**Project Protocol – TourPlanner (Final Hand-in)**

**Layered Architecture:**

The TourPlanner application is designed following a layered architecture, which strictly separates responsibilities across distinct parts of the system. The main layers are:

* *Data Access Layer*
* *Business Logic Layer*
* *Presentation Layer*

**Data Access Layer**

The DAL is responsible for communication with the PostgreSQL database via Entity Framework Core.

* It defines the database schema through the TourDbContext class, which manages tables for Tours and Tourlog entities.
* A 1:n relationship exists between Tours and Tourlogs: one Tour can have many Tourlogs. This is enforced via a foreign key from Tourlog to Tours (configured in the OnModelCreating method of TourDbContext).
* Connection details are read securely from user secrets via a custom ConfigReader class, ensuring no hardcoded secrets in source control.

**Business Logic Layer**

The BLL includes multiple services, among them:

* TourService
* TourLogService
* ApiHandler
* InputValidator

These services encapsulate all business operations, such as:

* CRUD operations on Tours and Tourlogs
* Importing and exporting Tours to JSON files
* Performing fuzzy search across Tours and Tourlogs
* Generating PDF reports using QuestPDF
* Randomly generating Tours with valid start/end points
* Integrating with the OpenRouteService API to compute distances, durations, and routing geometry between points

Business exceptions are defined to enforce the principle "Layers define their own exceptions" (e.g., ApiException, InsertionException, NotFoundException).

**Presentation Layer**

The UI is implemented as a WPF application, following the MVVM pattern.

* Views like MainWindow, TourListView, AddTourView, EditTourView, etc. handle visual components.
* The TourImageView integrates a WebView2 control that displays an interactive Leaflet map with OpenStreetMap tiles.
* Maps dynamically update routes for each Tour, using JavaScript interop with the WebView.
* Reports can embed screenshots of the map for inclusion in PDF reports.

The MainViewModel orchestrates application flow:

* Exposes commands bound to UI actions
* Maintains state like selected tours/logs
* Coordinates loading and saving of data
* Connects to the business layer through Dependency Injection

The UI components are manipulated via several different Sub-ViewModels, all of which can interact with each other. These include:

* SearchViewModel - searches for tours relative to the user’s input
* TourLogViewModel – handles all Tourlog interactions
* TourViewModel – handles all Tour interactions

Views Implemented:

* MainWindow
* TourListBox – Displays all available Tours by name
* TourListView – Shows detailed info for the selected Tour
* TourLogsView – Lists logs belonging to the selected Tour
* AddTourView / EditTourView
* AddTourLogView / EditTourLogView
* TourImageView – Displays map visualization of the Tour

**Important Features**

Several new or enhanced features have been implemented since the intermediate hand-in:

* **Route Geometry & Distance Calculation:**  
  Tours automatically retrieve precise routing geometry and estimated travel times from OpenRouteService, depending on the selected mode of transport (e.g. driving, cycling, walking).
* **Dynamic Map Integration:**  
  The map displays either simple lines between points or precise route polylines retrieved from the API.
* **Import/Export:**  
  Tours can be exported and imported as JSON files, while avoiding serialization issues (e.g. circular references) through the use of DTOs.
* **Report Generation:**  
  Supports:
  + **Single Tour Reports** (includes route map and tour logs)
  + **Summary Reports** (aggregate statistics for all tours, e.g. average duration, distance, and rating)
* **Proper Exception Handling Across Layers:**  
  Each layer raises domain-specific exceptions rather than leaking framework-specific errors.

**Unique Feature**

**Random Tour Generation**

The TourPlanner application includes a unique feature that allows users to generate entirely new, random tours. This feature selects a start and end city from a predefined list of handpicked cities spread across various continents. The system ensures that the chosen cities are not identical and verifies that the distance between them does not exceed the 6000 km limit imposed by the OpenRouteService API.

Once valid cities are selected, the application retrieves the precise route geometry, distance, and estimated duration using the OpenRouteService directions endpoint. The resulting route is saved as a new tour in the database, complete with calculated coordinates, distance, and duration.

This feature demonstrates:

* **Integration with external APIs** for real-time route data
* Practical handling of external service constraints (maximum distance limits)
* A method to create varied, realistic test data without manual input
* An engaging functionality for users to discover new tour possibilities

Overall, random tour generation not only enriches the application’s functionality but also showcases dynamic use of routing services, making TourPlanner more interactive and versatile.

**Unit Tests (XUnit)**

For the TourPlanner project, unit testing was applied selectively to verify core logic, API integrations, and data operations. The following decisions guided the testing approach:

* **Layered Testing Focus**  
  Unit tests were mainly written for the business logic and data access layers. UI testing was not prioritized due to the time constraints and because much of the WPF UI behavior relies on data binding, which is indirectly verified via ViewModel tests.
* **Dependency Injection and Mocks**  
  Many services, including the ApiHandler and data contexts, were injected via dependency injection, making them straightforward to mock. Moq was used to create mocks of:
  + The API handler, to simulate external HTTP responses without consuming real API calls.
  + The database context (via EF Core In-Memory provider) for testing data persistence without touching a real database.
* **In-Memory Database for EF Core**  
  Entity Framework Core’s In-Memory provider was used to test repository and service methods. This allowed for:
  + Testing CRUD operations in isolation.
  + Verifying relations between entities like Tours and TourLogs.
* **Testing DTO Serialization**  
  Tests were written to ensure that import/export of Tours via DTOs correctly mapped all necessary fields and handled scenarios like null references or missing properties.
* **API Integration Tests**  
  A few real calls to OpenRouteService were implemented in integration tests (using real API keys). These tests:
  + Verified that valid requests returned route data with expected structure.
  + Helped debug request formats and API-specific errors like distance limitations.
* **Error Handling Testing**  
  Custom exceptions were tested to ensure meaningful errors propagate upwards. This includes tests for:
  + NotFoundException when searching for missing tours.
  + ApiException handling in API calls.
  + Validation exceptions during import or update operations.
* **Random Tour Generation Testing**  
  Specific unit tests were implemented for the random tour generation feature, verifying that:
  + Generated tours fall within API constraints (max 6000 km).
  + Tour objects have valid randomized properties.
  + Integration with routing APIs returns usable data.
* **Trade-off: No UI Automation**  
  No automated UI tests (e.g. UI automation tools like FlaUI) were implemented, due to time and complexity constraints. Manual testing was performed to validate UI behaviors.

Overall, unit testing provided confidence in the core functionality of TourPlanner, particularly around data consistency, external service calls, and business logic correctness.

**Design Patterns**

The TourPlanner utilizes several important design patterns, which are crucial in its implementation and logic

* **MVVM Pattern:**  
  Used for all UI development in WPF. ViewModels expose bindable properties and commands to Views, ensuring separation of presentation and business logic.
* **Dependency Injection:**  
  Configured via Microsoft’s DI framework, services and configurations are injected instead of tightly coupling dependencies.
* **Data Transfer Object Pattern:**  
  Used for serialization and deserialization of data objects (e.g. Tours and Tourlogs) during import/export.
* **Command Pattern**  
  The Command pattern encapsulates actions as objects. In the TourPlanner, this is visible in the form of RelayCommand, used for UI buttons. Each button click is bound to a command object that executes business logic, keeping the ViewModel and View decoupled and improving testability and reusability.

**Current Limitations & Future Work**

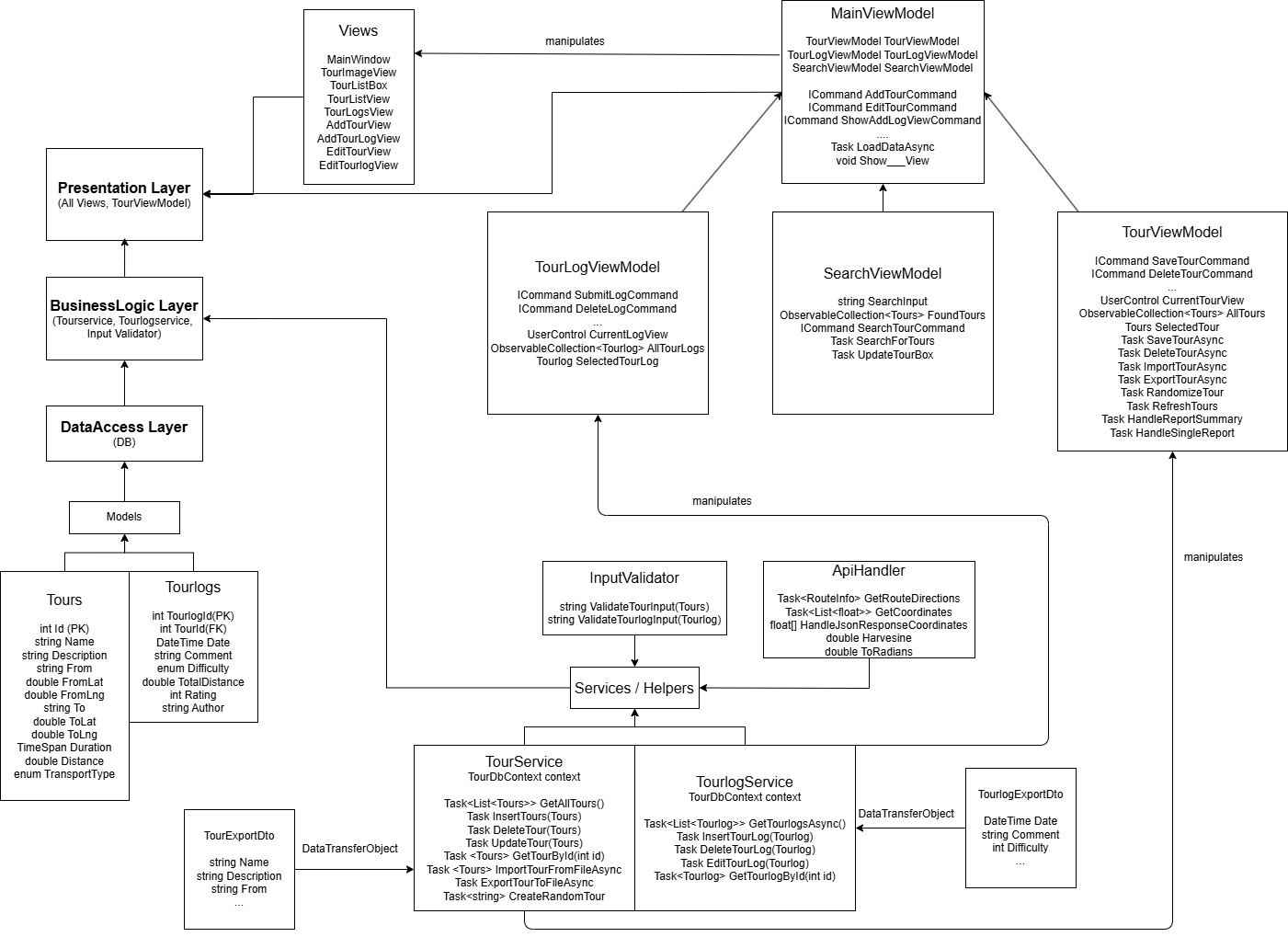
* The user interface can still be further improved for responsiveness.
* Only a subset of transport types is currently supported due to API limitations (e.g. plane/boat routing is not meaningful).
* The random tour generator could be expanded with more parameters for user control.

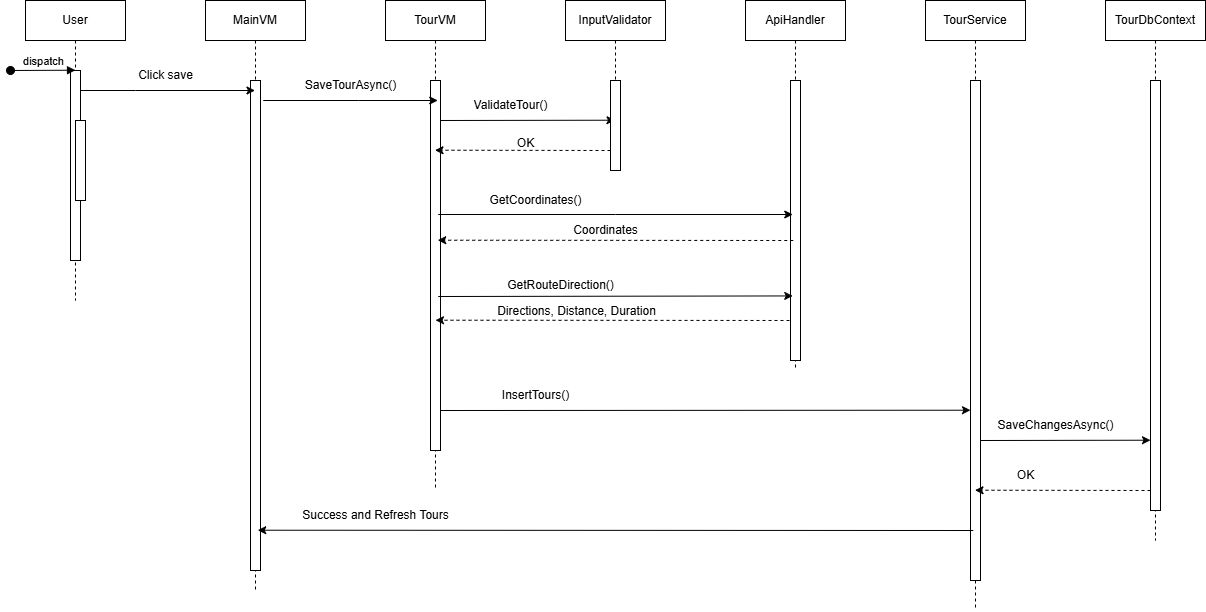
Tracked Time:

| **Date Range** | **Description of Work** | **Hours** |
| --- | --- | --- |
| 2025-03-01 – 2025-03-23 | Initial implementation for intermediate hand-in: core architecture, MVVM structure, basic CRUD for Tours and TourLogs, initial UI setup, database design, entity relationships, simple routing API integration | ~60 h |
| 2025-03-24 – 2025-06-24 | **Pause & minor research**, no major implementation work | ~5 h |
| 2025-06-25 – 2025-07-03 | Start final hand-in work: integration of OpenRouteService for real routing data, map display with Leaflet and WebView2, report generation with QuestPDF, import/export with DTOs, unique random tour generation feature, exception handling, UI improvements, testing | ~55 h |
| 2025-07-04 – 2025-07-06 | Writing protocol, updating diagrams, final bug fixing, preparing hand-in | ~10 h |
| **Total** |  | **~130 h** |

**GIT Link:**

[**https://github.com/LeafOfBread/TourPlanner**](https://github.com/LeafOfBread/TourPlanner)

**Diagrams**

****

****