Tour Planner SWEN2

Ibrahim Moalim, Yehor Zudikhin

GitHub Repository: https://github.com/imoalim/TourPlanner

Architecture Description

The TourPlanner backend is designed with a strong adherence to SOLID principles, ensuring high cohesion and loose coupling across various components. This design facilitates maintainability, scalability, and testability. A unique feature of this system is the deployment of generated TourReports and SummarizeReports to AWS S3 cloud object storage, enhancing accessibility and robustness of the reporting functionality.

Key Components and Design Decisions

Report Generation and Deployment:

- **ReportGenerator (Abstract Class):** This abstract class follows the Single Responsibility Principle (SRP) by focusing on report generation and file upload logic. It uses Thymeleaf templates for generating HTML content and iTextRenderer for PDF creation.
- **TourReportGenerator and SummarizeReportGenerator (Concrete Classes):** These classes extend ReportGenerator, implementing specific logic for generating tour and summarize reports respectively. They adhere to the Open/Closed Principle (OCP) by being extendable without modifying existing code.
- **AWS S3 Integration:** The S3FileUploadService handles file uploads to AWS S3, encapsulating the complexity of S3 interactions and adhering to the Dependency Inversion Principle (DIP) by depending on abstractions rather than concrete implementations.

Service Layer and Business Logic:

- **GenericService (Interface):** Determines predefined CRUD methods that are similar for TourServiceImpl and TourLogServiceImpl.

- **TourServiceImpl:** Implements the business logic for managing tours. It uses the ORSService to fetch route details and calculates various attributes like distance and estimated time.
- **TourLogServiceImpl:** Manages tour logs, ensuring consistency and updating related tour attributes upon changes. Both services follow the Interface Segregation Principle (ISP) by exposing only relevant methods to the controller layer.

Exception Handling:

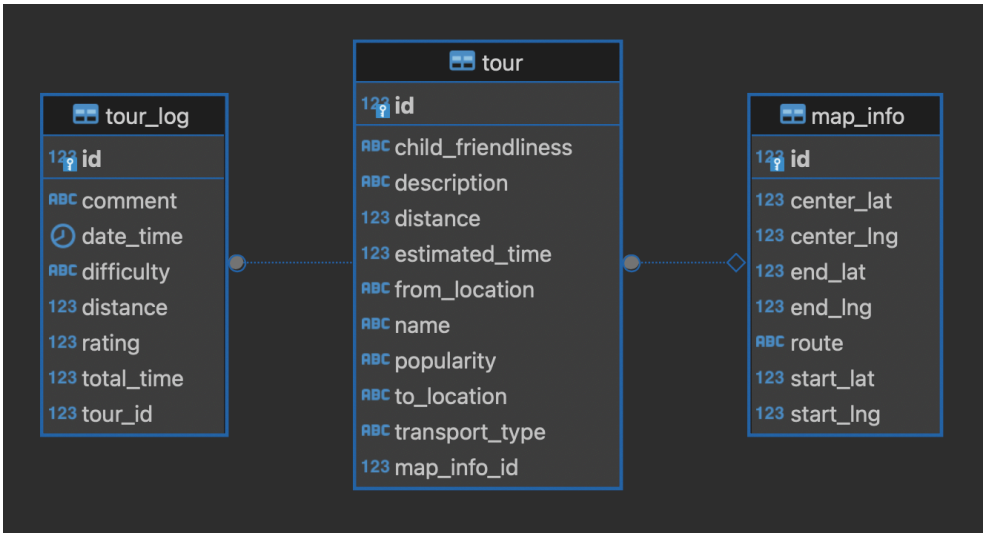
- **GlobalExceptionHandler**: Centralized exception handling using @ControllerAdvice to manage application-wide exceptions and provide consistent error responses.
- **ResourceNotFoundException**: Custom exception to handle cases where requested resources are not found, ensures clarity and specific error messaging.

Controller Layer:

- **TourController and ReportController**: These controllers manage HTTP requests, delegate business logic to services, and ensure proper response handling. They stick to the Single Responsibility Principle (SRP) by focusing only on request handling and response generation.

Entities and DTOs:

- **Entities**: Represent the database models for tours, associated tour logs and map info. They encapsulate the core attributes and relationships. Class diagram:



- **DTOs**: Data Transfer Objects used to transfer data between different layers, ensuring separation of concerns and data encapsulation: TourDTO, TourLogDTO, SummaryDTO, ReportResponseDTO, MapInfoDTO.

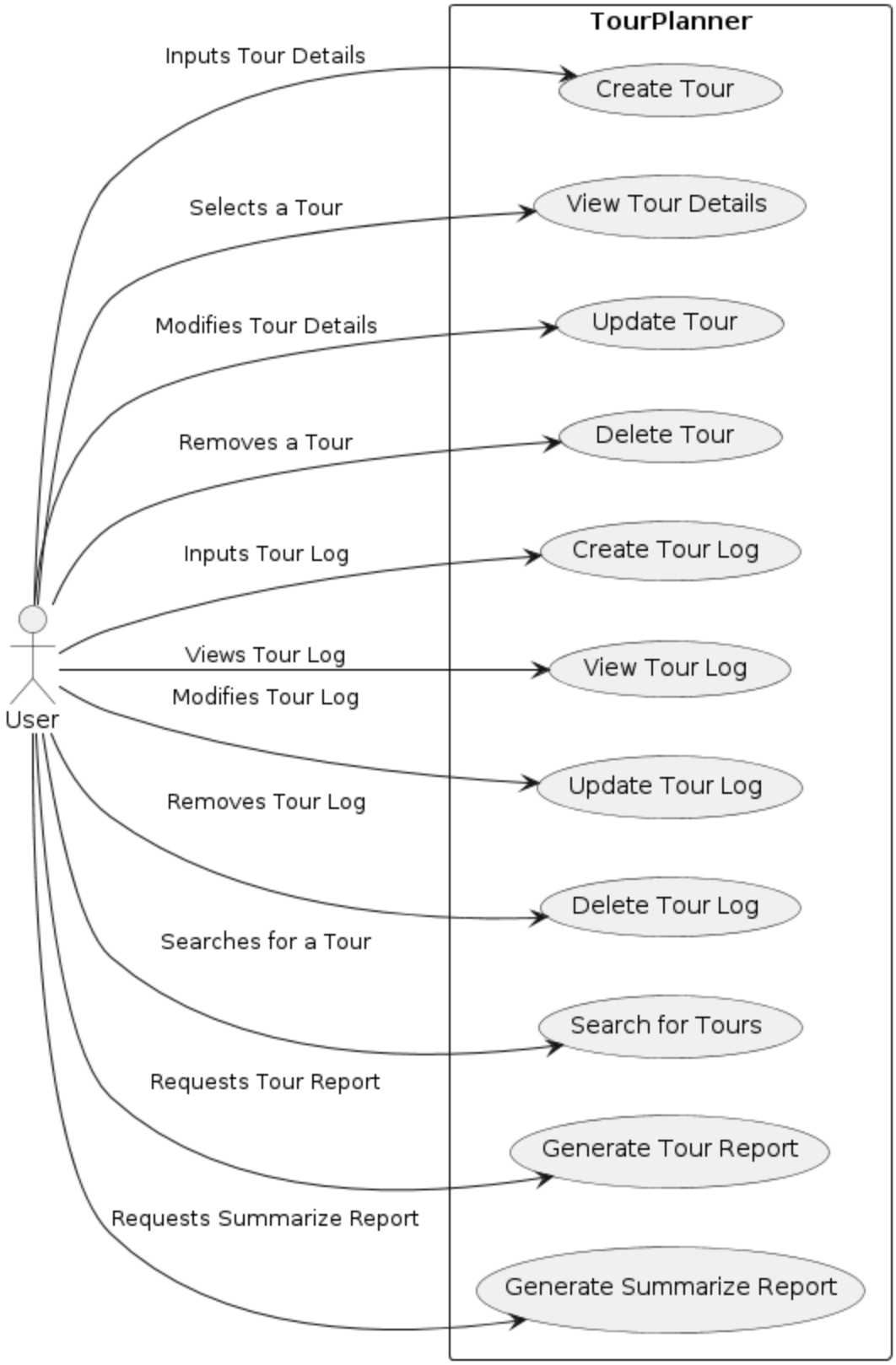
SOLID Principles Applied

- **Single Responsibility Principle**: Each class has a single responsibility. For example, ReportGenerator is responsible for report generation, while TourServiceImpl handles tour-related business logic.
- **Open/Closed Principle**: The ReportGenerator class is open for extension but closed for modification. New types of reports can be added by extending this class without altering the existing code.
- **Liskov Substitution Principle**: Subtypes (e.g., TourReportGenerator and SummarizeReportGenerator) can replace their base type (ReportGenerator) without affecting the correctness of the program.
- **Interface Segregation Principle**: The service interfaces expose only the necessary methods, keeping them lean and focused.
- **Dependency Inversion Principle**: High-level modules do not depend on low-level modules but on abstractions. For instance, TourLogController depends on GenericService interface rather than its concrete implementation.

Unique Features

- **Deployment to AWS S3**: The system generates and uploads tour reports and summary reports to AWS S3, ensuring accessibility and robustness.
- **Global Exception Handling**: A custom global exception handler provides consistent and clear error handling across the application.
- **Thymeleaf Template for Map Generation**: The map, including coordinates for routes, is generated using Thymeleaf templates and sent to the frontend, enhancing the user experience with dynamically generated content.

Use Case Diagram



Unit Test Decisions

TourControllerTest

The following tests use the `@WebMvcTest` Spring Boot annotation and ensure that the controller correctly handles requests and input validation, and that it interacts appropriately with the service layer.

- *whenCreateTourHasValidInput_thenReturn200*: Verifies that a tour with valid input can be created successfully.
- *whenCreateTourHasNullValue_thenReturn400*: Ensures that input validation is working correctly by returning a 400 Bad Request when the input is invalid, i.e. when name, fromLocation, toLocation or transportType are missing.
- *whenCreateTourHasValidInput_thenCallsService*: Checks that the service method is called with the correct parameters when a tour is created.
- *whenCreateTourHasValidInput_thenReturnTourResource*: Validates that the correct resource is returned after creating a tour.

TourLogControllerTest

The next tests ensure that the controller properly handles tour log creation requests and validates inputs, ensuring seamless interaction with the service layer.

- *whenCreateTourLogHasValidInput_thenReturn200*: Verifies that a tour log with valid input can be created successfully.
- *whenCreateTourLogHasNullTourId_thenReturn400*: Ensures that a 400 Bad Request is returned when the tour ID is null, validating input.
- *whenCreateHasValidInput_thenCallsService*: Checks that the service method is called with the correct parameters when a tour log is created.

ReportControllerTest

The following tests ensure that the controller handles report generation requests correctly, including error handling for non-existent tours.

- *whenGetTourReportHasValidInput_thenReturn200*: Verifies that a tour report is returned successfully when valid input is provided.
- *whenGetTourReportHasInvalidInput_thenReturn404*: Ensures that a 404 Not Found status is returned when the tour ID does not exist.
- *whenCreateTourReportHasValidInput_thenCallService*: Checks that the service method for generating a tour report is called correctly.

- *whenCreateSummarizeReport_thenCallService*: Verifies that the summarize report service method is called correctly.

TourServiceImplTest

These tests ensure the service layer correctly handles business logic for creating, updating, deleting, and retrieving tours, including handling edge cases where data is not found.

- *createTour*: Verifies that a tour is created correctly and saved to the repository.
- *findAllTours*: Ensures that all tours can be retrieved from the repository.
- *findTourById*: Checks that a tour can be found by its ID and handles cases where the tour is not found.
- *updateTour*: Verifies that an existing tour is updated correctly.
- *deleteTour*: Ensures that a tour can be deleted by its ID.
- *updateComputedTourAttributes*: Verifies that computed attributes of a tour are updated correctly based on its logs.

TourLogServiceImplTest

The following tests ensure the service layer correctly handles business logic for creating, updating, deleting, and retrieving tour logs, including handling cases where data is not found.

- *createTourLog*: Verifies that a tour log is created correctly and saved to the repository.
- *findAllTourLogs*: Ensures that all tour logs can be retrieved from the repository.
- *findTourLogById*: Checks that a tour log can be found by its ID and handles cases where the tour log is not found.
- *updateTourLog*: Verifies that an existing tour log is updated correctly.
- *deleteTourLog*: Ensures that a tour log can be deleted by its ID.

ReportServiceTest

The tests ensure the service layer correctly handles the logic for generating reports and handles cases where the tour does not exist.

- *generateTourReport_TourExists*: Verifies that a tour report is generated successfully when the tour exists.
- *generateTourReport_TourNotExists*: Ensures that an exception is thrown when attempting to generate a report for a non-existent tour.

- *generateSummarizeReport*: Verifies that the summarized report is generated successfully.