Marvin Löhlein, Noah Howadt

TourPlanner Protocol

Architecture and Features

Our Tourplanner application is divided into a frontend using JavaFX, and a backend using Java Spring. The frontend is using the MVVM pattern and we also implemented services/managers to communicate with our backend. The backend has controllers, services and repositories, with the controllers handling the requests, services containing the business logic and repositories handling data access.

Users can manage tours and tour logs, import/export tours and generate reports. This enables users to see, how other people liked a certain tour, and how much time it took them to complete. Furthermore, the reports of the tours provide a great overview of a certain tour, or all tours that are saved in the system.

When adding or editing a tour, locations are suggested to ensure that the correct start and end locations are used. The Open Route Service is used to provide these suggestions. On input, a request is sent to the Open Route Service via our backend, which returns 5 suggestions for the given input, including the coordinates. When a suggestion is selected and the tour is saved, a display name and the coordinates are saved in the database.

External libraries

We included open-source libraries into our project to streamline development and avoid reinventing well-established solutions. This allowed us to focus on core functionality while benefiting from tested, reliable code.

Leaflet

We used Leaflet to present a map view of the tour to the user. Leaflet is a popular open-source JavaScript library for interactive maps. It was chosen for its simplicity, active community support, and ease of integration via a WebView in our application.

Log4j

To handle logging, we integrated Log4j. This library is widely used and well-supported, offering flexible and efficient logging capabilities. It helped us maintain a consistent and scalable logging strategy throughout the project.

OpenPDF

We used OpenPDF to generate PDF documents within the app. We selected it because of its simplicity and lightweight design, which made it easy to implement without introducing unnecessary complexity.

Marvin Löhlein, Noah Howadt

Design patterns

MVVM (Model–View–ViewModel) Pattern

The MVVM (Model–View–ViewModel) design pattern is implemented in the JavaFX frontend to ensure a clear separation of concerns between the UI and business logic. View components such as EditLogController and TourListController are responsible solely for user interaction and bind their UI elements directly to observable properties in their respective ViewModels (e.g., EditLogViewModel, TourListViewModel). These ViewModels contain the logic for validation and state management, while delegating all data-related operations to service classes like LogManager or TourManager. This structure promotes modularity, enhances testability, and improves overall maintainability.

Observer Pattern

The Observer pattern is used throughout the application to enable event-driven communication between loosely coupled components. It is implemented via PropertyChangeSupport and PropertyChangeListener in classes such as EditLogViewModel and TourListViewModel. For example, when a log is saved or an editing operation is canceled, the ViewModel notifies registered listeners, allowing the UI or other components to react appropriately. This design promotes flexibility and responsiveness without introducing tight coupling between components.

Service Pattern

The Service design pattern is applied in both the frontend and backend layers to encapsulate business logic and provide clean interfaces for data operations. In the backend, service classes such as LogService and TourService handle core logic and delegate persistence to repositories. In the frontend, manager classes like LogManager and TourManager expose similar service-like functionality to ViewModels. By centralizing logic in dedicated services, the application maintains a clean architecture with clearly defined responsibilities and high reusability.

Unit Testing

This project applies a unit testing approach across both frontend (JavaFX) and backend (Spring-style) components. All tests are written using JUnit 5 and Mockito, with a clear focus on correctness, isolation, and maintainability.

Frontend (JavaFX ViewModels & Service Managers)

LogManagerTest and TourManagerTest validate the core business logic such as creation, deletion, and updates of LogItem and TourItem objects. Event-based behaviors (e.g. REFRESH_LOG) are verified using AtomicBoolean flags and listener assertions. Edge

cases are explicitly tested, e.g., handling of null or missing IDs during deletion to ensure robustness. ViewModel tests (EditLogViewModelTest, EditTourViewModelTest) ensure that UI-bound properties are correctly loaded, reset, and persisted. Listener invocation and data transformation (e.g. RatingOption → String rating) are verified using mock listeners and argument capturing.

Backend (Service Layer)

LogServiceTest and TourServiceTest focus on verifying the interaction between the service layer and the repository. Mocks are used to isolate logic and verify calls (e.g. save(), deleteById()). Logic-heavy methods such as distance and duration calculation in TourService are tested using mocks for OpenRouteService. Negative scenarios, such as updating a non-existent log, are tested to ensure proper exception handling and robustness.

The project took about 12 work days each to complete.

https://github.com/Methuzalem/Tour-Planner-SS2025-