Tourplanner-Documentation

Students:

Gabriel Helm, if21b024

Gerrit Kreuzer, if21b008

Total time spent on project:

Helm: 122h

Kreuzer: 106h

Github: https://github.com/GerritKreuzer/Tourplanner.git

Design choices:

We decided to split our application into three layers, namely the Business Logic (BL), Data Access Layer (DAL), and TourPlannerApp, to achieve easier overview and maintainability.

Business Logic (BL):

The "BL" directory contains the business logic of the application. By separating the business logic into its own component, the architecture achieves a clear separation of concerns. This separation helps with maintainability and testability, as modifications to the business rules can be made independently without affecting other components.

Data Access Layer (DAL):

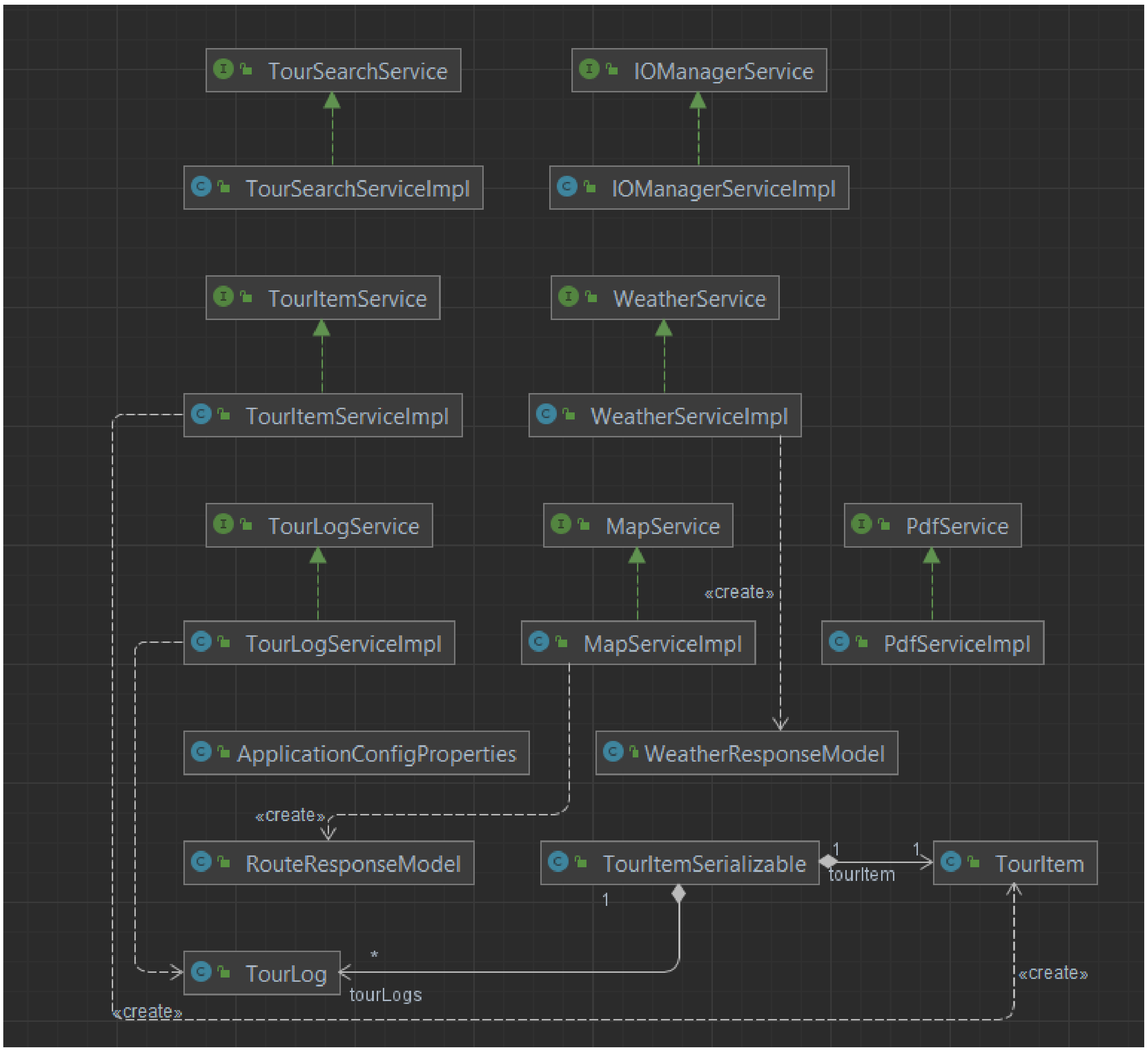
The "DAL" directory handles data access and persistence operations. This design choice simplifies the management of data-related operations, such as interactions with databases or other data sources.

TourPlannerApp:

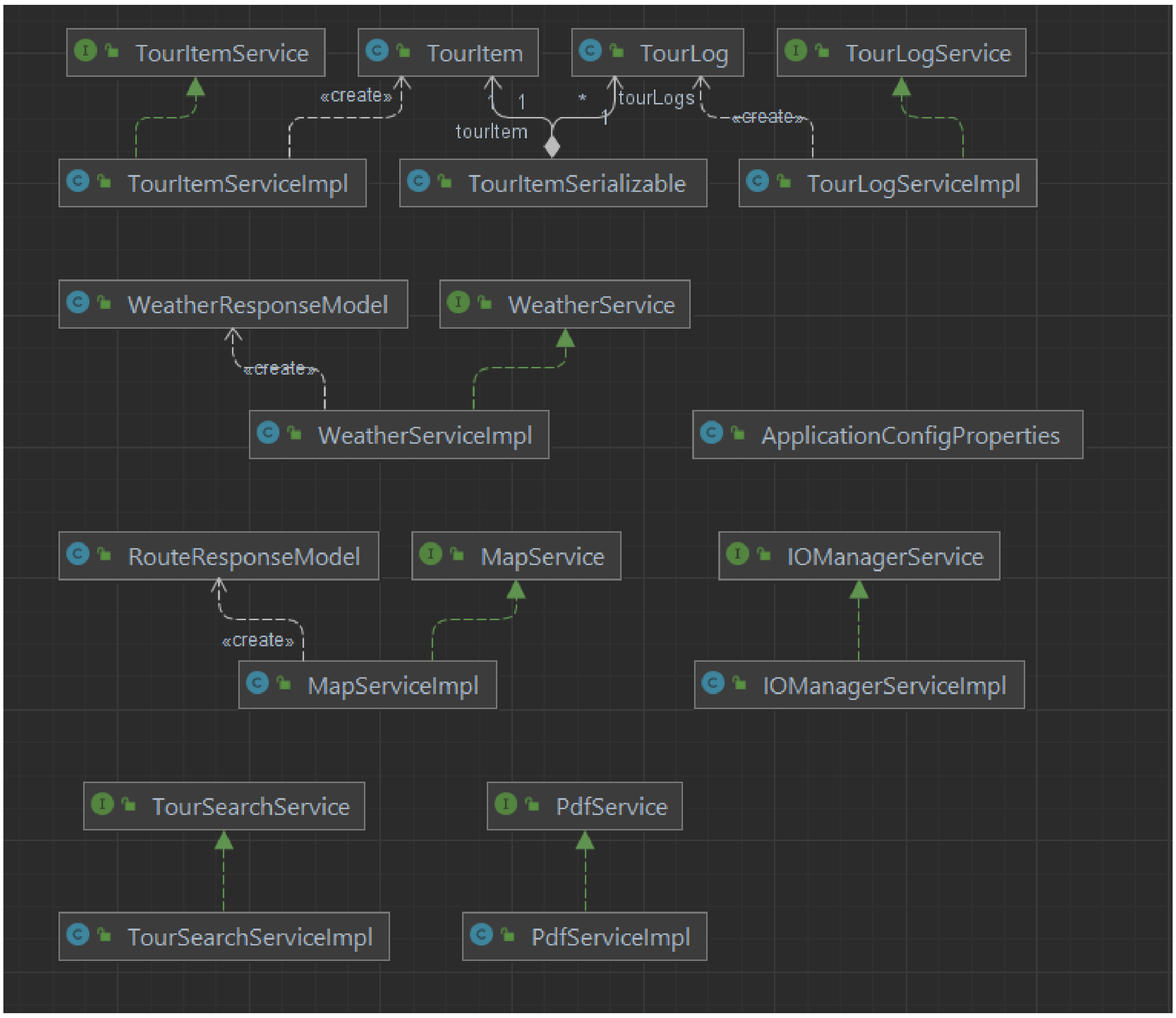
The "TourPlannerApp" directory represents the user interface or application module. The separation of the user interface into its own component enables independent development and updates to the UI without impacting the underlying business logic or data access layer.

Class diagrams:

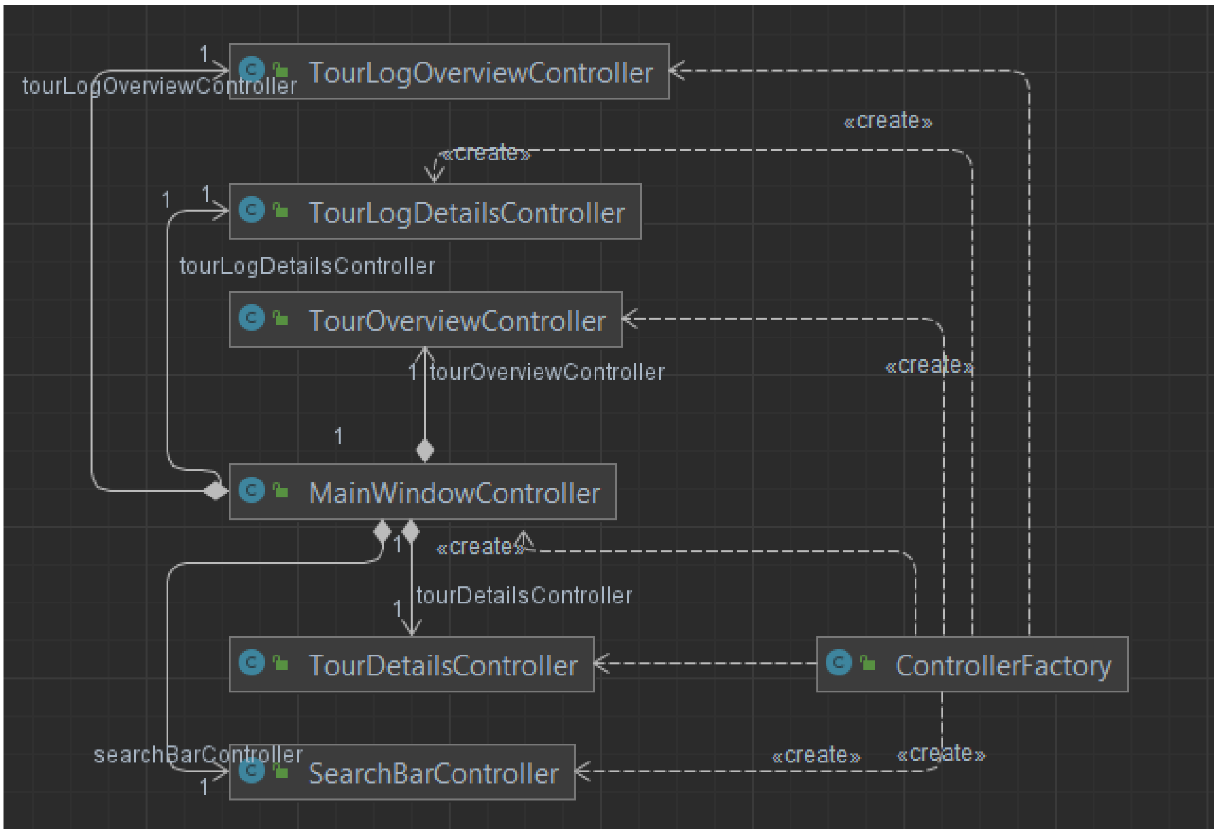
BL-Classdiagram



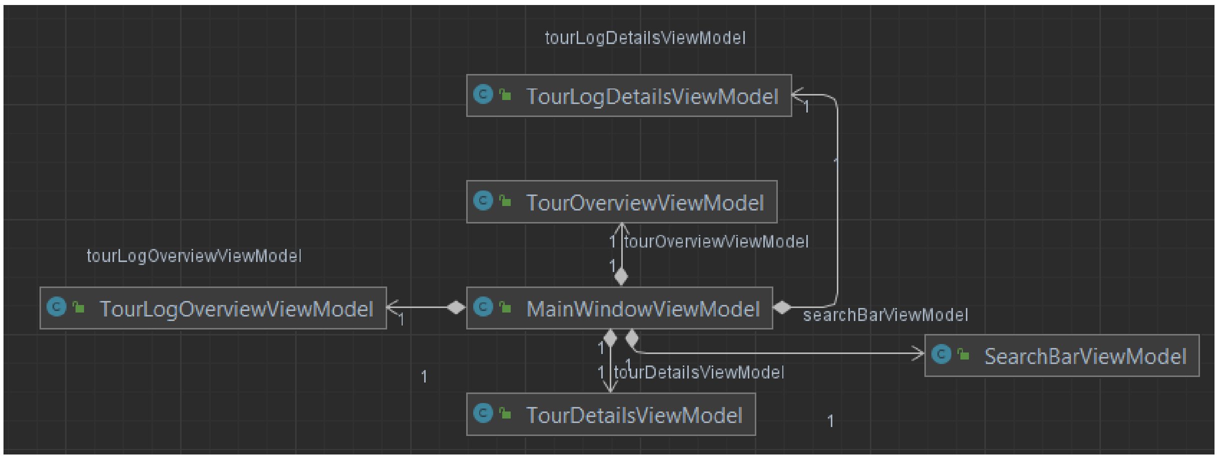
DAL-Classdiagram



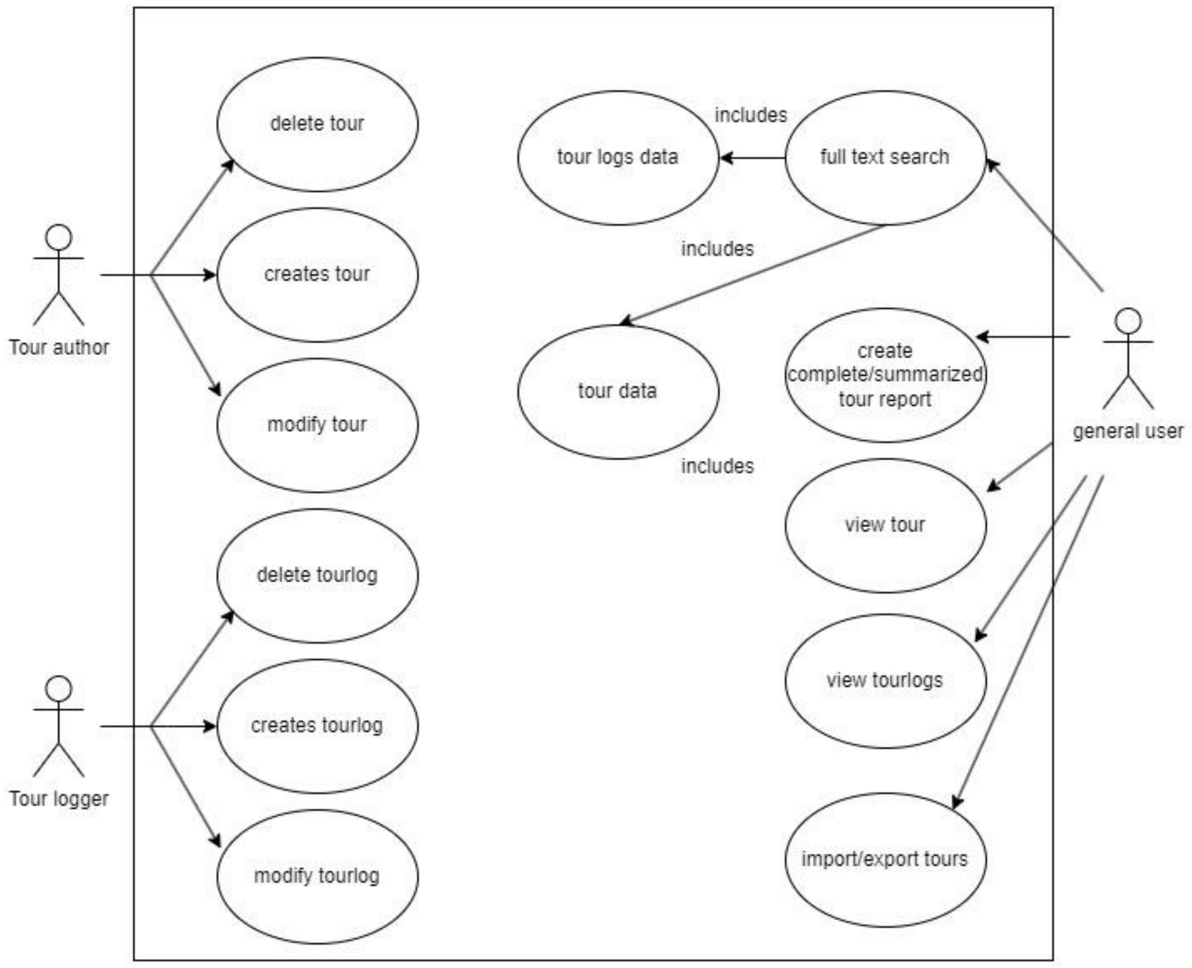
View-Classdiagram



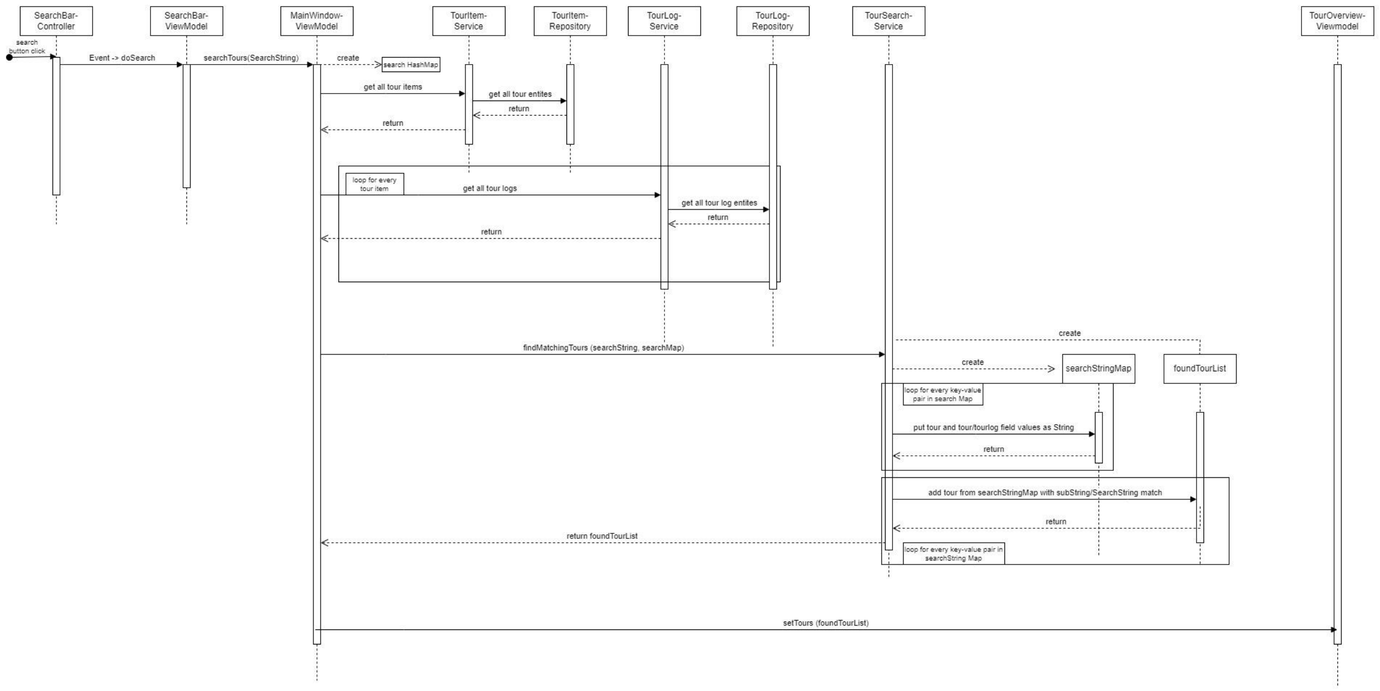
Viewmodel-Classdiagram



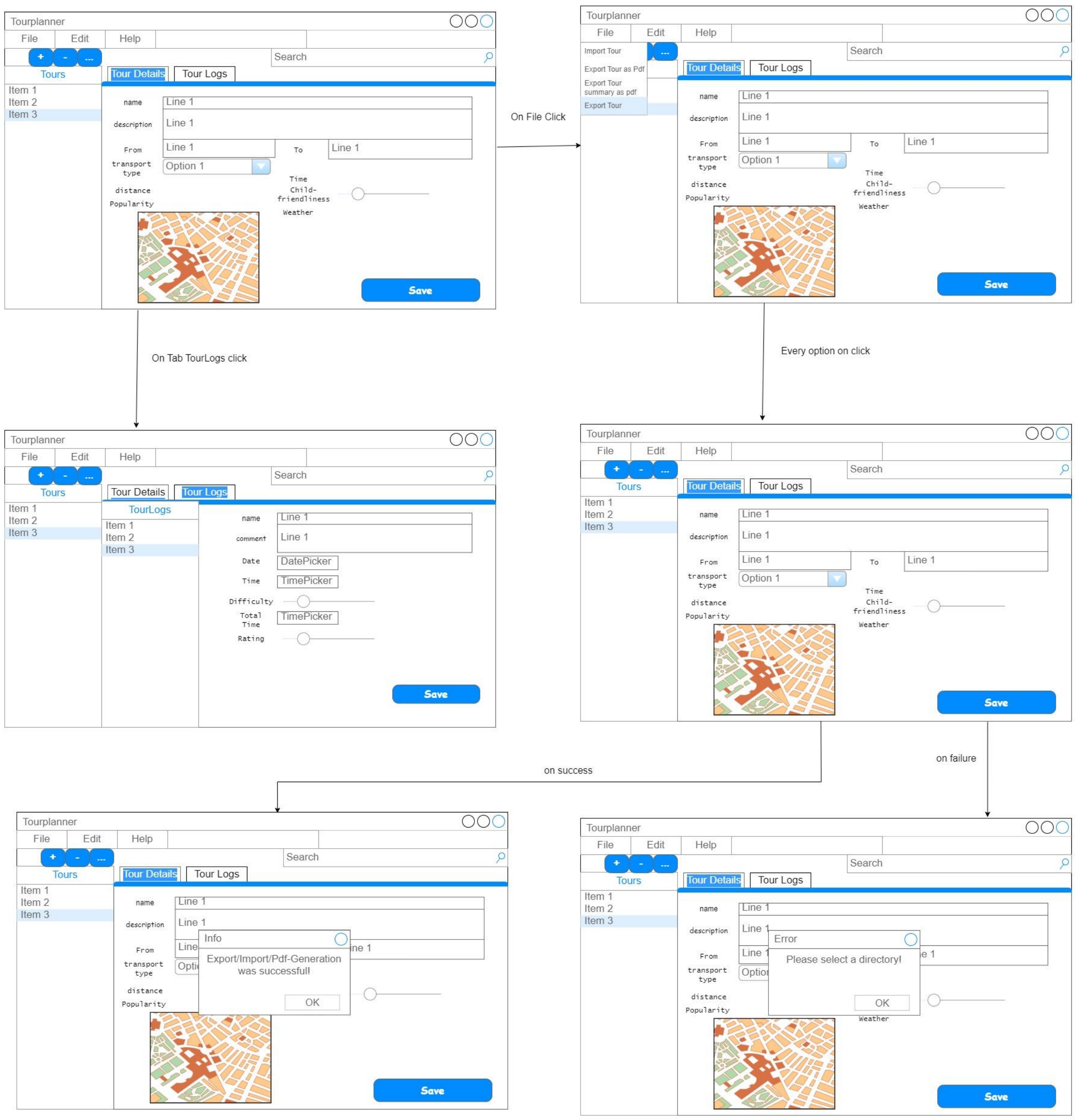
UseCase-Diagram:



Sequence-Diagram:



Wireframe-Diagram:



Chosen Libraries:

JUnit: JUnit is a testing framework that simplifies the creation and execution of unit tests for Java applications. It provides annotations, assertions, and test runners.

Log4j2: Log4j2 is a logging framework that enables developers to log application messages for debugging and analysis. It offers various logging levels, appenders, and layouts to configure the logging behavior, helping in troubleshooting and error analysis.

iText: iText is a library for creating and manipulating PDF documents programmatically. It offers APIs for generating PDF files, modifying existing documents, adding images and text, and performing other PDF-related operations.

Jackson: Jackson is a widely-used JSON processing library for Java. It provides APIs for serializing Java objects to JSON and deserializing JSON to Java objects. Jackson simplifies the conversion of Java objects to JSON format.

Lessons learned:

This was the largest project we had worked on yet. Working on this project provided us with valuable learning experiences, particularly in dealing with new Java dependencies and applying various design patterns.

We recognised the importance of proper project planning. We made sure to plan the project properly and created a detailed plan outlining tasks and goals to ensure efficient progress.

A large focus of this project was on acquiring knowledge of new Java dependencies. To integrate features such as maps, route calculations and Report generation, we had to incorporate external libraries and frameworks. We learned how to effectively utilize these dependencies and integrate them into our code to enhance the app's functionality.

Furthermore, we applied various design patterns. By implementing the MVC (Model-View-Controller) pattern, we were able to separate the app's logic from the user interface and improve code organization. Utilizing different design patterns, made it much easier to maintain and expand the code.

Implemented design patterns:

Consumer/Producer Pattern: The Consumer/Producer pattern establishes a communication mechanism between two entities, where one entity generates data or events, and the other entity consumes or processes that data or events.

Dependency Injection: Dependency Injection is a design pattern that allows the dependencies of a class to be injected from external sources, rather than the class creating or managing its dependencies directly. It promotes loose coupling and improves testability and maintainability.

Interface Segregation: Interface Segregation is a principle that states that clients should not be forced to depend on interfaces they do not use. It promotes the creation of specific and focused interfaces, to avoid unnecessary dependencies and coupling.

Observer Pattern: The Observer Pattern defines a one-to-many relationship between objects, where the subject maintains a list of observers and notifies them automatically of any state changes.

Repository Pattern: The Repository Pattern provides a layer of abstraction between the data access logic and the business logic of an application. It encapsulates the storage, retrieval, and querying of data, providing an interface for data operations.

Chosen unit tests:

The **IOManagerServiceTest** class is responsible for testing the export and import functionalities of the IOManagerService. It verifies that tour items can be successfully exported to a file and imported back with accurate data.

The **MapServiceTest** class is responsible for testing the functionalities of the MapService, including retrieving route information, calculating distances and times, fetching map images, and handling invalid routes.

The **PdfServiceImplTest** class is responsible for testing the functionalities of the PdfServiceImpl. It includes tests for creating tour reports and summaries in PDF format, ensuring that the generated files exist and can be successfully deleted.

The **TourItemTest** class focuses on testing the formatting of estimated time in the TourItem class. It includes tests to verify the correct formatting of estimated time strings for different durations, such as hours only, hours and minutes, and days, hours, and minutes. The tests ensure that the expected time formats are returned accurately.

The **WeatherServiceImplTest** class is responsible for testing the WeatherServiceImpl class's makeApiCall method. It verifies that the method successfully makes an API call to retrieve the current weather forecast for a specified location, in this case, "Vienna." The test asserts that the returned WeatherResponseModel object contains non-null values for the current weather text and temperature, ensuring the successful retrieval of weather data.

The **TourSearchServiceImplTest** class is responsible for testing the TourSearchServiceImpl class's search functionality. It sets up a mock search map containing tour items and tour logs. The test methods evaluate different search scenarios by providing search strings and asserting the expected search results.

The **TourItemRepositoryTest** class and the TourLogRepositoryTest class both test the functionality of their respective repository classes using the DataJpaTest annotation. These tests ensure the proper functionality of the repository classes and validate the basic CRUD operations (create, read, update, delete) for tour items and tour logs.

Unique Feature - Current Weather Display for Selected Routes:

As our unique feature we decided to implement a weather display, enabling users to view the current weather conditions at the start and end destinations when selecting a route. This feature provides textual information about the weather, such as "partly cloudy," along with the corresponding temperature. By integrating weather data into the application, users can obtain real-time weather updates that help them with planning their tours.