# Tour Planner - Project Documentation

This document provides a detailed overview of the Tour Planner desktop application, covering its architecture, features, design patterns, and development insights.

**1. Application Architecture**

The Tour Planner is a desktop application built with JavaFX, following the **Model-View-ViewModel (MVVM)** architectural pattern. This pattern promotes a clean separation of concerns between the user interface (View), the UI state and logic (ViewModel), and the business logic and data (Model and Services).

Communication between decoupled components (like different ViewModels) is managed through a custom **Event Bus** (Publisher/Subscriber pattern), which prevents tight coupling and improves maintainability.

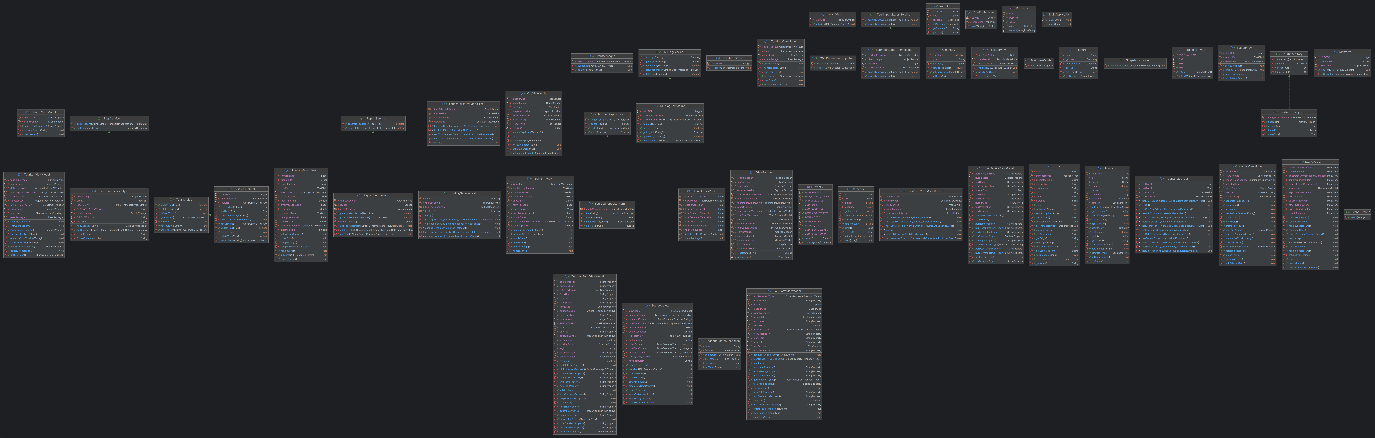
**Architectural Layers**

The application is structured into several distinct layers:

1. **View Layer** (com.teameight.tourplanner.view)
   * **Contents:** FXML files for UI layout and corresponding Java controller classes (\*View.java).
   * **Functionality:** Responsible for displaying the user interface and delegating user actions (e.g., button clicks) to the ViewModel. It observes the ViewModel's properties and updates the UI in response to state changes.
2. **Presentation Layer (ViewModel)** (com.teameight.tourplanner.presentation)
   * **Contents:** \*ViewModel.java classes.
   * **Functionality:** Acts as an intermediary between the View and the Model/Services. It holds the UI state (e.g., the text in a search field, the list of tours), exposes it to the View via JavaFX properties, and contains the presentation logic for handling user input and preparing data for display.
3. **Service Layer (Business Logic)** (com.teameight.tourplanner.service)
   * **Contents:** Service interfaces and their implementations (\*ServiceImpl.java).
   * **Functionality:** Encapsulates the core business logic of the application. It coordinates operations between repositories and external services. For example, TourService manages the lifecycle of tours, while MapService handles communication with the OpenRouteService API.
4. **Repository Layer (Data Access)** (com.teameight.tourplanner.repository)
   * **Contents:** Repository interfaces and their ORM implementations (\*RepositoryOrm.java).
   * **Functionality:** Abstracts the data persistence mechanism. It provides a clean API for CRUD (Create, Read, Update, Delete) operations on domain entities without exposing the underlying database details. This layer uses **Hibernate (JPA)** to interact with a **PostgreSQL** database.
5. **Model Layer** (com.teameight.tourplanner.model)
   * **Contents:** POJO (Plain Old Java Object) classes like Tour, TourLog, and enums like TransportType.
   * **Functionality:** Represents the application's domain data. These classes are used by all layers, from being JPA entities in the repository to being displayed in the View.

**Class Diagram**

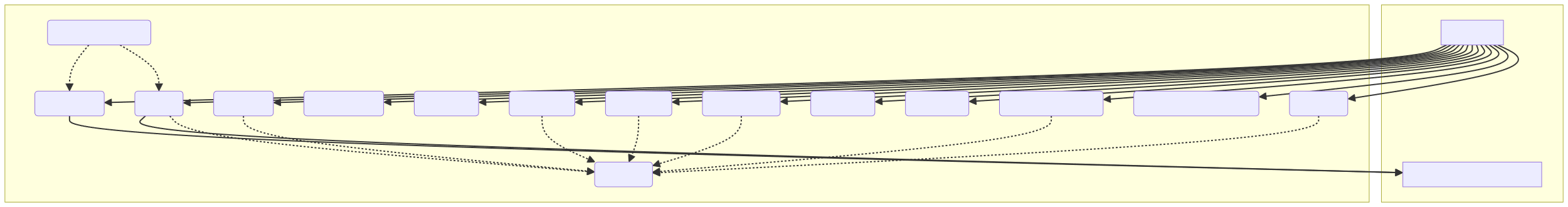
The following diagram illustrates the key classes and their relationships within the architecture.



**2. Use Cases**

**Use-Case Diagram**

The primary actor is the **User**, who can perform several key actions within the application.



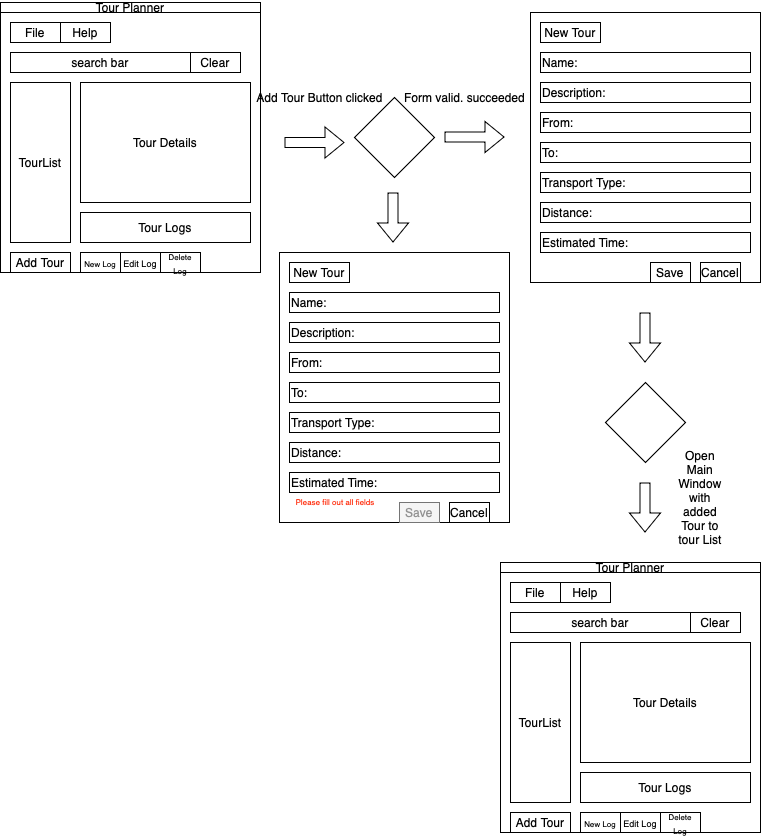
If texts are not visible, please see the file “Use\_Case\_Diagram.svg”

No Sequence Diagram as this is way too difficult for me

**3. User Experience (UX)**

The user interface is designed to be intuitive and efficient. The main window is divided into logical sections to provide all relevant information at a glance.

The wireframe below, found in WireframeER\_KRASNIQI.drawio.png, outlines the structure of the main application window.



* **Left Pane:** Contains the search bar for quick filtering and the main list of all available tours. Below the list is a prominent "Add Tour" button for easy access to the creation workflow.
* **Center Pane:** This area is split vertically into three sections:
  1. **Tour Details:** Displays comprehensive information about the currently selected tour.
  2. **Map View:** Shows a visual representation of the tour's route.
  3. **Tour Logs:** A table listing all logs associated with the selected tour, with controls to add, edit, or delete logs.
* **Top Bar:** A standard menu bar provides access to all application features, including file operations (import/export), report generation, and help.

**4. Library and Technology Decisions**

The project leverages several well-established libraries to ensure robustness and functionality.

|  |  |  |
| --- | --- | --- |
| Library/Technology | Version(s) | Purpose |
| JavaFX | 17.0.6 | The core framework for building the rich-client desktop user interface. |
| Maven | 3.8.5 (Wrapper) | A powerful build automation and dependency management tool, standard for Java projects. |
| Hibernate/JPA | 6.6.11 / 3.2.0 | Industry-standard Object-Relational Mapping (ORM) framework to abstract database interactions. |
| PostgreSQL | 17 (Docker) / 42.7.5 | A robust, open-source relational database for persisting tour and log data. Managed via Docker Compose for easy setup. |
| Jackson | 2.18.3 | High-performance library for processing JSON, used for handling API responses and the import/export feature. |
| Apache PDFBox | 3.0.5 | A standard Java library for creating and manipulating PDF documents, used for generating tour reports. |
| Log4j 2 | 2.24.3 | A flexible and powerful logging framework for application monitoring and debugging. |
| JUnit 5 & Mockito | 5.11.4 / 5.17.0 | The modern standard for unit testing in Java, enabling a test-driven approach and ensuring code quality. |
| Leaflet.js | 1.8.0 | A lightweight, open-source JavaScript library for interactive maps, integrated via a JavaFX WebView. |

**5. Lessons Learned**

* **MVVM and Data Binding:** The MVVM pattern, combined with JavaFX's data binding capabilities, is extremely powerful for creating maintainable UIs. However, it requires a disciplined approach to ensure that all state is managed within the ViewModels and that bindings are correctly established.
* **Decoupling with an Event Bus:** The event bus was crucial for allowing components like the SearchViewModel and TourListViewModel to communicate without direct dependencies. This makes the system more modular but requires careful tracking of event flows during debugging.
* **Database Schema Management:** Using hibernate.hbm2ddl.auto=create in persistence.xml is excellent for development as it automatically creates the schema. For a production environment, this would be dangerous and should be switched to validate or managed with a dedicated migration tool like Flyway or Liquibase.
* **Manual Dependency Injection:** The ViewFactory acts as a simple, manual dependency injection (DI) container. While effective for this project's scale, a larger application would benefit from a dedicated DI framework like Spring or Guice to manage object creation and wiring automatically.

**6. Implemented Design Patterns**

The project incorporates several fundamental design patterns:

* **Model-View-ViewModel (MVVM):** The core architectural pattern separating UI, presentation logic, and business logic.
* **Singleton:** The ViewFactory is implemented as a singleton to ensure a single point of access for creating view and ViewModel instances.
* **Factory Method:** The ViewFactory.create() method acts as a factory, encapsulating the logic for instantiating different view components.
* **Observer (Publisher/Subscriber):** The EventManager implements this pattern, allowing objects (subscribers) to be notified of state changes in other objects (publishers) without being tightly coupled.
* **Repository:** The data access layer uses the Repository pattern to abstract the data source, providing a collection-like interface for domain objects.
* **Facade:** The ReportService and MapService act as facades. They provide a simple, unified interface to more complex underlying subsystems (the PDFBox library and the OpenRouteService API, respectively).

**7. Unit Testing Decisions**

The testing strategy focuses on verifying the application's logic while keeping the tests fast and isolated.

* **Focus on Logic Layers:** Unit tests are concentrated on the **ViewModel** and **Service** layers, as these contain the application's core logic. The View layer (FXML and UI controllers) is not unit-tested, as this is better suited for integration or end-to-end testing.
* **Mocking Dependencies:** **Mockito** is used extensively to create mock objects for dependencies. For instance, TourServiceTest mocks the TourRepository to test the service's logic in isolation from the database. Similarly, TourFormViewModelTest mocks both TourService and MapService to verify its validation and interaction logic without making real service or API calls.
* **Behavior Verification:** Tests use Mockito.verify() to ensure that methods on mocked objects are called as expected (e.g., verifying that tourRepository.save() is called exactly once when adding a new tour).
* **Testing Event-Driven Components:** The TourDetailsViewModelTest demonstrates an effective technique for testing event-driven components. It uses an ArgumentCaptor to capture the EventListener subscribed to the EventManager, allowing the test to simulate events and assert that the ViewModel's state updates correctly.

**8. Unique Feature: Keyboard Shortcuts**

To enhance user experience and efficiency, the application implements keyboard shortcuts (accelerators) for many common actions. These allow power users to navigate and operate the application without relying solely on the mouse.

Key shortcuts are defined in navbar.fxml and include:

* **New Tour:** Ctrl+N
* **Edit Tour:** Ctrl+E
* **Delete Tour:** Delete
* **Import Tours:** Ctrl+I
* **Export Tours:** Ctrl+X
* **Export Map:** Ctrl+M
* **Generate Tour Report:** Ctrl+R
* **Generate Summary Report:** Ctrl+S
* **Exit Application:** Alt+F4
* **Help:** F1

**Link to GitHub:** https://github.com/Tamior930/tourplanner