++) {
      if (record.account == this.chartOptions.labels[indexAccs] && record.type == "PAID") {
         this.series[indexAccs] += record.duration;
      }
   }
}
```

Ebenfalls war es notwendig, eine Funktion zu erstellen, die den gesamten Lohn des jeweiligen Accounts nach den ermittelten Stunden berechnet.

Bei den Säulendiagrammen müssen lediglich die nötigen Zeiteinträge beim zuständigen Endpoint angefragt werden. Dieser Endpoint liefert alle Einträge zwischen einem Startdatum und einem Enddatum. Hier wird immer das aktuelle Datum verwendet und die Zeitspanne entsprechend zurückgerechnet.

Um riesige Anfragen zu verhindern wird Paging verwendet, das heißt, es werden so oft 50 Einträge angefragt, bis die letzte Seite erreicht ist.

5.3.7 Administrator Funktionalitäten

Ein Administrator hat die Möglichkeit zur vollen Nutzerverwaltung. Er kann Nutzer löschen, und bearbeiten. Als Bearbeitungsmöglichkeiten hat er die Accountverwaltung von Nutzern, das Setzen der Arbeitslocation für einen Nutzer und das Ändern des Namens.

5.4 Funktionen der Website

5.4.1 Home

Die Home Seite hat zwei Ansichten. Wenn kein User angemeldet ist, sieht man lediglich einen Wilkommensgruß und hat die Möglichkeit, sich anzumelden. Wenn man angemeldet ist, sieht man seine persönlichen Informationen, die einem zugeordnete Abeitslocation, die Tagesarbeitszeit und die eigenen Accounts.

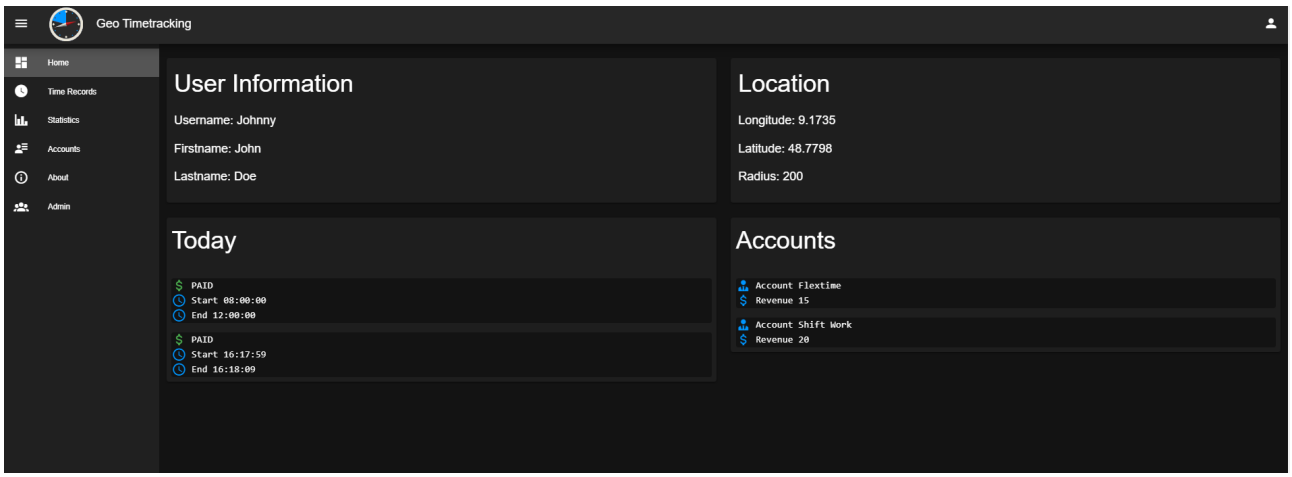


Abbildung 5.3: Home eingeloggt

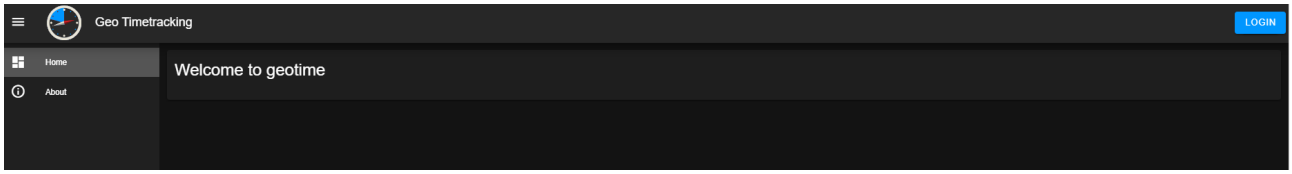


Abbildung 5.4: Home ausgeloggt

5.4.2 Time Records

Auf der Time Records Seite kann man die eigenen Arbeitszeiten einsehen. Außerdem hat man die Möglichkeit, fehlerhafte Einträge zu verbessern oder zu löschen, indem man über den Stift hovert. Neue Einträge können erstellt werden, indem man den „+“-Button am Ende der Seite anklickt.

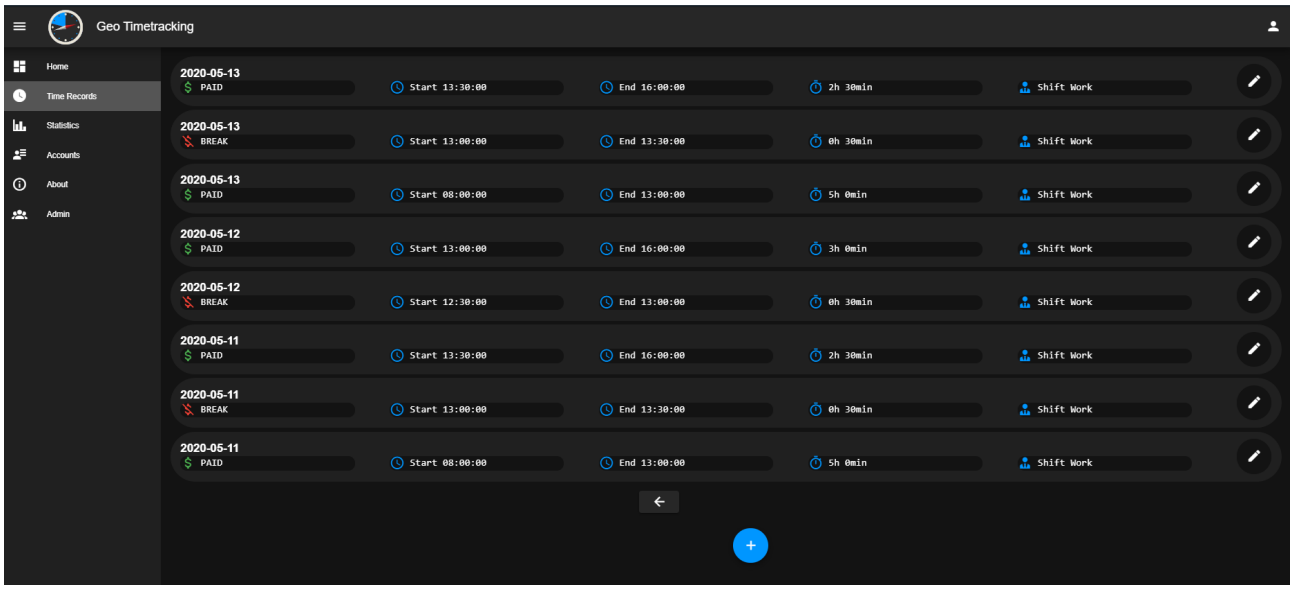


Abbildung 5.5: Time Records

5.4.3 Statistics

Auf der Statistics Seite sind die Daten der Time Records übersichtlich ausgewertet. Hier werden zwei verschiedene Diagrammtypen eingesetzt, um dem Benutzer eine bestmögliche Auswertung seiner

Zeiteinträge zu bieten.

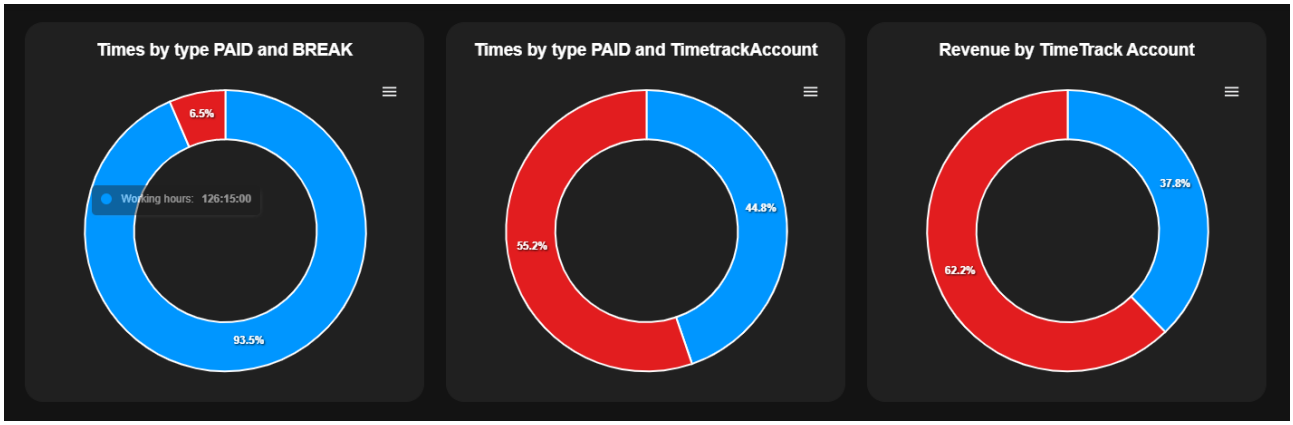


Abbildung 5.6: Kreisdiagramme

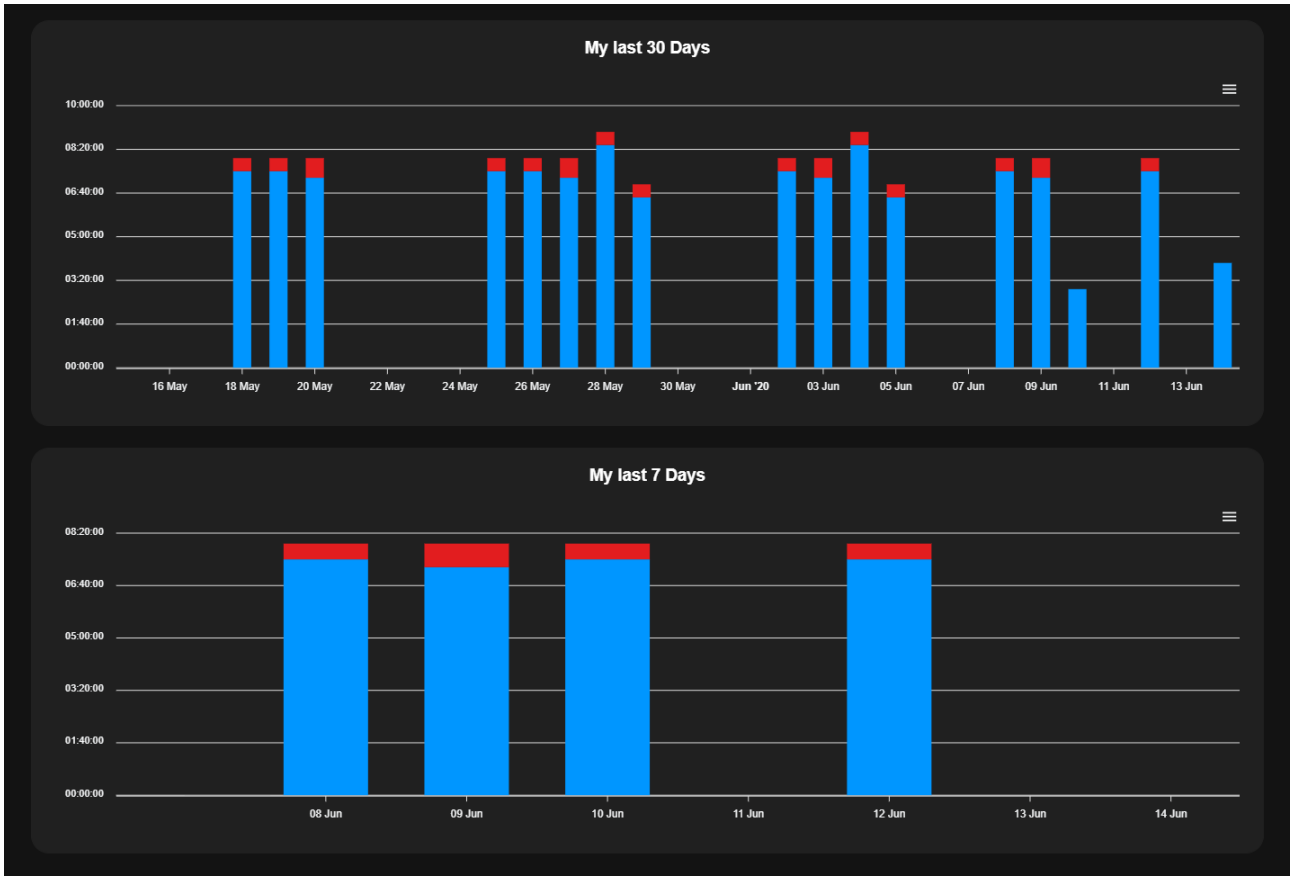


Abbildung 5.7: Säulendiagramme

5.4.4 Accounts

Die Accouts Seite bietet Möglichkeiten, um eigene Accounts einzusehen und zu verwalten. Es ist möglich, neue Accounts hinzuzufügen und bestehende Accounts zu löschen oder anzupassen.

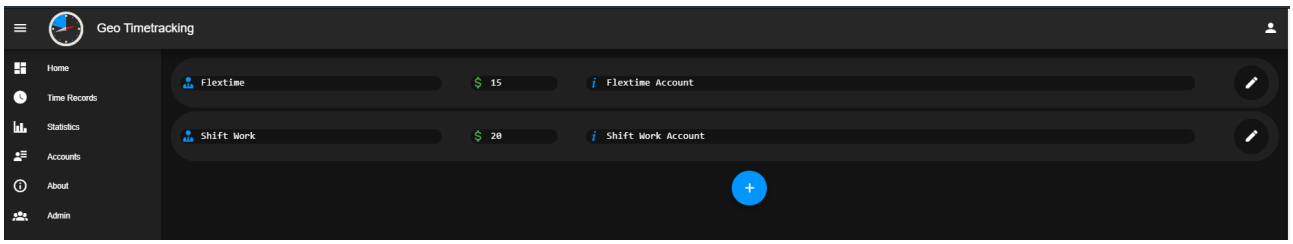


Abbildung 5.8: Accounts

5.4.5 Admin

Die Admin Seite bietet einem die Möglichkeiten, welche in 5.3.7 beschrieben werden. Um die einzelnen Verwaltungsmöglichkeiten zu sehen, reicht es über den Stift zu hovern. Das linke Zeichen (rote Mülltonne) löscht den jeweiligen Nutzer, das mittlere (grünes Papier mit Stift) ist zum Ändern der Nutzerinformationen und der Position der Arbeitsstelle. Das rechte Zeichen (blaue Person mit drei Strichen) führt zur Accountverwaltung für den jeweiligen Nutzer.

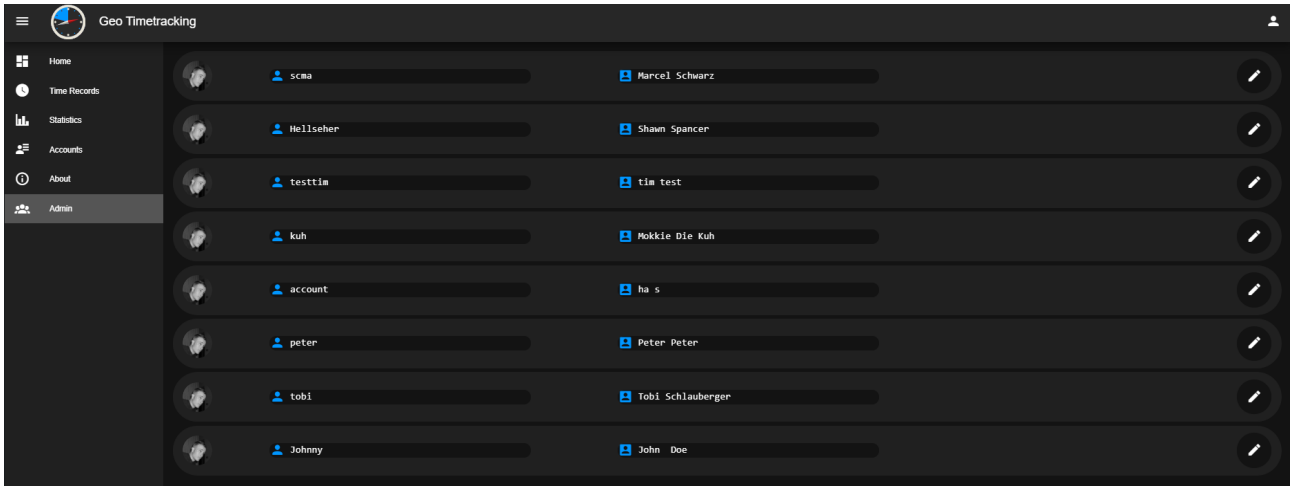


Abbildung 5.9: Admin

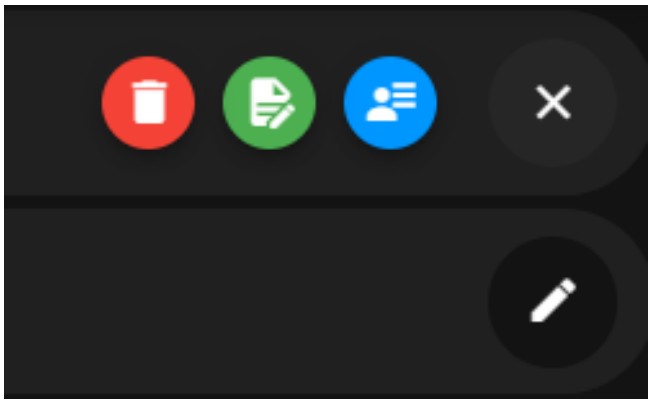


Abbildung 5.10: Nutzerverwaltung

5.5 Probleme und Lösungen

5.5.1 Diagramme

Beim Erstellen der Säulendiagramme sind wir auf den Fehler gestoßen, dass der erste Eintrag von links nicht richtig angezeigt wird. Dieser Fehler ist den Entwicklern von Apexcharts bekannt, aber noch nicht behoben. Wir haben das Problem behoben, indem wir die Daten an der ersten Stelle entfernen. Dies führt zu einem kleinen Abstand, jedoch wird das Diagramm so optimal ohne fehlende Beschriftungen dargestellt.

5.5.2 Custom Headers Chrome

Ein weiteres unserer Probleme war, dass Chrome sich geweigert hat, auf den selbst erstellten Header zuzugreifen. Dieses Problem konnten wir im Backend lösen, indem wir den Header zu den „Access-Control-Expose-Headers“ hinzugefügt haben.

```
res.setHeader("Access-Control-Expose-Headers", "Authorization");
```

5.5.3 Kein Patch möglich

Beim Erstellen eines Patch Requests hatten wir das Problem, dass dieser verweigert wurde. Dies lag daran, dass in den Standard Cors-Konfigurationen nur Get, Head und Post erlaubt sind. Da wir noch Delete und Patch brauchen, haben wir alle Methoden erlaubt.

```
final CorsConfiguration configuration = new
    CorsConfiguration().applyPermitDefaultValues();
configuration.addAllowedMethod("*");
```

6 Android-App

6.1 Technologiebeschreibung

6.1.1 Android SDK

Die Android-Entwicklung wurde, aufgrund der Ausgereiftheit und den Emulatoren, mit Android-Studio realisiert. Android-Studio verwaltet auch das SDK und unterstützt beim aktuell Halten der Bibliotheken.

Das minimale API-Level, welches das Endgerät haben darf, wurde auf 23 „Marshmallow“ festgelegt. Dadurch werden ca. 85% der Geräte unterstützt und ist aktuell genug um gewisse Features, wie das neue Berechtigungssystem, zu unterstützen. Die Zielversion ist das aktuelle Android 10 mit API-Level 29. In dieser Version wurden erneut Berechtigungen geändert, wodurch im Code einige Anpassungen gemacht werden mussten (siehe: 6.5).

6.1.2 Kotlin

Die Entscheidung fiel auf Kotlin als Programmiersprache, da die Sprache von Google für die Entwicklung von Android-Apps bevorzugt wird. Außerdem bietet dies die Gelegenheit, eine neue Programmiersprache zu erlernen. Dadurch musste jedoch viel Zeit investiert werden um zum einen die Sprache und zum anderen die Entwicklungsumgebung, sowie den Aufbau einer Android-App zu lernen. Dafür wurden zwei von insgesamt fünf Sprints eingeplant, weswegen die App nur die Grundfunktionen besitzt.
Die aktuellste Kotlin-Version zur Zeit der Fertigstellung ist 1.3.72.

6.1.3 Retrofit

Für die Kommunikation mit dem Backend wurde die Bibliothek Retrofit in der Version 2.8.1 verwendet. Retrofit ist ein HTTP-Client für Android, mit dem man REST-Endpunkte simpel ansprechen kann. Zusammen mit der Gson-Bibliothek lassen sich JSON-Nachrichten senden und empfangen.
Das angefragte API wird mit Klassen und Methoden in der Anwendung modelliert. Dadurch ist es möglich, nur die Felder abzufragen, welche auch benötigt werden. Genaueres in Kapitel 6.3.3.

6.1.4 Material Design

Material ist eine Bibliothek, die Komponenten und Richtlinien bereitstellt. Nach einmaligem Einbinden der Bibliothek können die Komponenten verwendet werden, indem der Komponente der Style zugewiesen wird.

6.2 Farbschema und Designsprache

In einem gemeinsamen Meeting mit dem Web-Frontend einigten wir uns auf Farbcodes, die auf beiden Oberflächen verwendet werden. So haben wir uns auf ein dunkles Schema festgelegt, mit den Farben aus dem Logo für Schrift und Akzente. Als Schriftart wird Montserrat verwendet (siehe: Abbildungen 6.1 - 6.4).

6.3 Umsetzung

6.3.1 Design der Activities

Insgesamt besitzt die App die vier Activities: Login, MainActivity, Register und Settings. Wobei die Register- und die Settings-Activity aus zeitlichen Gründen ohne Funktion sind. Sie haben auch noch die alten unschönen Eingabefelder, sind aber für die Funktion der gesamten Anwendung nicht sonderlich relevant, weshalb entschieden wurde, diese zu vernachlässigen und den Fokus auf die Funktionalität zu legen.

Jeder Bildschirm hat eine Top-Bar auf der, je nachdem auf welchem Bildschirm man sich befindet, unterschiedliche Inhalte angezeigt werden. Beim Einloggen und Account erstellen wird außer dem Logo und dem Namen der App nichts angezeigt. In den Einstellungen erscheint anstatt des Logos ein Zurück-Button und auf dem Hauptbildschirm gibt es ein Menü zum Ausloggen und um zu den Einstellungen zu gelangen.

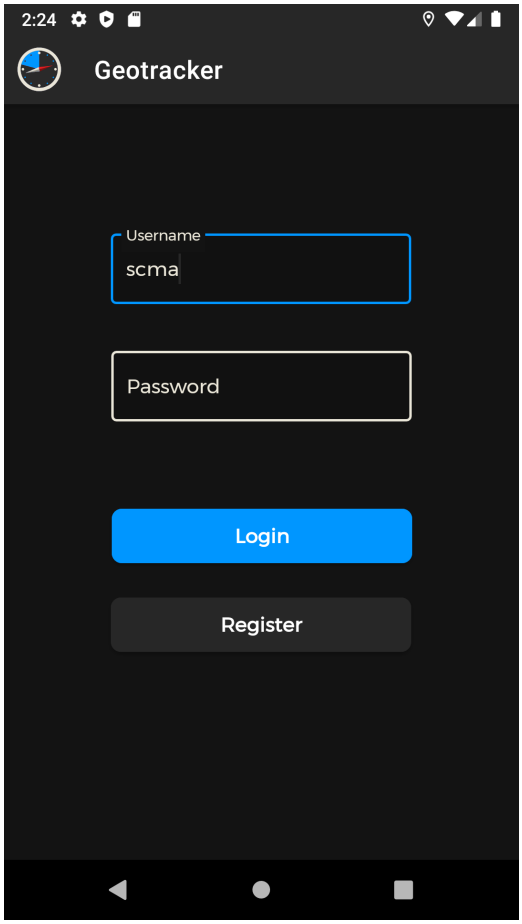


Abbildung 6.1: Login Activity

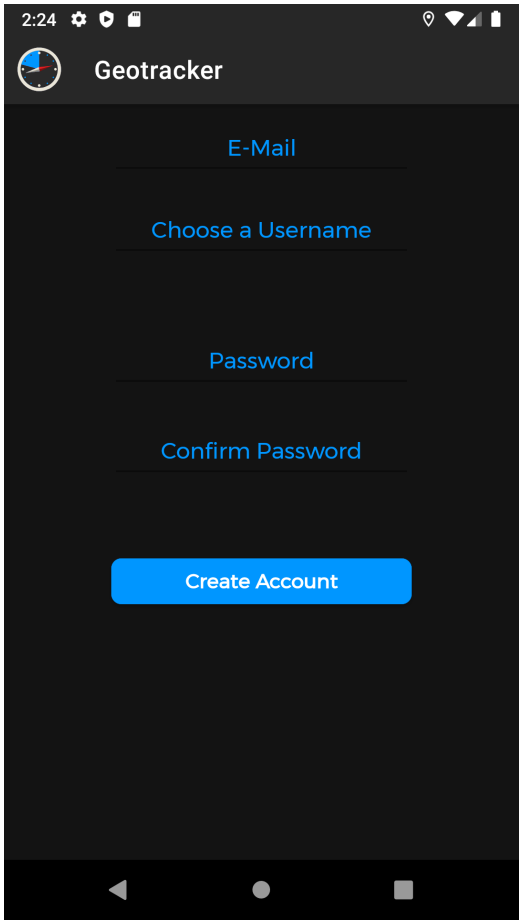


Abbildung 6.2: Register Activity

Links die Eingabefelder mit Material Design und rechts die alten, selbst erstellten.

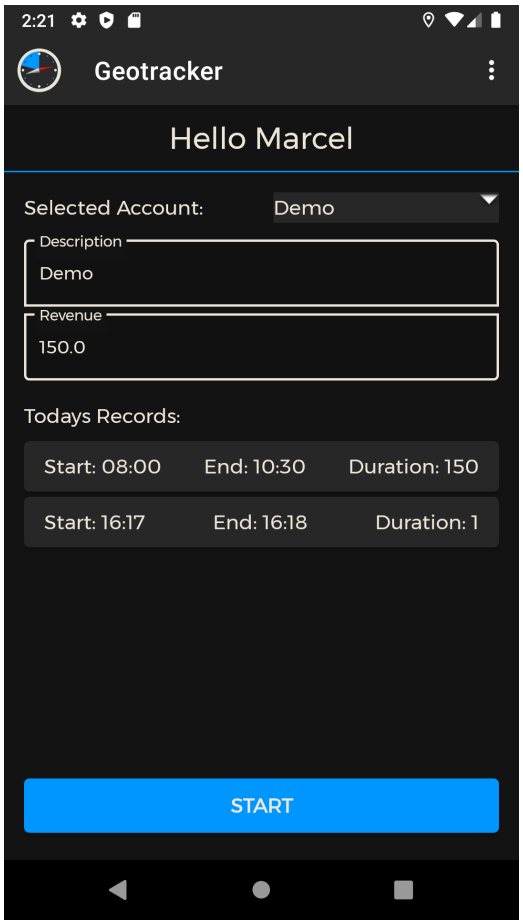


Abbildung 6.3: Main Activity

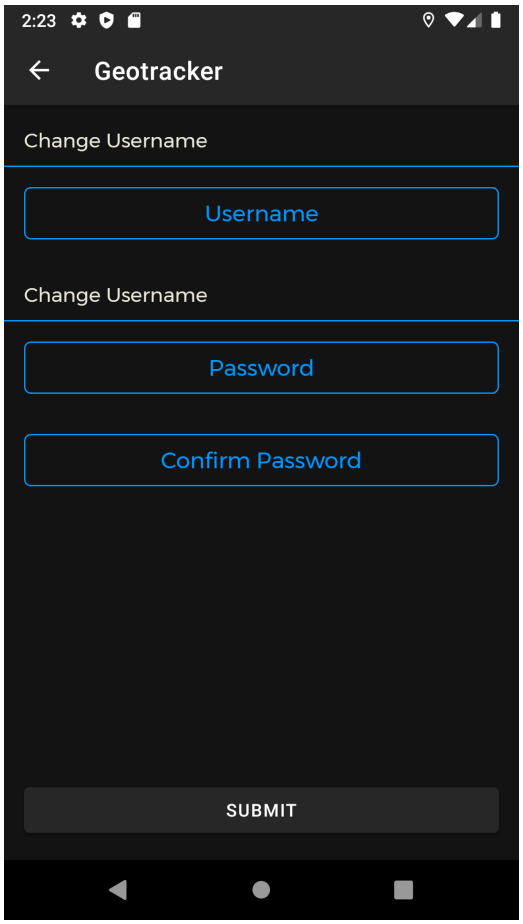


Abbildung 6.4: Settings Activity

Wie zu erkennen ist lag, der Fokus der Implementierung deutlich auf der Main Activtiy, da sie auch das Wichtigste der App beinhaltet. Prominent ist dabei der „START“-Knopf an der Unterseite, mit dem die Aufzeichnung gestartet werden kann (genaueres im Kapitel 6.4.2).

6.3.2 Authentifizierung

Zur Authentifizierung benutzen wir JWT, welches bei jeder Anfrage ans Backend mit geschickt werden muss. Das Token erhält man beim Einloggen mit den richtigen Daten und muss persistiert werden, bis sich der Benutzer ausloggt. Dazu wird das Token im privaten Speicher der App gespeichert. In allen weiteren Activities kann dann auf den Speicher zugegriffen werden und das Token beim Erstellen des „AuthenticationInterceptor“s mitgegeben werden. Beim Ausloggen wird einfach die Datei mit dem Token aus dem Speicher gelöscht.
Der „AuthenticationInterceptor“ ist Kind von der „Interceptor“-Klasse aus der „okhttp3“-Bibliothek, welche in Retrofit eingebunden ist. Mithilfe des Interceptors können REST-Aufrufen Header-Daten mitgegeben werden. In unserem Fall ist das das „Authorization“-Feld mit dem Token.

Listing 6.1: AuthenticationInterceptor

```
class AuthenticationInterceptor(pToken: String) : Interceptor {
   private val token = pToken
   override fun intercept(chain: Interceptor.Chain): Response {
      val original = chain.request()
```

```
        val builder = original.newBuilder()
        .header("Authorization", token)
        val request = builder.build()
        return chain.proceed(request)
    }
}
```

Der Interceptor wird dem HTTP-Client hinzugefügt, welcher später bei der Erzeugung des Retrofit-Builders notwendig ist.

Listing 6.2: HTTP Client

```
val httpClient = OkHttpClient.Builder()
val interceptor = AuthenticationInterceptor(token)
httpClient.addInterceptor(interceptor)
```

6.3.3 Anzeige der Daten in der Main Activity

Die Daten werden per REST-Aufruf mithilfe vom Retrofit-Framework vom Backend geholt. Um Anfragen zu senden, benötigt man einen Retrofit-Builder. Diesem wird die anzufragende URL, ein JSON-Konverter und ein HTTP-Client mitgegeben. Aus diesem Builder und einer Service-Klasse, in der die Methoden definiert sind, wird ein Objekt erzeugt, mit dem die Methoden aufrufbar sind.

Listing 6.3: Retrofit Builder

```
val builder = Retrofit.Builder()
    .baseUrl("http://plesk.icaotix.de:5000")
    .addConverterFactory(GsonConverterFactory.create())
    .client(httpClient.build())
val retrofit = builder.build()
service = retrofit.create(GeofenceService::class.java)
```

Die Klasse „GeofenceService“ dient, wie oben beschrieben, zur Definition der Endpunkte in Form von Methodenaufrufen. Dort wird definiert, ob es ein „POST“- oder „GET“-Entpunkt ist, wie der Pfad lautet und was für Parameter mitgegeben werden.

Listing 6.4: GeofenceService

```
@POST("/login")
fun login(@Body login_data: ValuesUserLogin): Call<Void>

@GET("whoami")
fun getUser(): Call<ValuesUser>

@GET("accounts/search/findByUsername")
fun getAccounts(@Query("username") username : String): Call<EmbeddedAccounts>
```

Der Rückgabewert der Methoden ist immer vom Typ „Call“. Wenn aus dem Body Werte gelesen werden sollen, muss eine Art Skelett-Klasse angelegt werden mit den, für die Anwendung relevanten, Feldern. Die Klasse „ValuesUser“ stellt Werte der Antwort bereit, wie z. B. den Vornamen.

Listing 6.5: ValuesUser

```
class ValuesUser(firstname: String) {
   @SerializedName("firstname")
   var firstname = firstname
}
```

Der Aufruf der Methode erfolgt asynchron. Deshalb darf sich nicht auf das Ergebnis des Aufrufs direkt danach verlassen werden, sonst bekommt man eine Null-Pointer-Excetion. Die Methode „enqueue“ besitzt ein Callback-Objekt als Parameter, welches „onResponse“ und „onFailure“ überschreibt. Dort wird entsprechend definiert, was in den jeweiligen Fällen ausgeführt werden soll.

Listing 6.6: Callback der „getUser“ Funktion

```
val call = service.getUser()
call.enqueue(object : Callback<ValuesUser> {
   override fun onResponse(call: Call<ValuesUser>, response: Response<ValuesUser>) {
      if (response.isSuccessful) {
         val firstname = response.body()?.firstname
         lbl_username.text = "Hello " + firstname
      } else {
         println("Response not successful: ${response.code()}")
      }
   }
   override fun onFailure(call: Call<ValuesUser>, t: Throwable) {
      println("Response 'whoami' failed. " + t.message)
   }
})
```

In dieser Art und Weise werden alle Anfragen ans Backend gehandhabt. Dazu zählen:

- Abfragen der Location-Daten zu dem Benutzer für den Geofence
- Befüllen des Dropdown-Menüs mit den Timetrack-Accounts des Benutzers
- Anzeigen der Beschreibung und der Vergütung
- Befüllen des RecyclerViews mit den heutigen Einträgen
- Auslösen des Start-/Stopp-Events
- Einloggen

6.3.4 Geofencing

Die Geofencing-Funktion ist die zentrale Funktion für die App und auch für das gesamte Projekt. Deshalb war es wichtig, dass sie frühzeitig funktioniert.

Um die Position eines Gerätes zu bestimmen, bedarf es einer Berechtigung, die vom Benutzer bestätigt werden muss. Für Geräte mit API-Level 28 und niedriger muss dafür die

„ACCESS_FINE_LOCATION“-Berechtigung gesetzt werden und für API-Level 29 und höher „ACCESS_BACKGROUND_LOCATION“.

Der Geofence wird initialisiert, wenn für den Benutzer Geo-Daten gespeichert sind. Ist dies der Fall, so wird ein „GeofencingClient“ angelegt, dem dann der Geofence hinzugefügt wird. Der Geofence wird erzeugt mit den Parametern: Breitengrad, Längengrad, Radius, der Lebenszeit des Fence und den Übergangstypen. Die Typen sind in unserem Fall „GEOFENCE_TRANSITION_ENTER“ und „GEOFENCE_TRANSITION_EXIT“, da wir immer reagieren wollen, wenn der Nutzer den Bereich verlässt oder betritt.

Listing 6.7: Anlegen des Geofencing Clients

```
geofencingClient = LocationServices.getGeofencingClient(this)
geofence = Geofence.Builder().setRequestId("Geofence")
   .setCircularRegion(lat, long, rad)
   .setExpirationDuration(Geofence.NEVER_EXPIRE)
   .setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER or
       Geofence.GEOFENCE_TRANSITION_EXIT)
   .build()
```

Um den Geofence-Client zu starten wird auf das Objekt die „addGeofences“-Methode ausgeführt mit einem „GeofencingRequest“-Objekt und einem „PendingIntent“-Objekt als Parameter.

Listing 6.8: „addGeofences“ Methode

```
geofencingClient.addGeofences(getGeofencingRequest(), geofencePendingIntent)?.run {
   addOnSuccessListener { ... }
   addOnFailureListener { ... }
}
```

In der „getGeofencingRequest“-Methode wird festgelegt, auf welches initiale Event reagiert werden soll und der oben erstellte Geofence wird hinzugefügt. Als initiales Event haben wir „INITIAL_TRIGGER_ENTER“ gewählt, da es ausgelöst wird, wenn man sich bereits im Bereich befindet und die App startet. Denn erst mit dem Eintrittsevent wird der Button zum Starten der Aufzeichnung freigeschaltet. Das „geofencePendingIntent“ definiert die BroadcastReceiver-Klasse, welche bei jedem Event aufgerufen wird.

Listing 6.9: Setzen der Geofence Trigger

```
private fun getGeofencingRequest(): GeofencingRequest {
   return GeofencingRequest.Builder().apply {
      setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER)
      addGeofence(geofence)
   }.build()
}
private val geofencePendingIntent: PendingIntent by lazy {
   val intent = Intent(this, GeofenceBroadcastReceiver::class.java)
   PendingIntent.getBroadcast(this, 0, intent, PendingIntent.FLAG_UPDATE_CURRENT)
}
```

Die „GeofenceBroadcastReceiver“-Klasse definiert, was bei den jeweiligen Events ausgeführt wer-

den soll. In unserem Fall ist dies das Verändern einer boolean Shared-Prefrences-Variable, je nachdem ob der Bereich betreten oder verlassen wurde. Warum diese Art und Weise gewählt wurde lesen Sie in Kapitel 6.5. Das Code-Beispiel zeigt die Aktion beim Betreten des Bereichs.

Listing 6.10: Ändern der Shared-Preferences

```
context!!.getSharedPreferences("LOCATION", Context.MODE_PRIVATE)
   ?.edit()
   ?.putBoolean("ENABLED", true)
   ?.apply()
```

In der „MainActivity“ wird ein Listener für diese Shared-Prefrences-Variable definiert. Je nachdem, zu welchem Wert sich die Variable ändert, wird der Start/Stopp-Button freigeschaltet oder gesperrt. Wenn der Benutzer den Bereich verlässt, aber noch aufzeichnet, wird dadurch die Aufzeichnung automatisch gestoppt und gespeichert.

6.4 Funktionen der App

Wie oben beschrieben, besteht die Android-App aus vier Activities. Die Register- und Settings-Activity sind aus zeitlichen Gründen ohne Funktion und layouttechnisch nicht überarbeitet. Der Fokus lag stark auf der Main-Activity, die das Kernstück der App darstellt. Im Folgenden die Funktionalitäten der Activities Login und Main.

6.4.1 Login Screen

In der Abbildung 6.1 ist der Login Screen zu sehen. Er besteht aus der Top-Action-Bar mit Logo und App-Name, den Eingabefeldern und zwei Buttons. Alle Komponenten sind aus der Material-Design-Bibliothek.
Zum Einloggen werden die Daten in die jeweiligen Felder eingegeben. Wenn ein Feld markiert ist, wird das ausgewählte Feld blau umrandet und der Hinweis wird auf die obere Linie verschoben. Das Passwortfeld zeigt nur kurz den eingegebenen Buchstaben an und wird dann zu einem „*“, sodass das Passwort nicht offen lesbar ist.
Der Login-Button sendet die Daten an das Backend und prüft, ob die Daten korrekt sind. Wenn dies der Fall ist, enthält die Antwort das Token, welches in den privaten Speicher abgelegt wird, und die App wechselt zum Hauptbildschirm. War der Login nicht erfolgreich, wird dem Benutzer eine Pop-Up-Meldung angezeigt und nichts weiter unternommen. Mit dem Betätigen des Registrieren-Knopfes wird man auf die Register-Activity weitergeleitet.

6.4.2 Main Activity

Auf dem Hauptbildschirm erscheint in der Top-Action-Bar ein drei Punkte Menü (Kebab-Menü), von dem aus man zu den Einstellungen gelangen oder sich ausloggen kann. Beim Ausloggen wird die Datei mit dem Benutzer-Token gelöscht und die Login-Activity aufgerufen.

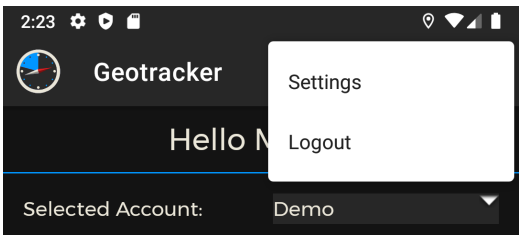


Abbildung 6.5: Menü auf dem Hauptbildschirm

Direkt unter der Top-Action-Bar wird der Benutzer mit dem Vornamen begrüßt (Abb.: 6.3).
In der Bedienfläche kann der Benutzer den Timetrack-Account auswählen und dessen Details ansehen, seine heutigen Arbeitszeiten ansehen und die Aufzeichnung starten oder stoppen.

Die Auswahl des Accounts erfolgt über ein Dropdown-Menü. Bei Auswahl wird sofort die zugehörige Beschreibung und die Vergütung angezeigt. Wenn die Aufzeichnung am Laufen ist, wird das Dropdown-Menü ausgeblendet. Das verhindert, dass der Benutzer eine Aktivität für einen anderen Account stoppen kann, als für den, auf dem er sie gestartet hat. Ist für den Benutzer noch kein Account vorhanden, wird „None“ im Menü angezeigt und die beiden Felder für Beschreibung und Vergütung werden ausgeblendet.

Für die Anzeige der heutigen Arbeitszeiten haben wir eine RecyclerView verwendet. Das Layout dazu wird in einer extra XML-Datei definiert und mit Daten in einer Adapter-Klasse befüllt. Durch eine Backendabfrage bekommen wir die nötigen Daten dafür. Bei aktiver Aufzeichnung wird ein Element angezeigt mit der Startzeit und der Info, dass das Ende offen ist.

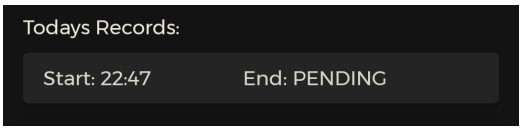


Abbildung 6.6: Laufende Aufzeichnung

Der Start-Stop-Button schaltet die Aufzeichnung um, in dem ein Backend-Endpunkt angesprochen wird. In der App haben wir eine boolean-Variable „running“ definiert, welche speichert, ob die Aufzeichnung aktiv ist. Anhand ihr wird entschieden, wie der Start-Stop-Button aussieht und ob beim Verlassen des Geofence noch gestoppt werden muss. Der Button ist nicht auswählbar, wenn sich der Nutzer außerhalb seines Arbeitsplatzes befindet und zeigt dies auch an (Abb.: 6.7). Ist der Nutzer dann im Bereich, wird „Start“ angezeigt und der Button ist freigeschaltet. Während der Aufzeichnung trägt der Button die Schrift „Stop“. Hat der Nutzer noch keine Geo-Daten für seinen Arbeitsplatz definiert, wird auch das auf dem Button angezeigt.

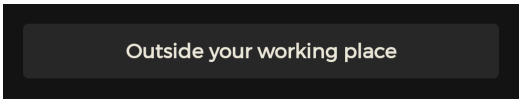


Abbildung 6.7: Nutzer außerhalb seines Geofence

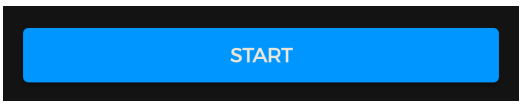


Abbildung 6.8: Aufzeichnung kann gestartet werden

Abbildung 6.9: Laufende Aufzeichnung

Abbildung 6.10: Nutzer hat noch keine Geo-Daten

Um versehentliches Stoppen der Aufzeichnung zu verhindern, muss der Nutzer in einem Pop-Up-Dialog seine Aktion bestätigen.

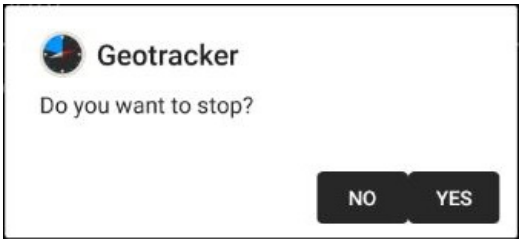


Abbildung 6.11: Bestätigungsdialog zum stoppen

6.5 Probleme und Lösungen

Damit die App auch die aktuellste Android Version unterstützt, mussten einige zusätzliche Punkte berücksichtigt werden. Neben der Berechtigung aus Kapitel 6.3.4 mussten in der „build.gradle“-Datei Kompilierungsoptionen gesetzt werden.

Zu Beginn wollten wir alle Activities mit Fragments realisieren, sodass es nur eine Activity gibt und alles weitere Fragments sind. Allerdings war es schwieriger, zwischen den Fragments zu wechseln, als in den Tutorials beschrieben. Deshalb sind wir auf reine Activities umgestiegen. Zwischen diesen ist das Hin- und Herschalten deutlich einfacher, es besteht jedoch kein Zugriff auf Elemente der anderen Activities.

Das wurde aber erst dann problematisch, als wir aus der Klasse „GeofenceBroadcastReceiver“ eine Methode der „MainActivity“ zur Änderung der Oberfläche aufrufen wollten. Das hat den Grund, dass Android nicht sicher sagen kann, dass diese Activity gerade auch aktiv ist. Deshalb haben wir den Weg über die Shared-Preferences gewählt mit einem Listener in der „MainActivity“.

Initial wollten wir das Token in einer Datenklasse abspeichern, welche beim Einloggen befüllt wird. Dazu müsste allerdings das Objekt oder die Referenz zu jeder anderen Activity übergeben werden.

Eine andere Möglichkeit stellen erneut die Shared Preferences dar. Das wäre auch eine gute Lösung gewesen, welche wir aber zu spät entdeckt haben. Deshalb haben wir das Problem mit dem privaten Speicher gelöst. Er ist durch andere Apps und den Benutzer nicht einsehbar, bildet deshalb also kein Sicherheitsrisiko.

Unerwartet war, dass die Geofence-Funktion die normale Android Positionsbestimmung zusätzlich benötigt. Denn zuerst hatten wir die Positionsbestimmung implementiert und dann die Geofence-Funktion, was funktioniert hat. Da in der Geofence-Funktion kein Code der normalen Positionsbestimmung referenziert wurde, dachten wir, man könne diesen weglassen, was ein Trugschluss war. Auch der Versuch, Teile der Positionsbestimmung wegzulassen, war ohne Erfolg. Deshalb beinhaltet die App auch Code für die normale Positionsbestimmung.

6.6 Deployment

Das Deployment spielte im Entwicklungsprozess der App keine große Rolle, da es Android-Studio benötigt um die App zu starten. Zum Abschluss haben wir allerdings den aktuellen Stand des Projekts in einer APK-Datei persistiert. Damit lässt sich die App auf andere Geräte installieren und in den App-Store laden. Zur Erstellung einer solchen APK muss ein Key zur Signatur angegeben werden.

7 Vollständiger Application Stack

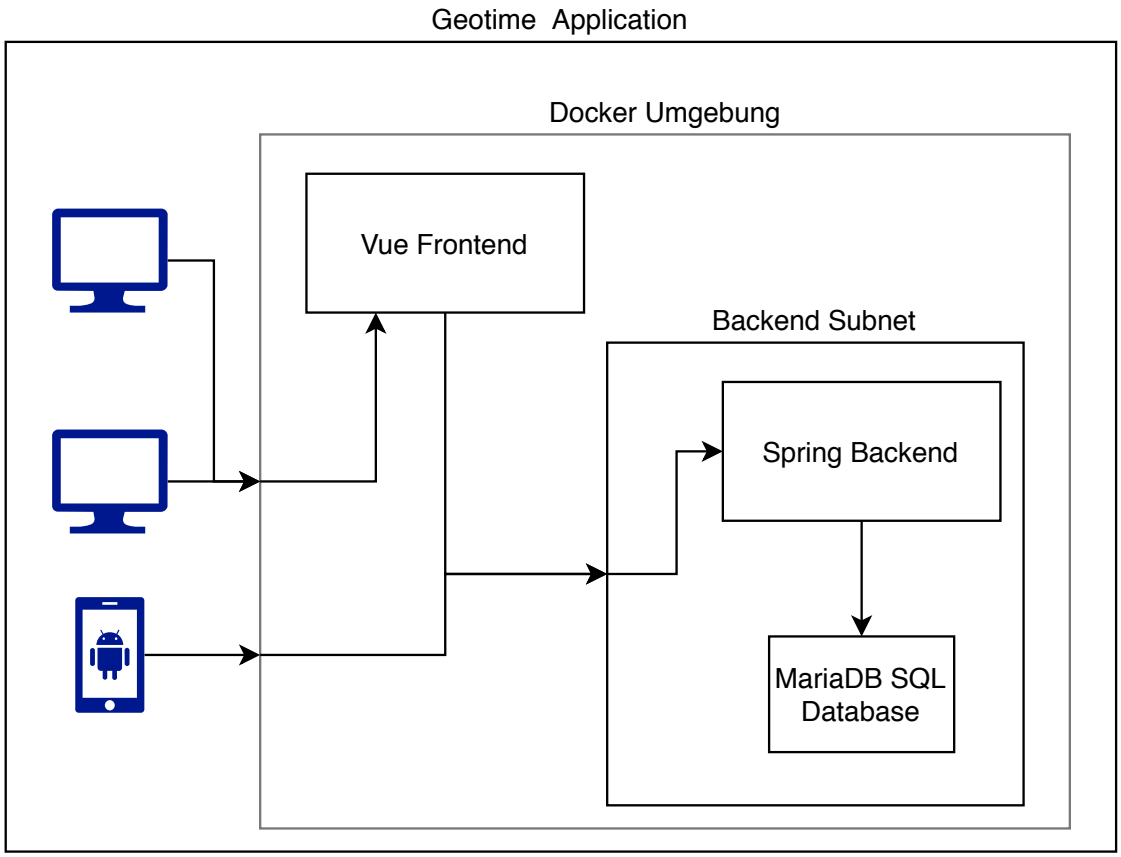


Abbildung 7.1: Application Stack

Das Deployment der Geo Timetracking Application ist in drei große Schichten aufgeteilt. Zunächst wäre hier die Backend Schicht, die Schicht der Datenhaltung und der API. Dieser Teil der Anwendung braucht am meisten Schutz, da er der wichtigste ist und dort alle Daten gespeichert werden. Der Zugriff auf die Datenbank ist nur auf das Backend beschränkt. Um nun die Applikation zu Nutzen gibt es zwei Möglichkeiten: Eine Android App oder ein Webbrowser.

Die Android App implementiert die View Schicht selbst und fragt nur für Daten den Backend-Dienst an. Diese Anfragen gehen zunächst an den Server, der die App hostet und werden dann von dem darauf laufenden Docker Deamon an den entsprechenden Container weitergeleitet.

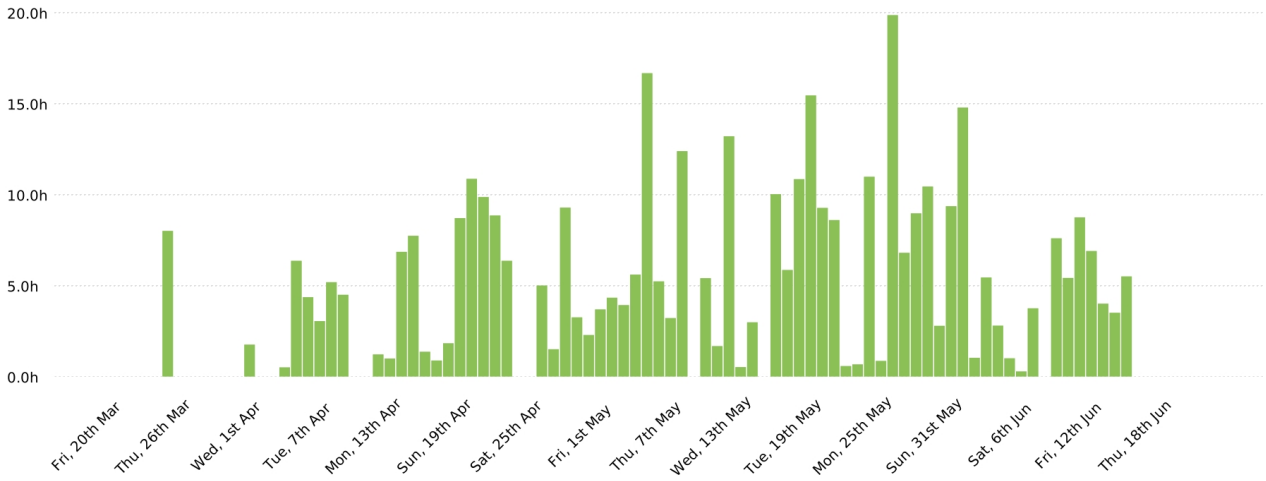
Beim Zugriff über den Webbrowser funktioniert die Kommunikation geringfügig anders. Zunächst wird vom Client der nginx Container nach dem statischen Teil der Website gefragt, dieser lädt dann über ähnliche Anfragen wie in der Android App die Daten vom Backend. Das global gesprochene Protokoll ist hierbei immer HTTP.

8 Projektjournal

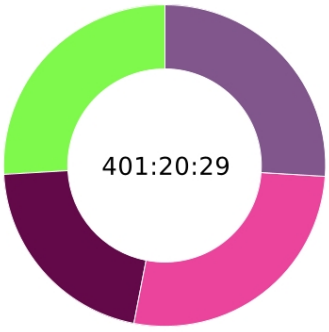
Summary report

20.03.2020 - 20.06.2020

Total: 401:20:29 Billable: 401:20:29 Amount: 0.00 USD



User



Marcel Schwarz	103:45:00	25.85%
Simon Kellner	84:41:51	21.10%
Tim Zieger	108:05:56	26.93%
Tobias Wieck	104:47:42	26.11%

Dieser Report zeigt eine Übersicht der geleisteten Arbeit jedes Gruppenmitglieds. Der vollständige Report ist separat angehängt. Dort kann jede Aktivität auf Issue Ebene genau nachvollzogen werden. Da lediglich die Issuenummern angegeben wurden, können die eigentlich dahinter liegenden Aufgaben auf GitLab[1] eingesehen werden.

[1] https://gitlab.com/marcel.schwarz/2020ss-qbc-geofence-timetracking/-/issues?scope=all&utf8=%E2%9C%93&state=all

9 Projektfazit und Ausblick

Bei dem Projekt im Rahmen von Ubiquitous/Pervasive Computing konnten wir Bekanntes anwenden und Neues lernen. Wir alle konnten uns gut einbringen und zusammen auf unser gemeinsames Ziel hinarbeiten. Im Rückblick auf die vergangenen fünf Sprints lässt sich sagen, dass diese erfolgreich verlaufen sind. Die Verteilung der Aufgaben war gleichmäßig und funktionierte reibungslos. Die Idee des Projekts konnte vollständig umgesetzt werden, zudem konnten anfangs nicht geplante Features umgesetzt werden. Hierzu zählen z.B. tagesübergreifende Time Records. Wir alle sind mit dem Ergebnis unserer Arbeit zufrieden und können das Projekt als erfolgreich bezeichnen.

Ebenso sehen wir ein großes Potential in der Weiterentwicklung unseres Endprodukts. Hier haben wir Ideen wie: Zuordnung der Benutzer in Gruppen, Benutzerprofile mit Daten über den Benutzer und dessen Tätigkeit oder auch Zuweisung von Kernarbeitszeit und Zeitrahmen, um Timetracking nur in einem festgelegten Zeitfenster zu erlauben. Mit ein paar Verbesserungen könnte unser Produkt von kleinen Unternehmen verwendet werden, die ein auf Vertrauen basiertes Zeitmeldesystem suchen.