TerrainAwareRouting

Niniejsza dokumentacja opisuje prototypowy moduł przeznaczony do generowania rekomendowanych tras ewakuacji pomiędzy wybranym punktem początkowym a docelowym, z uwzględnieniem dynamicznych stref zagrożeń – w szczególności obszarów powodziowych. Projekt stanowi podstawę do budowy bardziej zaawansowanego systemu wspierającego podejmowanie decyzji w sytuacjach kryzysowych (np. powodzie, katastrofy infrastrukturalne), a jego architektura umożliwia łatwe rozszerzanie o kolejne źródła danych i algorytmy.

Moduł korzysta z danych drogowych pochodzących z OpenStreetMap, dostarczonych w formacie **GeoJSON** (lub opcjonalnie Shapefile po wcześniejszej konwersji). Dodatkowo integruje się z backendem Sentinel Hub w celu pobrania warstw „flood zones", reprezentujących wykryte obszary zalania na podstawie danych satelitarnych. Dane te są następnie przetwarzane do postaci wielokątów geometrycznych, które pozwalają na odfiltrowanie lub obniżenie priorytetu dla segmentów dróg znajdujących się w strefie zagrożenia.

Za obliczanie tras odpowiada zmodyfikowany wariant algorytmu Dijkstry, rozszerzony o mechanizmy omijania lub penalizowania segmentów znajdujących się w obszarze powodziowym. Takie podejście umożliwia dynamiczne generowanie najbezpieczniejszej dostępnej trasy, a nie tylko najkrótszej.

Udostępniony interfejs HTTP/API umożliwia pobranie rekomendowanej trasy za pomocą jednego zapytania:

```
GET /api/evac/route?start=lat,lon&end=lat,lon
```

Odpowiedź zawiera wynikową trasę w formacie GeoJSON oraz dodatkowe metadane dotyczące przebiegu obliczeń, takie jak liczba analizowanych segmentów, czas przetwarzania czy informacja o pominiętych obszarach zagrożeń.

Projekt został opracowany z myślą o późniejszej rozbudowie i łatwym przenoszeniu na inne środowiska. Kod jest modularny, udokumentowany i opatrzony logowaniem na wielu poziomach, co ułatwia debugowanie i analizę działania systemu. Repozytorium zawiera również zestaw testów jednostkowych oraz instrukcje niezbędne do uruchomienia narzędzia na dowolnym komputerze/serwerze, bez konieczności zadawania dodatkowych pytań o konfigurację.

RouteController – opis techniczny

`RouteController` udostępnia endpoint HTTP pozwalający na obliczenie trasy ewakuacyjnej pomiędzy dwoma punktami geograficznymi, z uwzględnieniem danych o

zalaniu terenu. Kontroler pełni rolę warstwy API, integrując trzy główne komponenty logiki domenowej:

- **GeoJsonRoadLoader** – odpowiedzialny za wczytywanie geometrii sieci drogowej z plików GeoJSON.
- **FloodOverlayService** – przetwarza listę segmentów drogowych i filtruje te, które są niedostępne z powodu zagrożenia powodziowego.
- **RouteService** – wyznacza trasę na podstawie przefiltrowanej sieci drogowej oraz współrzędnych punktu startowego i końcowego.

Endpoint

`GET /api/evac/route`

Parametry zapytania:

- `start` – współrzędne w formacie `lat,lon`
- `end` – współrzędne w formacie `lat,lon`

Przebieg działania:

1. **Parsowanie współrzędnych użytkownika**
   Metoda `parseCoord()` waliduje format wejściowy oraz konwertuje go do obiektu `Coordinate`.
   Uwaga: konstruktor `Coordinate` przyjmuje parametry w kolejności `(lon, lat)`.
2. **Wczytanie i przefiltrowanie segmentów drogowych**
   Kontroler pobiera pełną listę od `GeoJsonRoadLoader`, a następnie przekazuje ją do `FloodOverlayService`, który usuwa segmenty nieprzejezdne.
3. **Wyznaczenie bezpiecznej trasy**
   `RouteService.computeRoute()` zwraca listę punktów (węzłów trasy) reprezentujących obliczoną trasę z uwzględnieniem ograniczeń powodziowych.
4. **Budowa odpowiedzi API**
   Obliczone współrzędne przekształcane są do formatu `RouteStep`, zgodnego z oczekiwanym API (kolejność `lat`, `lon`).
   Zwracany obiekt `RouteResponse` zawiera listę kroków oraz pole błędu, aktualnie ustawione na `0`.

Obsługa błędów

- Niepoprawny format współrzędnych powoduje zgłoszenie `IllegalArgumentException`.

FloodOverlayService – Dokumentacja

Opis klasy

`FloodOverlayService` odpowiada za pobieranie, przetwarzanie i udostępnianie informacji o strefach zalewowych (flood zones) oraz filtrowanie bezpiecznych odcinków dróg.

- Wykorzystuje:
    - `RestTemplate` do pobierania danych JSON z backendu flood.
    - `ObjectMapper` do parsowania JSON.
    - `GeometryFactory` (JTS) do tworzenia obiektów geometrycznych (`Polygon`, `LineString`).
- Dane z backendu są buforowane w liście `floodZones`, aby uniknąć wielokrotnego pobierania.

Publiczne metody klasy

Metoda	Opis
`loadFloodZones()`	Pobiera dane z backendu, parsuje je i zapisuje w `floodZones`. Wywołuje metody pomocnicze: `fetchFloodData`, `extractFeatures`, `parsePolygons`.
`fetchFloodData()`	Pobiera surowy JSON z backendu. Zwraca `String`. Rzuca `IllegalArgumentException` w przypadku pustego lub nieudanego pobrania.
`extractFeatures(String json)`	Ekstrahuje tablicę `features` z JSON. Zwraca `List<JsonNode>`.
`parsePolygons(List<JsonNode> features)`	Konwertuje listę geometrycznych węzłów JSON do listy `Polygon`.
`parsePolygon(JsonNode geomNode)`	Tworzy `Polygon` z pojedynczego węzła `geometry`.

`validateGeometryNode(JsonNode geomNode)`	Sprawdza poprawność typu geometrii i obecność współrzędnych.
`extractCoordinates(JsonNode geomNode)`	Wyciąga współrzędne z węzła `Polygon` i zwraca `Coordinate[]`.
`filterSafe(List<RoadSegment> segments)`	Filtruje odcinki dróg, które nie przecinają żadnej strefy zalania.
`isSafe(Geometry road)`	Sprawdza, czy dana geometria drogi nie przecina żadnego poligonu zalania.

Notatki o testach jednostkowych

Testy wykorzystują **JUnit 5** oraz **AssertJ** i sprawdzają wszystkie kluczowe scenariusze:

1. **fetchFloodData**
    - Pobranie poprawnego JSON.
    - Rzucenie wyjątku przy pustym JSON.
2. **extractFeatures**
    - Poprawne wyciągnięcie tablicy `features`.
    - Obsługa niepoprawnej struktury JSON.
3. **parsePolygon**
    - Tworzenie `Polygon` z poprawnych współrzędnych.
    - Walidacja liczby wierzchołków.
4. **filterSafe**
    - Filtrowanie bezpiecznych odcinków dróg.
    - Usuwanie dróg przecinających strefy zalania.
5. **isSafe**
    - Zwracanie `true` dla drogi poza strefą zalania.

- Zwracanie `false` dla drogi przecinającej strefę zalania.

Przykładowy RoadSegment do testów:

```
public record RoadSegment(String id, LineString geometry, double
cost, boolean flooded) {}
```

Zalety i uwagi

- Wszystkie metody pomocnicze są publiczne, aby umożliwić testowanie jednostkowe.
- Klasa wspiera **wstrzykiwanie zależności** (`RestTemplate`, `ObjectMapper`, `GeometryFactory`) dla łatwiejszego testowania i mockowania.
- Obsługuje różne przypadki błędów (pusty JSON, brak `features`, brak współrzędnych).
- Wykorzystuje strumienie Java 8+ dla przetwarzania kolekcji i JSON.

Dokumentacja klasy `GeoJsonRoadLoader`

Cel klasy

`GeoJsonRoadLoader` odpowiada za wczytywanie i przetwarzanie danych dróg zapisanych w formacie **GeoJSON**.
Klasa konwertuje obiekty typu `LineString` na rekordy `RoadSegment`, które zawierają identyfikator, geometrię, koszt (długość) i informację, czy droga jest zalana (`flooded`).

Główne metody

Metoda	Opis	Typ zwracany
`loadRoadSegments()`	Wczytuje plik GeoJSON z dróg i zwraca listę segmentów. Loguje ilość wczytanych dróg.	`List<RoadSegment>`
`readGeoJson()`	Wczytuje plik GeoJSON z klasyloadera. Zwraca `Optional<JsonNode>` dla łatwiejszego testowania.	`Optional<JsonNode>`

`getFeatures(JsonNode root)`	Pobiera węzeł `features` z drzewa JSON.	`Optional<JsonNode>`
`getRoadSegments(JsonNode features)`	Przetwarza tablicę `features`, filtruje tylko `LineString` i tworzy obiekty `RoadSegment`.	`List<RoadSegment>`
`getGeometry(JsonNode feature)`	Pobiera obiekt `geometry` z pojedynczego feature.	`Optional<JsonNode>`
`isLineString(JsonNode geom)`	Sprawdza, czy geometria jest typu `LineString`.	`boolean`
`parseLineString(JsonNode geom)`	Konwertuje `LineString` w JSON na obiekt `LineString` z biblioteki JTS.	`LineString`
`createRoadSegment(LineString lineString)`	Tworzy rekord `RoadSegment` z geometrii.	`RoadSegment`

Uwagi implementacyjne

- Wszystkie metody są **package-private**, aby umożliwić testowanie jednostkowe.
- Wykorzystano `StreamSupport` i `Streams` z Javy 8 do przetwarzania tablic JSON.
- Obsługa wyjątków: w przypadku błędów wczytywania lub niepoprawnego formatu GeoJSON rzucany jest `IllegalStateException` lub `IllegalArgumentException`.
- Plik GeoJSON domyślnie pobierany jest z `classpath:roads.geojson`, ale można zmienić ścieżkę poprzez właściwość `app.roads.geojson-path`.

Testy jednostkowe

Cel testów

Testy mają na celu weryfikację poprawności przetwarzania danych GeoJSON i działania metod klasy `GeoJsonRoadLoader`. Skupiają się na testowaniu **poszczególnych kroków**, a nie integracji ze Springiem.

Testowane metody

1. **`getFeatures(JsonNode)`**
    - Testuje poprawne pobranie tablicy `features`.
    - Sprawdza sytuację, gdy tablica nie istnieje (`Optional.empty()`).
2. **`isLineString(JsonNode)`**
    - Sprawdza, czy metoda poprawnie identyfikuje geometrię typu `LineString`.
3. **`parseLineString(JsonNode)`**
    - Weryfikuje, że współrzędne JSON są poprawnie przetwarzane na obiekt `LineString`.
    - Sprawdza wyjątek, gdy współrzędne są brakujące.
4. **`createRoadSegment(LineString)`**
    - Testuje poprawne tworzenie rekordu `RoadSegment`.
    - Weryfikuje pola: `id`, `geometry`, `cost` i `flooded`.
5. **`getRoadSegments(JsonNode)`**
    - Testuje filtrację nie-LineString i poprawne tworzenie segmentów.
    - Weryfikuje liczbę wygenerowanych segmentów oraz poprawność geometrii.

Uwagi do testów

- Testy wykorzystują **JUnit 5** i **AssertJ** do asercji.
- Wszystkie testy działają na JSON-ach wbudowanych w testy (bez zależności od plików z resources).
- Testy są nazwane w stylu `should...correctly`, co zwiększa czytelność i dokumentuje oczekiwane zachowanie metod.

Dokumentacja `SafeDijkstraPathFinder`

1. Opis klasy

`SafeDijkstraPathFinder` jest implementacją strategii wyszukiwania najkrótszej ścieżki między dwoma punktami w grafie drogowym, bazującą na algorytmie Dijkstry.
Uwzględnia warunki bezpiecznej jazdy – **ignoruje zalane (`flooded`) segmenty dróg**.

Główne cechy:

- Wykorzystuje **PriorityQueue** i mapy odległości (`dist`) oraz poprzedników (`prev`) do efektywnego wyznaczania ścieżki.
- Obsługuje grafy reprezentowane przez listę `RoadSegment`.
- Wydzielone **mniejsze metody publiczne** ułatwiające testowanie i utrzymanie.
- Współpracuje z **Java 8+** (streamy, `computeIfAbsent`, `Collections.emptyList()`).

---

2. Struktura klasy i publiczne metody

Metoda	Opis	Testowalność
`findPath(List<RoadSegment> segments, Coordinate start, Coordinate end)`	Wyznacza najkrótszą ścieżkę z punktu `start` do `end`, pomijając zalane segmenty	✔
`buildAdjacencyMap(List<RoadSegment> segments)`	Tworzy mapę sąsiedztwa (od punktu startowego do listy segmentów), filtrując zalane drogi	✔
`initializeDistanceMap(Coordinate start)`	Inicjalizuje mapę odległości startując od punktu startowego	✔
`relaxEdges(Coordinate current, Map<Coordinate, List<RoadSegment>> adjacency, Map<Coordinate, Double> dist, Map<Coordinate, Coordinate> prev, PriorityQueue<Coordinate> queue)`	Przeprowadza operację "relaksacji" krawędzi z aktualnego wierzchołka	✔

`reconstructPath(Map<Coordinate, Coordinate> prev, Coordinate start, Coordinate end)`	Odtwarza ścieżkę od `end` do `start` na podstawie mapy poprzedników	✔
`extractCoordinate(RoadSegment seg, boolean startPoint)`	Pobiera współrzędną punktu startowego lub końcowego segmentu	✔
`extractNeighbor(RoadSegment seg)`	Pobiera współrzędną końcową segmentu (`neighbor`)	✔

Wszystkie metody są publiczne, aby ułatwić **testy jednostkowe**.

---

3. Notatki do testów

3.1 Podejście

- Testy napisane w **JUnit 5**.
- Assercje w **AssertJ** (`assertThat`) dla czytelności.
- Testy **pokrywają wszystkie publiczne metody**.
- Każdy test nazwany w formacie `should...Correctly`.
- Testy działają na prawdziwych rekordach `RoadSegment` i `LineString` z JTS, brak stubów.

3.2 Kategorie testów

Typ testu	Opis
Adjacency map	Sprawdza, że metoda `buildAdjacencyMap` poprawnie ignoruje zalane segmenty i tworzy mapę sąsiedztwa
Distance map	Testuje, że `initializeDistanceMap` ustawia odległość startową na 0

Coordinate extraction	Testuje `extractCoordinate` i `extractNeighbor` dla start/koniec segmentu
Relax edges	Sprawdza, że metoda `relaxEdges` aktualizuje odległości, mapę poprzedników i kolejkę priorytetową
Reconstruct path	Testuje poprawne odtworzenie ścieżki i zwrócenie pustej listy, jeśli brak ścieżki
Find path	Testuje pełną metodę `findPath` w scenariuszach pozytywnych i negatywnych (zalana droga)

3.3 Przykłady scenariuszy testowych

1. Graf prosty: A → B → C, wszystkie drogi dostępne → sprawdzamy `findPath`.
2. Graf z zalanym segmentem: A → B (flooded) → B nieosiągalne → ścieżka powinna być pusta.
3. Odtwarzanie ścieżki z mapy poprzedników (rekonstruowanie listy współrzędnych).
4. Relaxacja krawędzi i aktualizacja odległości w mapie.

---

4. Wskazówki dla deweloperów

- Przy dodawaniu nowych funkcji (np. różne typy kosztów lub priorytetów dróg) należy pamiętać o **aktualizacji testów `relaxEdges` i `findPath`**.
- W przypadku większych grafów warto rozważyć użycie **`Heap` zamiast PriorityQueue**, jeśli zależy nam na wydajności przy wielu aktualizacjach.
- Wszystkie metody publiczne zostały zaprojektowane tak, aby można je było **bezpośrednio testować jednostkowo**, co ułatwia refaktoryzację i debugowanie.
- Rekord `RoadSegment` jest niezmienny (`immutable`), więc można bezpiecznie używać go jako klucza w mapach (np. `dist`, `prev`).

📘 Dokumentacja – Klasa `AStarPathFinder`

1. Opis ogólny

`AStarPathFinder` implementuje algorytm **A*** na zbiorze punktów (koordynatów) pochodzących z geometrii `LineString` zawartych w klasie `RoadSegment`.

Algorytm wyznacza najkrótszą ścieżkę między punktami:

- **start** – współrzędna początkowa
- **end** – współrzędna końcowa
- graf jest zbudowany dynamicznie na podstawie segmentów
- sąsiedztwo punktów jest określone przez odległość przestrzenną
- heurystyka = euklidesowa odległość → algorytm jest optymalny w 2D

Klasa nie korzysta z wag `cost` z `RoadSegment` ani flagi `flooded` (ale logika może być rozszerzona).

---

2. Założenia implementacyjne

✔ Reprezentacja węzła

Każdy węzeł A* to rekord:

```
Node(coord, g, h, parent)
```

- `g` — koszt od startu
- `h` — heurystyka (distance → end)
- `f = g + h`
- `parent` — poprzednik w ścieżce

✔ Otwarte i zamknięte zbiory

- `open` → PriorityQueue sortowana po `f`

- `closed` → zbiór odwiedzonych współrzędnych

✔ Wyszukiwanie sąsiadów

Metoda:

```
neighborsOf(Coordinate c, List<RoadSegment> segments)
```

zwraca wszystkie punkty z segmentów, które:

1. **nie są równe** punktowi `c` (equals2D)
2. **są w odległości < 0.0003** od `c`

To oznacza, że graf jest bardzo gęsty przy dużej liczbie punktów i zależny od progu 0.0003.

---

3. Charakterystyka zachowania

1. Jeśli `end` jest w odległości < 0.0003 od startu → algorytm zakończy się natychmiast i ominie punkty pośrednie.
2. Jeśli punkty leżą w jednej linii, ale któryś z nich przekroczy próg 0.0003 względem startu, to nie będzie uznany za sąsiada.
3. Węzły są poprawnie śledzone po `parent` i rekonstrukcja zwraca pełną ścieżkę.

---

4. Ograniczenia

- brak wsparcia dla geometrii 3D (Z ignorowane)
- brak wykorzystania `cost` z `RoadSegment`
- brak logiki "flooded" (segmenty zalane nadal są dostępne)
- duże dane wejściowe → O(n²) relacji sąsiedztwa

🧪 Dokumentacja – Testy jednostkowe

Testy zostały przygotowane w **JUnit 5** z użyciem **AssertJ** i **Mockito**.
Mockujemy `LineString`, aby uniezależnić testy od biblioteki JTS i geometrii.

1. Testy dla `findPath()`

✔ Zakres pokrycia:

- brak segmentów → brak ścieżki
- najprostsza ścieżka liniowa
- wybór najkrótszej ścieżki przy wielu opcjach
- brak powrotów do closed set (sprawdzanie cykli)
- start == end
- poprawne wywołanie `geometry.getCoordinates()`

✔ Najważniejsza uwaga

Progi odległości `< 0.0003` determinują, czy algorytm zobaczy dany punkt.

Przykładowy problem wykryty w testach:

> punkt pośredni był tak blisko startu, że algorytm przechodził bezpośrednio do punktu końcowego omijając punkt pośredni

Dlatego w testach zmodyfikowano współrzędne, aby wymusić łańcuch sąsiedztwa:

`start → mid → next → end`

gdzie:

- dist(start, mid) < 0.0003
- dist(mid, next) < 0.0003
- dist(start, end) ≥ 0.0003

Wymusiło to prawidłowe działanie A* jak w klasycznej siatce.

---

2. Testy dla `neighborsOf()`

Testy sprawdzają:

1. brak sąsiadów gdy brak segmentów
2. filtrowanie wyłącznie punktów bliższych niż 0.0003
3. pomijanie punktu `c` (self)
4. agregację punktów z wielu segmentów
5. wywołanie `getCoordinates()` w mocku
6. scenariusze graniczne (prawie na granicy 0.0003)

✔ Najważniejsza obserwacja z testów

Wielu programistów natrafia na błąd testów, gdy punkty układają się tak:

```
start → mid → end
```

ale:

```
distance(start, end) < 0.0003
```

Wtedy `neighborsOf(start)` zwraca:

```
[mid, end]
```

i A* przyciągany heurystyką może **ominąć mid**.

Testy zostały dostosowane, aby to wykluczyć poprzez zmianę współrzędnych.

📌 Rekomendacje dla przyszłych testów

- rozważyć parametrację progu sąsiedztwa zamiast stałej 0.0003

- dodać testy z realnym `GeometryFactory` JTS (integracyjne)
- dodać testy scenariuszy z `flooded == true` po implementacji logiki blokowania
- dodać testy wykorzystujące `cost` zamiast samych odległości
- w przypadku dużych map dodać testy performance (profiling A*)

🔧 Obsługa błędów – `DefaultErrorDto` oraz `DefaultErrorHandling`

1. `DefaultErrorDto` – standardowa struktura błędu

`DefaultErrorDto` jest obiektem transferowym (DTO), który reprezentuje ustandaryzowaną odpowiedź błędu zwracaną przez API. Dzięki temu wszystkie błędy mają spójny format, co ułatwia obsługę po stronie klienta.

Pola obiektu:

- **`timestamp`** – czas wystąpienia błędu (`java.util.Date`)
- **`status`** – kod HTTP błędu (np. 404, 400, 500)
- **`error`** – tekstowy opis statusu HTTP (np. „Not Found”, „Bad Request”)
- **`message`** – komunikat błędu przeznaczony dla klienta
- **`path`** – ścieżka żądania, w którym wystąpił błąd

DTO jest niezmienny (immutable) — wszystkie pola ustawiane są w konstruktorze generowanym przez Lombok (`@AllArgsConstructor`), a dostęp do nich odbywa się przez gettery.

---

2. `DefaultErrorHandling` – centralna obsługa wyjątków

`DefaultErrorHandling` to klasa oznaczona adnotacją `@ControllerAdvice`, która przechwytuje wyjątki zgłaszane przez kontrolery i zamienia je na odpowiedzi JSON o spójnym formacie (`DefaultErrorDto`). Dzięki temu eliminujemy konieczność lokalnego przechwytywania wyjątków w każdym kontrolerze.

Klasa definiuje trzy metody obsługi wyjątków:

a) `NoSuchElementException` → 404 Not Found

Metoda:

`handleNoSuchElementException(...)`

Zwraca błąd:

- **HTTP 404**
- komunikat: *„Zasób nie istnieje”*

Używana, gdy żądany zasób nie został znaleziony.

b) `IllegalArgumentException` → 400 Bad Request

Metoda:

`handleIllegalArgument(...)`

Zwraca błąd:

- **HTTP 400**
- komunikat pobrany z wyjątku (np. „Invalid data”)

Obsługuje sytuacje, gdy klient przesyła niepoprawne dane wejściowe.

c) `Exception` (dowolny inny wyjątek) → 500 Internal Server Error

Metoda:

`handleGeneral(...)`

Zwraca błąd:

- **HTTP 500**
- komunikat: *„Wewnętrzny błąd serwera”*

Pełni funkcję „bezpiecznego punktu końcowego” przechwytującego wszystkie nieobsłużone wyjątki.

---

3. Testy jednostkowe – `DefaultErrorHandlingTest`

Testy weryfikują poprawne działanie wszystkich trzech metod obsługi błędów:

- zwracane kody HTTP,
- poprawność pól w `DefaultErrorDto`,
- zachowanie komunikatów,
- obecność `timestamp` oraz `path`.

Zapewniają, że format błędów jest stabilny, a obsługa wyjątków działa zgodnie z założeniami projektu.

Podsumowanie

Zestaw klas `DefaultErrorDto` + `DefaultErrorHandling` stanowi centralny mechanizm obsługi błędów w projekcie, gwarantując:

✔ spójny format komunikatów błędów
✔ czytelną i przewidywalną strukturę odpowiedzi dla klienta
✔ prostszą i czystszą logikę w kontrolerach
✔ ułatwione testowanie i rozszerzanie

Mechanizm ten jest łatwy do rozwinięcia o kolejne wyjątki, logowanie zdarzeń czy obsługę specyficznych błędów domenowych.