# Protocol for Tour Planner Project – Rafael & Elias

## 1. Introduction

This document provides a detailed protocol for the Tour Planner application, developed using C# and Windows Presentation Foundation (WPF). The project implements various architectural patterns such as MVVM (Model-View-ViewModel) and follows industry best practices for software development.

## 2. Application Architecture

### 2.1 Architecture Overview

The Tour Planner application is built using the WPF framework, leveraging the MVVM pattern to ensure a clear separation between the user interface and the underlying business logic. The architecture consists of the following layers:  
  
- Models: Represent the data structure of the application, including Log, Tour, and TourLog.  
- ViewModels: Serve as the bridge between the Views and Models, managing the application's state and behavior.  
- Views: XAML files defining the visual interface of the application.  
- Repositories: Encapsulate the logic required to access data sources, implemented in TourRepository.  
- Services: Provide domain-specific logic, such as route management, implemented in RouteService.

### 2.2 Class Diagrams

- Model Classes: Define the structure of Log, Tour, and TourLog, each with properties like Id, Name, Description, Date, etc.  
- ViewModel Classes: Include MainViewModel, AddTourViewModel, AddLogViewModel, etc., each managing the state and logic for corresponding views.  
- Repository and Service Classes: TourRepository handles data access, while RouteService manages business logic related to routes.

## 3. Use Cases

The Tour Planner application supports several core use cases:  
  
- Create Tour: Users can create new tours, specifying details like name, description, and route.  
- Modify Tour: Existing tours can be updated with new information or routes.  
- Delete Tour: Users can remove tours from the planner.  
- Add Log: Logs can be associated with tours to track various details and events.  
- View Logs: Detailed logs can be viewed and managed from the application interface.  
- Validate Input: Ensures that user inputs are valid and prevents application crashes.  
  
Each use case follows the application's flow, from user interaction through the view layer, handled by view models, and ultimately affecting the models and repositories.

## 4. User Experience (UX)

The user interface is designed to be intuitive and user-friendly, following these principles:  
  
- Consistency: Uniform design and behavior across all views and components.  
- Feedback: Immediate visual feedback is provided on user actions, enhancing interaction clarity.  
- Accessibility: The application is designed to be accessible, with considerations for keyboard navigation and screen readers.

### 4.1 User Interface Components

- Main Window: The central hub displaying the list of tours, providing access to create, update, or delete tours and logs.  
- Dialog Windows: Used for adding or editing tours and logs, ensuring focused user interactions.

## 5. Library Decisions and Lessons Learned

- WPF: Chosen for its robust framework and rich feature set for desktop applications, offering strong support for the MVVM pattern.  
- log4net: Used for logging application events and errors, aiding in debugging and monitoring.  
- Entity Framework: Provides object-relational mapping (ORM) capabilities, simplifying data access and manipulation.

### 5.1 OR-Maped Libraries

Entity Framework: Leveraged for its robust ORM capabilities, Entity Framework simplifies database interactions through LINQ queries, change tracking, and automatic database migrations. This choice streamlined data access, reduced boilerplate code, and allowed for easy manipulation of complex data relationships.

### 5.2 Lessons Learned

- Data Binding: Mastering WPF data binding was crucial for a smooth user experience and efficient UI updates.  
- MVVM Implementation: The separation of concerns offered by MVVM greatly improved maintainability and testability.  
- Error Handling: Robust error handling and logging are vital for diagnosing issues and ensuring application stability.  
- ORM Libraries: Using an ORM library like Entity Framework helped streamline database operations and minimized the complexity of SQL queries in the codebase.

## 6. Implemented Design Patterns

- MVVM (Model-View-ViewModel): Facilitates a clean separation between UI and business logic, enhancing maintainability.  
- Repository Pattern: Encapsulates data access logic, promoting a more modular architecture.  
- Service Pattern: Provides a clean separation of domain logic from other concerns, improving code organization.

## 7. Unit Testing Decisions

The application follows a thorough unit testing strategy:  
  
- Test Coverage: Includes tests for ViewModels (MainViewModelTests), services (RouteServiceTests), and repositories (TourRepositoryTests).  
- Testing Framework: Utilizes a standard testing framework like MSTest or NUnit (based on typical C# practices) to automate testing.  
- Continuous Integration: Integrating testing into the development workflow ensures code quality and early detection of issues.

## 8. Tracked Time Breakdown

The project development was estimated to span approximately 160 hours. Below is an estimated breakdown of how the time was allocated:  
  
- Planning and Requirements Gathering: 6 hours  
- Design and Architecture: 26 hours  
- Implementation:  
 - Models and Data Access: 19 hours  
 - ViewModels and Business Logic: 32 hours

- Database: 20 hours   
 - User Interface Development: 25 hours  
- Testing:  
 - Unit Testing Development: 13 hours  
 - Integration and User Testing: 13 hours  
- Documentation and Final Review: 6 hours

## 9. Source Code Repository

The complete source code and documentation for this project are available on GitHub:

[EliasKnr/ROUTEPLANER\_Elias\_Rafael\_Abgabe: SWEN Abgabe (github.com)](https://github.com/EliasKnr/ROUTEPLANER_Elias_Rafael_Abgabe)