BD1 Mountains

Projekt z przedmiotu Bazy Danych 1

Franciszek Urbański

17 I 2025

Spis treści

1 Koncepcja

Projekt ma na celu stworzenie serwisu do planowania i zapisywania tras górskich oraz przeglądania tras innych użytkowników i porównywania ich dokonań.

Niezalogowany użytkownik ma możliwość:

- Przeglądania rankingu użytkowników i ich profili
- Przeglądania tras użytkowników
- Założenia konta

Po zalogowaniu możliwe jest dodatkowo tworzenie, usuwanie oraz modyfikowanie własnych tras w łatwym w obsłudze edytorze.

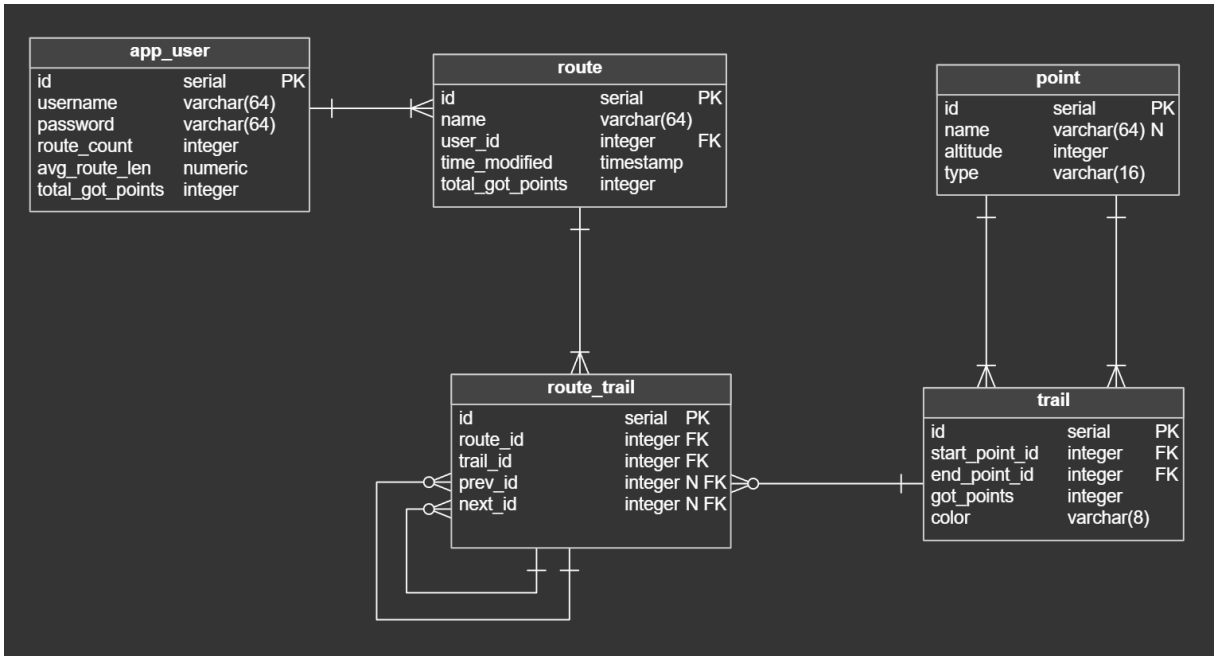
Z poziomu aplikacji, zgodnie z wymaganiami projektu, możliwe są również bezpośrednie operacje CRUD na tabelach bazy, oraz przywrócenie bazy do stanu początkowego (przykładowe dane - okolice Hali Gąsienicowej w Tatrach).

2 Projekt bazy danych

Projekt musi umożliwiać:

- Przechowywanie informacji o dostępnych szlakach, ich kolorach, długości oraz punktach startowych i końcowych
- Przechowywanie danych użytkowników - login+hasło, statystyki, trasy
- Przechowywanie tras - nazwa, statystyki, szczegółowy plan
- Łatwe wyciąganie planu trasy, wstawianie punktów na koniec i początek trasy
- Zapewnienie spójności tras

Rysunek 1. przedstawia opracowany diagram ERD bazy danych. Definicja bazy znajduje się w pliku **database/tables.sql**.



Rysunek 1: Diagram ERD

2.1 Tabele

2.1.1 point

Tabela zawierająca informacje o punktach w których krzyżują się szlaki:

- identyfikator
- nazwę (opcjonalna)
- wysokość nad poziomem morza
- typ - ograniczony do zbioru wartości ('peak', 'lake', 'shelter', 'pass', 'glade', 'valley', 'signpost', 'village', 'other') (szczyt, jezioro, schronisko, przełęcz, polana, dolina, drogowskaz, wioska, inne).

2.1.2 trail

Tabela zawierająca informacje o szlakach łączących punkty:

- identyfikator
- identyfikatory punktu startowego i końcowego - podwójna relacja 1:N między tabelami **point** a **trail**
- liczba punktów GOT które można otrzymać za przejście szlaku (jako miara dystansu)
- kolor szlaku (nazwa koloru, ograniczony do występujących w Polsce kolorów szlaków)

2.1.3 app_user

Tabela zawierająca dane zarejestrowanych w aplikacji użytkowników:

- identyfikator
- dane logowania (username (unikalne), password)
- statystyki (liczba tras, średnia długość trasy, sumaryczna liczba punktów GOT) - odświeżane przez triggery

Statystyki zostały dołączone do tabeli, aby nie trzeba było za każdym zapytaniem o ranking użytkowników wykonywać wymagającego zapytania (3 złączenia, potencjalnie duża ilość danych) - są one aktualizowane triggerami, mniejszymi kawałkami (w obrębie zmian statystyk jednej trasy).

2.1.4 route

Tabela zawierająca metadane o trasach:

- identyfikator
- nazwa trasy
- identyfikator użytkownika, który stworzył trasę - relacja 1:N między tabelą **app_user** a **route**
- czas ostatniej modyfikacji - odświeżany przez triggery
- sumaryczna liczba punktów GOT - odświeżana przez triggery

2.1.5 route_trail

Tabela pełniąca rolę encji asocjacyjnej w relacji M:N między tabelami **route** i **trail**, zawierająca również informację o kolejności szlaków. Kolejność szlaków jest zakodowana w sposób podobny do listy dwukierunkowej - kolumny **prev_id** (null dla punktu początkowego) i **next_id** (null dla punktu końcowego) są kluczami obcymi tabeli do samej siebie. Na wartości kolumny **next_id** oraz **prev_id** jest nałożony constraint UNIQUE, dzięki czemu nie ma możliwości mieszania tras między sobą oraz zapętlania istniejących sekcji.

Wybrałem taką strukturę, w odróżnieniu od prostego ponumerowania kolejnych szlaków liczbą porządkową, bo zapewnia to potencjalnie większą elastyczność - umożliwia np. łatwe wstawienie w środek trasy innego segmentu trasy. Tworzenie tras na podstawie szlaków a nie punktów wybrałem przez to, że bywają sytuacje gdzie między dwoma punktami przebiega więcej niż jeden szlak - wtedy wyznaczenie trasy przez punkty nie jest jednoznaczne.

Spójność tras mogłaby być zapewniona przez triggery na bazie danych, przy strukturze listy dwukierunkowej jest to jednak zbyt skomplikowane - wstawienie nowego szlaku wymaga też podmienienie adresu następnego w poprzednim punkcie, co powoduje wiele wywołań triggera i niezłe zamieszanie. W celu ułatwienia wstawiania i usuwania punktów do tras utworzyłem więc w bazie danych funkcje - **route_append(route_id, trail_id)**, **route_pop_back(route_id)**, **route_prepend(route_id, trail_id)**, **route_pop_front(route_id)**. Każda z tych funkcji wywołuje również funkcję **validate_route(route_id)**, która korzystając z funkcji pomocniczych **route_count_nulls(route_id, col_name)** oraz **route_find_incoherence(route_id)** sprawdza, czy dana trasa jest poprawna (ma po jednym punkcie startowym i końcowym, oraz czy wchodzące w jej skład kolejne szlaki faktycznie są ze sobą połączone).

Dla ułatwienia wyciągania listy kolejnych punktów z trasy, stworzony został widok **route_trail_ordered**, który zwraca listę szlaków w każdej trasie z liczbą porządkową definiującą kolejność.

2.2 Analiza

Projekt bazy:

- jest w 1NF - zapewnione przez SQL, żadna kolumna nie zawiera tabel jako wartości
- jest w 2NF, ponieważ dodatkowo w każdej tabeli, każdy atrybut jest nieredukowalnie zależny od klucza głównego
- jest w 3NF, ponieważ dodatkowo wszystkie kolumny tabeli są wzajemnie niezależne funkcyjnie

Wątpliwości może jedynie sprawiać projekt tabeli **route_trail**, gdzie w obrębie dwóch "sąsiednich"wierszy mamy pewną redundancję wprowadzoną przez kolumnę **next_id** z wcześniejszego i **prev_id** z późniejszego wiersza - obie kolumny są jednak konieczne do łatwego zachowania informacji o początku i końcu trasy, zależność nie jest też przez to funkcyjna.

3 Zapytania

3.1 Widok route_trail_ordered

Uwaga - nie zwraca szlaków po kolei, a jedynie z dodaną liczbą porządkową.

```
CREATE OR REPLACE VIEW mountains.route_trail_ordered AS
(
WITH RECURSIVE route_trail_list AS (
    SELECT route_id, 1 as ordinal, id, trail_id
    FROM mountains.route_trail
    WHERE prev_id IS NULL

    UNION ALL

    SELECT rt.route_id, rtl.ordinal + 1, rt.id, rt.trail_id
    FROM route_trail_list rtl
        JOIN mountains.route_trail rt ON rtl.id = rt.prev_id)
SELECT *
FROM route_trail_list
);
```

3.2 Funkcje i triggery

Zaimplementowane w pliku **database/tables.sql**, od linii 137. - ze względu na dużą objętość kodu, nie umieszczam ich bezpośrednio w dokumentacji.

3.3 Podstawowe operacje CRUD

Zaimplementowane generycznie w pliku **api/src/main/java/.../service/CrudService.java**.

```
SELECT * FROM mountains.? ORDER BY id                    -- wszystkie wiersze z tabeli
SELECT * FROM mountains.? WHERE id = ?                   -- wiersz o danym identyfikatorze
INSERT INTO mountains.? (...) VALUES (...) RETURNING id -- wstawienie wiersza
UPDATE mountains.? SET ... WHERE id = ?                  -- modyfikacja wiersza
DELETE FROM mountains.? WHERE id = ?                     -- usunięcie wiersza
```

3.4 Logowanie i rejestracja

Zaimplementowane w pliku **api/src/main/java/.../service/LoginService.java**.

```
-- logowanie - zwraca pustą encję jeśli użytkownik nie istnieje
SELECT id
FROM mountains.app_user
WHERE username = ? AND password = ?
```

```sql
-- rejestracja
INSERT INTO mountains.app_user (username, password)
VALUES (?, ?) RETURNING id
```

3.5 Edycja tras

Zaimplementowane w pliku **api/src/main/java/.../service/RouteEditorService.java**.

```sql
-- wyciągnięcie pełnej informacji o trasie (kolejne szlaki, punkty i ich szczegóły)
SELECT
    rto.ordinal,
    t.id,
    sp.name AS sp_name,
    sp.type AS sp_type,
    sp.altitude AS sp_altitude,
    ep.name AS ep_name,
    ep.type AS ep_type,
    ep.altitude AS ep_altitude,
    t.color,
    t.got_points
FROM mountains.route_trail_ordered rto
    JOIN mountains.trail t ON t.id = rto.trail_id
    JOIN mountains.point sp ON t.start_point_id = sp.id
    JOIN mountains.point ep ON t.end_point_id = ep.id
WHERE route_id = ?
ORDER BY rto.ordinal;

-- wyciągnięcie metadanych o trasie
SELECT
    r.id,
    r.name,
    u.username,
    r.time_modified,
    r.total_got_points
FROM mountains.route r
    JOIN mountains.app_user u ON r.user_id = u.id
WHERE r.id = ?;

-- lista szlaków, które można dołożyć na koniec trasy
WITH route_end_point_id AS (
    SELECT max(t2.end_point_id) AS id -- max returns null for no elements
    FROM mountains.route_trail rt
    JOIN mountains.trail t2 ON rt.trail_id = t2.id
    WHERE rt.route_id = ? AND rt.next_id IS NULL
)
SELECT
    cast(NULL as INT) AS ordinal,
    t.id,
    sp.name AS sp_name,
    sp.type AS sp_type,
    sp.altitude AS sp_altitude,
    ep.name AS ep_name,
    ep.type AS ep_type,
    ep.altitude AS ep_altitude,
    t.color,
    t.got_points
FROM mountains.trail t
    JOIN mountains.point sp ON sp.id = t.start_point_id
```

```
    JOIN mountains.point ep ON ep.id = t.end_point_id
    JOIN route_end_point_id rsp ON rsp.id = t.start_point_id OR rsp.id IS NULL;

-- lista szlaków, które można dołożyć na początek trasy
WITH route_start_point_id AS (
    SELECT max(t2.start_point_id) AS id -- max returns null for no elements
    FROM mountains.route_trail rt
    JOIN mountains.trail t2 ON rt.trail_id = t2.id
    WHERE rt.route_id = ? AND rt.prev_id IS NULL
)
SELECT
    cast(NULL as INT) AS ordinal,
    t.id,
    sp.name AS sp_name,
    sp.type AS sp_type,
    sp.altitude AS sp_altitude,
    ep.name AS ep_name,
    ep.type AS ep_type,
    ep.altitude AS ep_altitude,
    t.color,
    t.got_points
FROM mountains.trail t
JOIN mountains.point sp ON sp.id = t.start_point_id
JOIN mountains.point ep ON ep.id = t.end_point_id
JOIN route_start_point_id rsp ON rsp.id = t.end_point_id OR rsp.id IS NULL;

-- wywołania funkcji (append, prepend, pop_back, pop_front)
SELECT mountains.route_append(?, ?);
SELECT mountains.route_prepend(?, ?);
SELECT mountains.route_pop_back(?);
SELECT mountains.route_pop_front(?);

-- dodanie pustej trasy
INSERT INTO mountains.route (user_id, name) VALUES (?, ?) RETURNING id
```

3.6 Route explorer - ułatwienie odkrywania tras użytkownikom

Zaimplementowane w pliku **api/src/main/java/.../service/RouteExplorerService.java**.

```
-- najdłuższe trasy każdego użytkownika
WITH ranked AS (SELECT r.id as route_id,
                       r.name,
                       u.id as user_id,
                       u.username,
                       row_number() OVER (PARTITION BY u.username ORDER BY r.total_got_points DESC) a
                FROM mountains.route r
                    JOIN mountains.app_user u ON r.user_id = u.id)
SELECT route_id, name, user_id, username
FROM ranked
WHERE rn = 1;

-- trasy przekraczające 2000 m n.p.m.
SELECT r.id as route_id,
       r.name,
       u.id as user_id,
       u.username
FROM mountains.route r
    JOIN mountains.app_user u on u.id = r.user_id
    JOIN mountains.route_trail rt ON rt.route_id = r.id
```

```
    JOIN mountains.trail t ON rt.trail_id = t.id
    JOIN mountains.point sp ON t.start_point_id = sp.id
    JOIN mountains.point ep ON t.end_point_id = ep.id
GROUP BY r.id, u.id
HAVING max(greatest(sp.altitude, ep.altitude)) >= 2000;

-- trasy zawierające szlaki przynajmniej trzech różnych kolorów
SELECT r.id as route_id,
       r.name,
       u.id as user_id,
       u.username
FROM mountains.route r
    JOIN mountains.app_user u on u.id = r.user_id
    JOIN mountains.route_trail rt ON rt.route_id = r.id
    JOIN mountains.trail t ON rt.trail_id = t.id
GROUP BY r.id, u.id
HAVING count(distinct t.color) >= 3;
```

3.7 Informacje o użytkownikach

Zaimplementowane w pliku **api/src/main/java/.../service/UserInfoService.java**.

```
-- dane o wybranym użytkowniku, wraz z jego pozycją w głównym rankingu
WITH leaderboard AS (SELECT id,
                            dense_rank() OVER (ORDER BY total_got_points DESC) as r,
                            username,
                            route_count,
                            avg_route_len,
                            total_got_points
                     FROM mountains.app_user)
SELECT *
FROM leaderboard
WHERE id = ?;

-- trasy użytkownika
SELECT
    r.id,
    r.name,
    u.username,
    r.time_modified,
    r.total_got_points
FROM mountains.route r
    JOIN mountains.app_user u ON r.user_id = u.id
WHERE u.id = ?;

-- ranking użytkowników (na postawie różnych kryteriów - stąd ...)
SELECT
    id,
    dense_rank() OVER (ORDER BY ... DESC) AS r,
    username,
    route_count,
    avg_route_len,
    total_got_points
FROM mountains.app_user
ORDER BY r;
```

4 Projekt funkcjonalny

4.1 Baza danych

Baza danych PostgreSQL uruchomiona jest w serwisie NeonDB.

4.2 Backend

Backend, w postaci RESTful API, został napisany w języku Java przy użyciu frameworka Spring Boot oraz Spring JDBC do komunikacji z bazą. Kod źródłowy API znajduje się w folderze **api/**. API można uruchomić (pod warunkiem instalacji Java 21 oraz Apache Maven) za pomocą:

```
cd api
mvn spring-boot:run
```

API powinno uruchomić się na porcie 8080.

4.3 Frontend

Frontend jest napisany w języku TypeScript przy użyciu frameworka React i ReactBootstrap. Do zapytań AJAX wykorzystana jest biblioteka Axios. Można go zaserwować (pod warunkiem instalacji npm) za pomocą:

```
cd app
npm install
npm run build && npm run preview
```

Następnie należy otworzyć w przeglądarce wyświetlony URL.

4.4 Instrukcja obsługi aplikacji

4.4.1 Browse

Umożliwia przeglądanie użytkowników/tras. Mamy do wyboru dwie opcje:

- Leaderboard - ranking użytkowników. Można wybrać kryterium rankingu, oraz wejść w profile użytkowników znajdujących się w rankingu, z których natomiast można przeglądać ich trasy.
- Route explorer - przeglądanie istniejących tras, możliwość wyboru kryterium filtrowania, wejścia w szczegóły trasy oraz profil twórcy.

4.4.2 Account

Umożliwia logowanie/rejestrowanie się w serwisie. Dane przykładowych użytkowników to:

- username: *frun36*, password: *qwerty*
- username: *test*, password: *test*
- username: *sebix*, hasło: *piwo*
- username: *obibok*, hasło: *1234*

Można również szybko utworzyć nowe konto tym samym formularzem.

Po zalogowaniu/rejestracji dostępna jest lista wszystkich tras użytkownika, z możliwością modyfikacji, usuwania oraz stworzenia nowej trasy (po uprzednim wpisaniu jej nazwy). Następnie zostaje się przekierowanym do edytora tras, który pozwala dodawać lub usuwać na początek lub koniec trasy wybrane szlaki (wyświetlając jedynie te, które zachowają spójność trasy).

4.4.3 Raw operations

Zgodnie z wymaganiami projektu, udostępnione są tam formularze do wykonywania operacji CRUD na każdej tabeli w bazie danych. Uwaga - wykonywanie surowych operacji na tabeli **route_trail** wiąże się z ryzykiem zepsucia integralności tras, jako że ta tabela docelowo powinna być obsługiwana tylko przez funkcje udostępniane przez bazę danych.

Panel poniżej formularza z danymi gromadzi odpowiedzi od API, które można przeglądać za pomocą strzałek.

4.4.4 Reset DB

W tej zakładce udostępniono, zgodnie z wymaganiami, możliwość przywrócenia bazy danych do początkowego stanu (wypełniona przykładowymi danymi - okolice Hali Gąsienicowej w Tatrach). Aby tego dokonać, API (po otrzymaniu odpowiedniego żądania) wykonuje skrypt **database/tatra.sql**.

4.5 Dokumentacja techniczna

4.5.1 Backend

API napisane jest zgodnie ze standardowym dla Spring Boot wzorcem projektowym Service layer. Poszczególne warstwy aplikacji odpowiedzialne są za:

- Model - format danych, które wyciągamy z bazy
- Service - zapytania do bazy danych
- Controller - obsługa zapytań HTTP

W ramach każdej funkcjonalności API mamy więc część model, service i controller. Odpowiednie funkcjonalności są to zaś:

- **Crud** - generyczna implementacja prostych operacji CRUD dla wszystkich tabel. W warstwie model i controller mamy konkretne implementacje, znajdujące się w pakiecie **raw**. Dodanie obsługi nowej tabeli wymaga dodania odpowiedniego typu rekordu w pakiecie model, oraz stworzenie instancji kontrolera dla tego typu, implementując metodę zwracającą nazwę tabeli.
- **Login** - funkcjonalności logowania i rejestracji (podstawowe i absolutnie niezabezpieczone)
- **RouteEditor** - endpointy używane przy edytowaniu tras
- **RouteExplorer** - endpointy używane w funkcjonalności Route Explorer
- **UserInfo** - endpointy z informacją o użytkownikach oraz ich ranking

Sam kod pisany był w sposób samo-dokumentujący się (opisowe nazwy funkcji i zmiennych, jasna separacja), jest więc dość łatwy w zrozumieniu i utrzymaniu.

4.5.2 Frontend

Każda podstrona aplikacji znajduje się w osobnym komponencie React (każdy w osobnym pliku **src/components/*.tsx**). Poza podstronami, mamy również kilka innych użytecznych komponentów:

- **ApiResponsePanel** - panel dla odpowiedzi od API
- **RouteCard** - widok prezentujący metadane trasy

Język TypeScript, dzięki statycznemu typowaniu, umożliwił implementację aplikacji w sposób czytelny i dość łatwy w utrzymaniu. Najbardziej problematycznym miejscem może być generyczny komponent **Crud**, który wymaga sparametryzowania przez interfejs odpowiadający strukturze danych z tabeli, przekazania domyślnego obiektu tego interfejsu **defaultItem**, parametrów formularza **inputs** oraz nazwy tabeli **tableName**. Definicje wszystkich interfejsów i formularzy **Crud** znajdują się w komponencie **Raw**.