Protocol for Tour Planner Application

Contents

[1. Introduction 2](#_Toc199086564)

[2. Technical Solutions/Decisions 2](#_Toc199086565)

[2.1 Design 2](#_Toc199086566)

[2.2 Implementation Specifications 2](#_Toc199086567)

[2.3 Library decisions and lessons learned 3](#_Toc199086568)

[3. UML – Diagrams 4](#_Toc199086569)

[3.1 Use Case 4](#_Toc199086570)

[3.2. Class Diagram 4](#_Toc199086571)

[4. UI Flow with Wireframes 5](#_Toc199086572)

[5. Unit Testing Design 5](#_Toc199086573)

[6. Time Tracking 6](#_Toc199086574)

[7. Lessons Learned 6](#_Toc199086575)

[8. Git Repository Link 6](#_Toc199086576)

# Introduction

The Tour Planner application empowers users to effortlessly design, organize and document a wide range of tours—whether cycling, hiking, running or vacation adventures—through an intuitive JavaFX interface that adheres to the Presentation-Model pattern. Its layered architecture cleanly separates responsibilities: the UI layer handles all graphical interactions, the business layer enforces tour-related rules and workflows, and the data access layer manages the storage, retrieval and updating of tour definitions and log entries.

With just a few clicks, users can create new tours, view or modify existing ones, maintain detailed logs, remove outdated records and generate comprehensive reports to track their experiences over time.

# Technical Solutions/Decisions

## Design

**The application is built on a three-tiered architecture** that promotes maintainability and clear separation of concerns. The presentation tier, implemented in JavaFX with FXML, handles all user interactions and keeps layout definitions distinct from event-handling logic.

**Beneath it, the business tier processes** incoming requests, applies validation and enforces rules before passing data along. Finally, the data access tier uses JPA with Hibernate to persist and retrieve information from a PostgreSQL database, ensuring efficient, reliable storage of tours and logs.

To further decouple components and simplify testing, the **Presentation-Model pattern** is employed in the JavaFX layer, cleanly isolating view-related code from business logic.

In addition, key services—such as the OpenRouteService—are **implemented as singletons**, guaranteeing a single shared instance throughout the application and optimizing resource utilization.

## Implementation Specifications

**The user interface is crafted in JavaFX with FXML**, keeping layout definitions entirely separate from application logic and enabling the creation of custom components—such as an AutoCompleteTextField—that dynamically suggest options as users type.

**Behind the scenes, tours and logs are persisted via JPA with Hibernate to a PostgreSQL database**, while any images associated with a tour are stored on the filesystem to preserve database performance.

Route data—distance, duration, and waypoints—**is retrieved in real time from the OpenRouteService API**, and map tiles are rendered using the OpenStreetMap Tile Server to give users an interactive, visual overview of their planned journeys.

For operational transparency and troubleshooting, all runtime events and **errors are recorded with log4j**, ensuring that any issues can be diagnosed and addressed quickly. The application’s reliability is further guaranteed by a comprehensive suite of over **JUnit tests**, which rigorously verify core functionality ranging from tour creation, editing, and deletion to log management and external API integration.

**Unique Feature,** In the General tab, just below the tour description, we’ve added a distinctive visual element: an image that brings each tour to life. By displaying a map or a representative photograph alongside the details, users can more quickly recognize and recall their journeys, making the interface both more engaging and easier to navigate.

Under the hood, this image is associated with each tour record in the database but stored separately on the filesystem for optimal performance. **Whenever a user opens the General tab, the application retrieves the linked file and renders it in place**, seamlessly integrating the visual with the textual description.

## Library decisions and lessons learned

**Library Decisions:**

1. **JavaFX**: Chosen for building the graphical user interface due to its rich set of UI components and ease of integration with Java applications.

2. **Spring Boot**: Selected for application configuration and dependency management, offering a streamlined setup and powerful features for building Java applications.

3. **JPA/Hibernate**: Used for ORM and database interaction to simplify data access and manipulation, leveraging its robust framework for managing relational data.

4. **PostgreSQL**: Picked as the database engine for its reliability, performance, and extensive support for advanced features.

5. **log4j**: Implemented for logging due to its flexible configuration and powerful logging capabilities.

6. **JUnit**: Chosen for unit testing to ensure code quality and reliability, providing a well-supported framework for writing and running tests.

**Lessons Learned**:

- Integrating external APIs requires careful handling of network exceptions and response validations to ensure robustness.

- Maintaining a clean separation between UI and business logic simplifies testing and enhances code maintainability.

- Proper configuration management is crucial for managing environment-specific settings and secrets securely.

# 3. UML – Diagrams

## 3.1 Use Case

- Actors: User

- Use Cases:

- Create Tour: Allows users to create new tours by providing necessary details.

- Modify Tour: Users can update details of existing tours.

- Delete Tour: Users can remove tours from the system.

- View Tour List: Displays a list of all created tours.

- Create Tour Log: Users can log details of completed tours.

- Modify Tour Log: Allows updating existing tour logs.

- Delete Tour Log: Users can delete tour logs.

- View Tour Log List: Displays logs associated with a selected tour.

- Generate Reports: Users can generate detailed and summary reports of tours and logs.

## 3.2. Class Diagram

- Classes:

- `TourPlannerApplication`: The main class that initializes and starts the application.

- `MainViewController`: Manages the main view and coordinates between different controllers.

- `GeneralTabController`: Handles the display and interaction of the general tab, showing tour details.

- `NewTourFormController`: Manages the form for creating and editing tours.

- `TourListViewController`: Handles the list view of tours and interactions with it.

- `OpenRouteService`: Provides methods to interact with OpenRouteService.org API.

- `TourService`: Contains business logic related to tours.

- `TourLogService`: Manages tour logs and their business logic.

- `Tour`: Entity class representing a tour.

- `TourLog`: Entity class representing a tour log.

# UI Flow with Wireframes

On launch, the **Main Screen** greets you with a comprehensive list of all your tours, each entry accompanied by intuitive controls for adding new tours or editing and deleting existing ones, making navigation and management quick and effortless.

**Selecting any tour opens the Tour Details view**, where you’ll see everything from its name and description to the origin, destination, and mode of transport, alongside a map image that brings your route into focus; from here you can also manage individual tour logs.

Whenever you want to plan a new adventure or tweak an existing one, **the New Tour Form presents:** fields for the tour’s name, description, origin and destination points, transport type, and even lets you upload a representative image.

Finally, the **New Log Form guides you through documenting** each outing—just enter the date, your comments, difficulty level, total distance and time, and give it a rating to capture all the details of your experience.

# Unit Testing Design

**The heart of the application’s functionality lies in three critical code areas, each ensuring reliability and correctness across different features.**

**First, the Tour Creation and Modification** component guarantees that new tours can be added and existing ones updated without error. It performs thorough data validation—such as checking for required fields, valid coordinates, and allowable transport types—and enforces integrity rules to prevent inconsistent or incomplete tour records.

Next, **the Tour Log Management module** handles the association between individual logs and their parent tours. It verifies that every log entry is correctly linked to its corresponding tour, enforces constraints on log data (like valid dates and ratings), and ensures that edits or deletions maintain the overall consistency of the tour’s history.

Finally, **the API Integration layer** is rigorously tested by mocking external responses, confirming that route data from the OpenRouteService API is fetched and parsed correctly. These tests validate that the UI updates seamlessly with live distance, duration, and waypoint information, and that error conditions—such as timeouts or malformed responses—are handled gracefully.

# Time Tracking

- **Initial Setup and Planning**: 20 hours

- **UI Design and Implementation**: 25 hours

- **Business Logic Implementation**: 20 hours

- **Data Access Layer Implementation**: 30 hours

- **API Integration**: 15 hours

- **Testing and Bug Fixing**: 30 hours

- **Documentation and Protocol**: 5 hours

- **Total Time Spent**: 145 hours

# Lessons Learned

- **Challenges**:

- Ensuring seamless integration between JavaFX UI components and business logic was challenging, requiring careful synchronization and data binding.

- Handling asynchronous API calls and updating the UI accordingly required robust exception handling and background task management.

- **Solutions**:

- Utilized JavaFX's binding properties and listeners to achieve real-time UI updates based on changes in the underlying data model.

- Implemented comprehensive exception handling and logging to capture and resolve issues efficiently, ensuring a smooth user experience.

# Git Repository Link

[githublink](https://github.com/if23b229/tour_planner_lamthi_kiri_puka.git)