Tour Planner

App architecture

The Tour Planner application is designed to help users plan and manage tours. It is built using a layered architecture pattern to ensure separation of concerns, maintainability, and scalability. This README provides an in-depth look at the architecture, layers, and key components of the application.

Architecture

The application is divided into three main layers:

1. Presentation Layer
2. Business Logic Layer (BL)
3. Data Access Layer (DAL)

1. Presentation Layer

Purpose: Manages the user interface and user interactions.

Contents:

- **Views**: XAML files defining the UI layout.
  - `AddTour.xaml`
  - `MainWindow.xaml`
  - Other view files in the `TourPlanner/Views` directory.
- **ViewModels**: Classes managing the data and behavior of the UI components.
  - `AddTourViewModel.cs`
  - `MainWindowViewModel.cs`
  - `TourLogsViewModel.cs`
  - `TourViewModel.cs`
  - Other view model files in the `TourPlanner/ViewModels` directory.

Functionality: Handles UI rendering, binds data to UI components, processes user input, and communicates with the business logic layer via view models.

2. Business Logic Layer (BL)

Purpose: Implements the core functionality and business rules of the application.

Contents:

- **Services**: Classes implementing business operations.
  - `TourService.cs`
  - `RouteService.cs`
  - Other service files in the `Bl` directory.
- **Interfaces**: Defines contracts for the services.
  - `IExportService.cs`
  - `IRouteService.cs`

    - `ITourLogService.cs`
    - `ITourService.cs`

Functionality: Handles the main business processes, such as adding tours, generating routes, and exporting data. It communicates with the data access layer to retrieve and persist data.

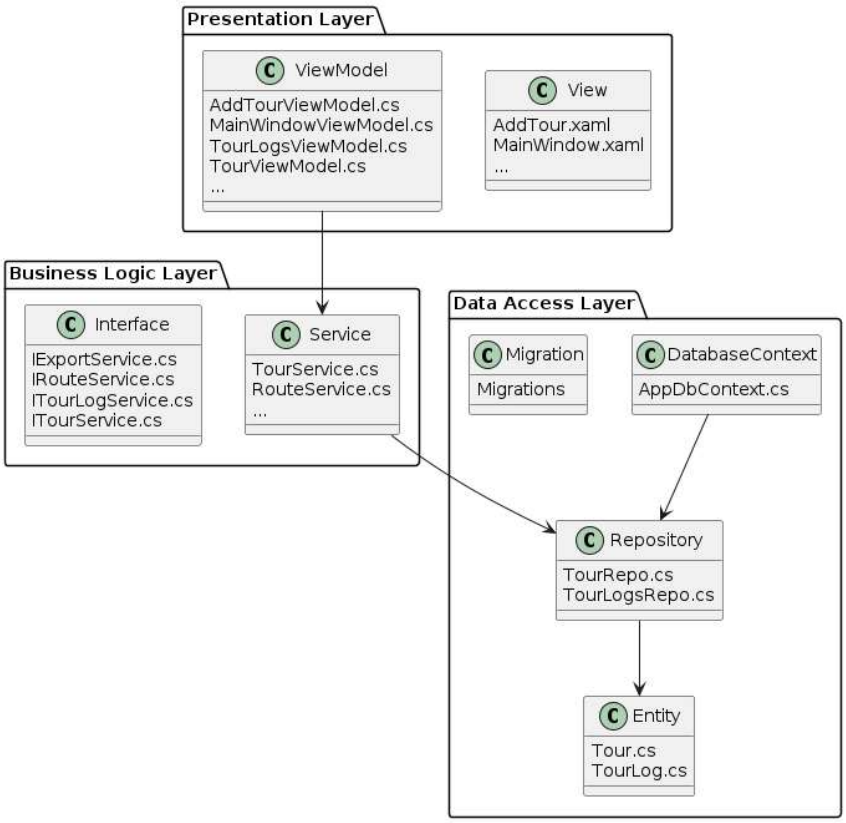
3. Data Access Layer (DAL)

Purpose: Manages data storage and retrieval.

Contents:

- **Database Context**: Configures the Entity Framework database context.
    - `AppDbContext.cs`
- **Repositories**: Classes for accessing data.
    - `TourRepo.cs`
    - `TourLogsRepo.cs`
- **Entities**: Classes representing the data models.
    - `Tour.cs`
    - `TourLog.cs`
- **Migrations**: Handles database schema changes.
    - `Migrations` directory

Functionality: Interacts with the database using Entity Framework, performs CRUD operations, and ensures data integrity.



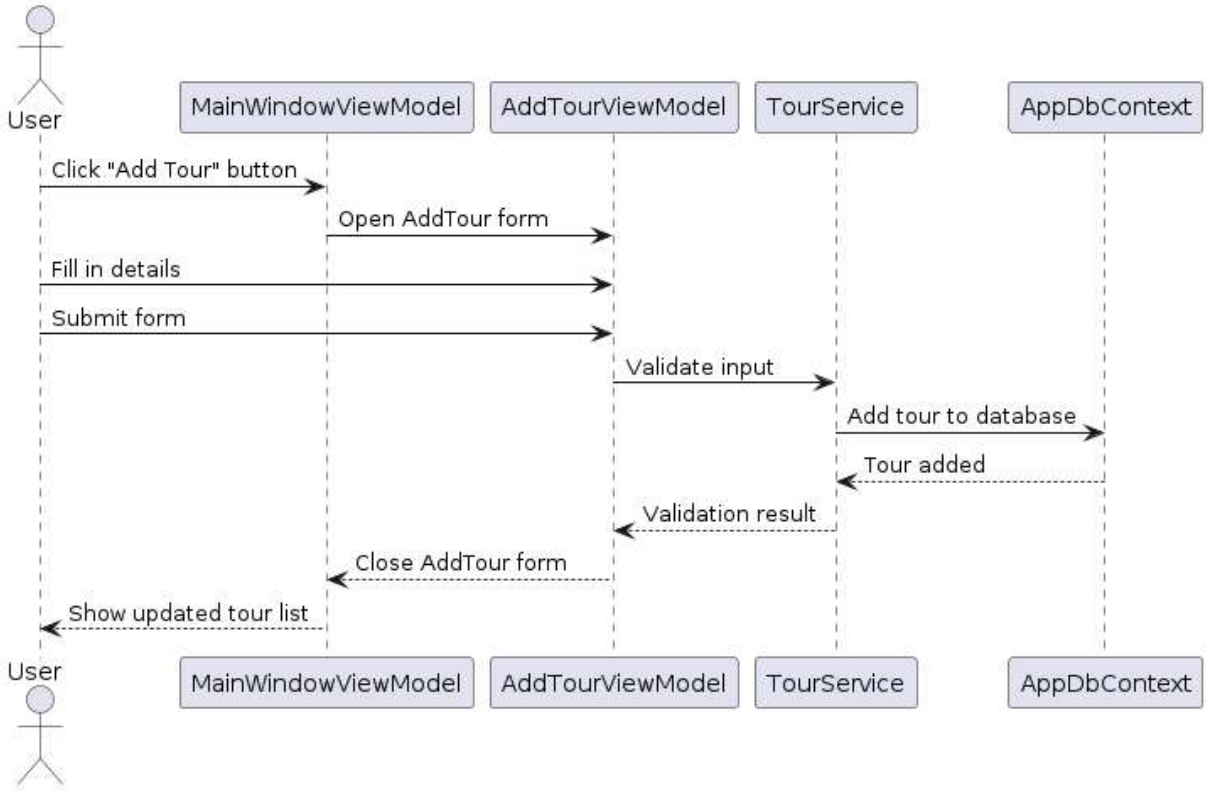
Use cases

1. **Add Tour**
    - **Actor**: User
    - **Description**: The user can add a new tour by providing details such as the tour name, description, start and end addresses, and transport type.
    - **Steps**:
        1. User clicks the "Add Tour" button.
        2. User fills in the required details in the form.
        3. User submits the form.
        4. The system validates the input and adds the tour to the database.
2. **Edit Tour**
    - **Actor**: User
    - **Description**: The user can edit the details of an existing tour.
    - **Steps**:
        1. User selects a tour from the list.
        2. User updates the desired fields in the form.
        3. User submits the form.
        4. The system validates the input and updates the tour in the database.
3. **Delete Tour**
    - **Actor**: User
    - **Description**: The user can delete an existing tour from the list.
    - **Steps**:
        1. User selects a tour from the list.
        2. User clicks the "Delete" button.
        3. The system confirms the deletion.
        4. The tour is removed from the database.
4. **View Tour Logs**
    - **Actor**: User
    - **Description**: The user can view the logs associated with a specific tour.
    - **Steps**:
        1. User selects a tour from the list.
        2. The system displays the detailed logs for the selected tour.
5. **Add Tour Log**
    - **Actor**: User
    - **Description**: The user can add a new log entry for a specific tour.
    - **Steps**:
        1. User selects a tour from the list.
        2. User clicks the "Add Log" button.
        3. User fills in the log details in the form.
        4. User submits the form.
        5. The system validates the input and adds the log entry to the database.
6. **Edit Tour Log**
    - **Actor**: User
    - **Description**: The user can edit an existing log entry for a specific tour.
    - **Steps**:
        1. User selects a tour from the list.
        2. User selects a log entry to edit.
        3. User updates the desired fields in the form.
        4. User submits the form.
        5. The system validates the input and updates the log entry in the database.

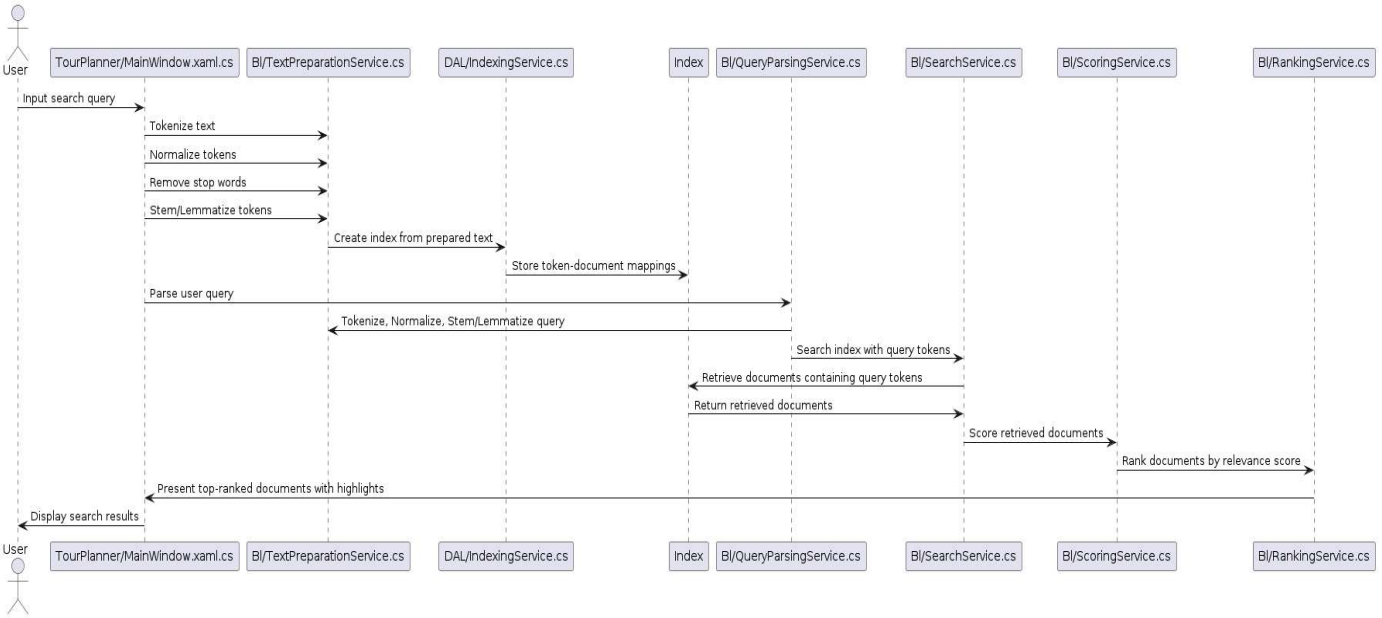
7. **Delete Tour Log**
    - **Actor**: User
    - **Description**: The user can delete an existing log entry for a specific tour.
    - **Steps**:
        1. User selects a tour from the list.
        2. User selects a log entry to delete.
        3. User clicks the "Delete" button.
        4. The system confirms the deletion.
        5. The log entry is removed from the database.
8. **Generate Route**
    - **Actor**: User
    - **Description**: The user can generate a route for a tour by specifying the start and end addresses.
    - **Steps**:
        1. User selects a tour from the list.
        2. User clicks the "Generate Route" button.
        3. User provides the start and end addresses.
        4. The system uses the OpenRouteService API to generate the route.
        5. The system displays the route and saves it in the database.
9. **Export Data**
    - **Actor**: User
    - **Description**: The user can export tour data for backup or sharing purposes.
    - **Steps**:
        1. User clicks the "Export" button.
        2. The system generates an export file containing tour data.
        3. User downloads the export file.

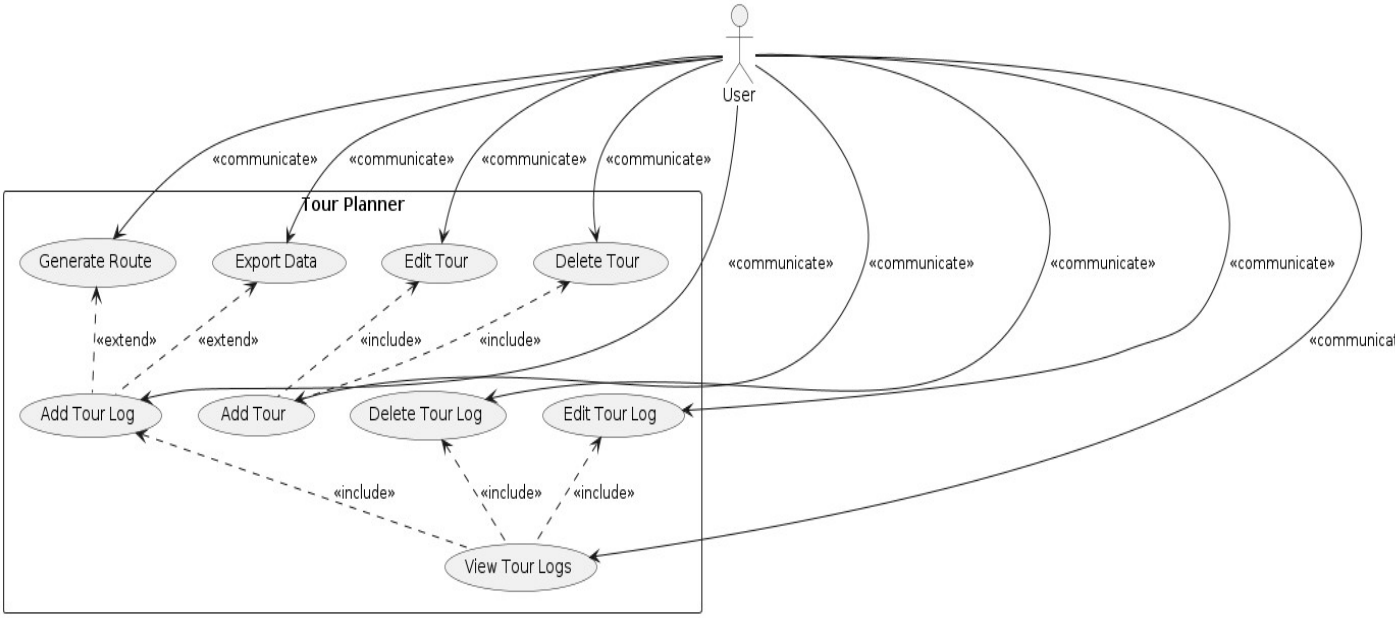
10. **Full-Text Search**
    - **Actor: User**
    - **Description: The user performs a full-text search to find documents containing specific keywords.**
    - **Steps:**
        1. User inputs a search query containing one or more keywords.
        2. The system tokenizes the text of the documents, breaking them into individual words or tokens.
        3. The system normalizes the tokens, converting them to a standard form (e.g., lowercasing, removing punctuation).
        4. The system removes stop words, eliminating commonly used words that do not carry significant meaning.
        5. The system performs stemming or lemmatization, reducing words to their root form (e.g., "running" to "run").
        6. The system creates an index from the prepared text, mapping each token to the documents in which it appears and storing positional information to support phrase searches.
        7. The user's search query is parsed and processed in a similar manner to the text preparation steps.
        8. The system uses the processed query tokens to search the index.
        9. The system retrieves documents that contain the query tokens.
        10. The system scores the retrieved documents based on relevance to the search query using algorithms like Term Frequency-Inverse Document Frequency (TF-IDF).

11. The system ranks the documents by relevance score.
12. The system displays the top-ranked documents to the user, highlighting the query terms within the context of the documents.



Full-Text search





Library decisions

1. **log4net** (Logging)
    - **Purpose**: To provide robust logging capabilities throughout the application.
    - **Decision Rationale**: Log4net is a widely-used, flexible logging library for .NET applications that supports various logging targets (files, databases, consoles, etc.). It allows detailed logging configuration through XML files, making it easy to manage and adjust logging behavior without changing the code.
    - **Lessons Learned**:
        - Proper logging is crucial for debugging and maintaining the application. It helps in tracking errors and understanding the application flow.
        - Configuring log4net using an XML file (e.g., `log4net.config`) makes it easy to change logging levels and targets as needed.
        - Ensure logging is implemented at critical points in the application to capture important events and errors.
2. **Entity Framework Core** (ORM - Object Relational Mapping)
    - **Purpose**: To handle database interactions and manage data access logic.
    - **Decision Rationale**: Entity Framework Core is a powerful ORM for .NET that simplifies database operations by allowing developers to work with database objects using .NET classes. It supports LINQ queries, change tracking, updates, and schema migrations.

- **Lessons Learned**:
  - Using EF Core speeds up development by handling much of the boilerplate code required for database operations.
  - Migrations in EF Core make it easier to evolve the database schema over time without losing data.
  - Properly configuring the DbContext and using dependency injection ensures that database operations are efficient and manageable.

3. **OpenRouteService API** (External Service for Route Generation)
   - **Purpose**: To provide route generation and geocoding services.
   - **Decision Rationale**: OpenRouteService API offers comprehensive routing and geocoding capabilities, which are essential for the Tour Planner application to generate routes based on user inputs.
   - **Lessons Learned**:
     - Integrating external APIs requires handling various types of responses and potential errors gracefully.
     - Rate limits and API key management are important aspects to consider when using external services.
     - Thorough testing with different input scenarios ensures the reliability of the integration.
4. **Newtonsoft.Json** (JSON Handling)
   - **Purpose**: To serialize and deserialize JSON data.
   - **Decision Rationale**: Newtonsoft.Json, also known as Json.NET, is a popular library for JSON processing in .NET. It provides easy-to-use methods for converting .NET objects to JSON and vice versa, and supports advanced features like LINQ to JSON.
   - **Lessons Learned**:
     - The library's ease of use and flexibility make it a great choice for handling JSON data.
     - Proper error handling when deserializing JSON ensures that the application can handle malformed or unexpected data gracefully.
     - Using attributes like `[JsonProperty]` helps in controlling the serialization and deserialization process to match the required JSON structure.
5. **NUnit and Moq** (Unit Testing)
   - **Purpose**: To write and execute unit tests for the application.
   - **Decision Rationale**: NUnit is a widely-used testing framework for .NET, and Moq is a popular mocking library that works well with NUnit. Together, they provide a comprehensive environment for writing and running unit tests.
   - **Lessons Learned**:
     - Writing unit tests helps ensure code quality and reliability by verifying that individual components work as expected.
     - Mocking dependencies using Moq allows for isolated testing of components without relying on their actual implementations.
     - Regularly running unit tests during development helps catch bugs early and maintain a high level of code quality.

By carefully selecting these libraries and learning from their integration and usage, the Tour Planner application benefits from robust logging, efficient database management, reliable routing services, flexible JSON handling, and comprehensive testing capabilities. These decisions contribute to the overall quality, maintainability, and scalability of the application.

Design pattern

The Tour Planner application leverages several design patterns to promote clean, maintainable, and scalable code. Here are the key design patterns used in the application:

1. **Model-View-ViewModel (MVVM)**
    - **Purpose**: To separate the user interface (UI) from the business logic and data, making the application easier to manage and test.
    - **Components**:
        - **Model**: Represents the data and business logic of the application.
            - Example: `Tour`, `TourLog` classes in the `Models` namespace.
        - **View**: Defines the structure and layout of the UI.
            - Example: XAML files such as `MainWindow.xaml`, `AddTour.xaml`.
        - **ViewModel**: Acts as an intermediary between the View and the Model, handling user input, processing data, and updating the View.
            - Example: `MainWindowViewModel`, `AddTourViewModel` in the `ViewModels` namespace.
    - **Benefits**:
        - Enhances the separation of concerns, making the UI independent of the business logic.
        - Improves testability by allowing unit tests to be written for ViewModels.
        - Facilitates easier maintenance and updates to the UI or business logic without affecting each other.
2. **Repository Pattern**
    - **Purpose**: To provide a layer of abstraction over data access logic, making the codebase easier to maintain and test.
    - **Components**:
        - **Repository Interface**: Defines the operations that can be performed on a data source.
            - Example: `ITourRepo`, `ITourLogRepo` in the `DAL` namespace.
        - **Repository Implementation**: Implements the operations defined by the repository interface.
            - Example: `TourRepo`, `TourLogsRepo` in the `DAL` namespace.
    - **Benefits**:
        - Encapsulates data access logic, providing a clean API for data operations.
        - Improves testability by allowing data access logic to be mocked in unit tests.
        - Makes it easier to switch data sources or databases without changing business logic.
3. **Dependency Injection (DI)**
    - **Purpose**: To achieve loose coupling between classes and their dependencies, improving the flexibility and testability of the application.
    - **Components**:

        - **Service Registration**: Registers services and their implementations in a central container.
            - Example: Service registration in `App.xaml.cs`.
        - **Constructor Injection**: Injects dependencies into classes via their constructors.
            - Example: `MainWindowViewModel` constructor receiving `ITourService`, `ITourLogService`, `IRouteService`, and `IExportService`.
    - **Benefits**:
        - Reduces coupling between classes, making the codebase more flexible and easier to modify.
        - Enhances testability by allowing dependencies to be easily replaced with mocks or stubs.
        - Simplifies the management of object lifetimes and dependencies.
4. **Singleton Pattern**
    - **Purpose**: To ensure that a class has only one instance and provides a global point of access to it.
    - **Components**:
        - **Singleton Class**: A class that maintains a single instance throughout the application's lifecycle.
            - Example: The logging configuration setup in `App.xaml.cs` using log4net.
    - **Benefits**:
        - Ensures that a single instance of a class is used throughout the application, which is useful for shared resources like logging.
        - Reduces memory footprint by avoiding the creation of multiple instances.
        - Simplifies access to the instance by providing a global access point.
5. **Factory Pattern**
    - **Purpose**: To create objects without specifying the exact class of object that will be created.
    - **Components**:
        - **Factory Method**: A method that returns an instance of a class, allowing subclasses to alter the type of objects that will be created.
            - Example: The creation of services in the ViewModels, where the ViewModel requests an instance of a service without needing to know the implementation details.
    - **Benefits**:
        - Promotes loose coupling by decoupling object creation from the implementation.
        - Makes it easier to manage and extend the codebase by centralizing object creation logic.
        - Enhances flexibility by allowing new types of objects to be introduced without changing the client code.

These design patterns contribute to the robustness, flexibility, and maintainability of the Tour Planner application, ensuring that it can be easily extended and adapted to future requirements

Describes unit testing decisions

- **Service Layer (Business Logic)**
  - **Tested Components**: `TourService`, `RouteService`
  - **Rationale**:
    - The service layer contains the core business logic of the application. Testing this layer ensures that the main functionalities, such as adding tours, generating routes, and managing tour logs, work as expected.
    - By focusing on the service layer, we ensure that the business rules and workflows are correctly implemented and validated.

- **Repository Layer (Data Access)**
  - **Tested Components**: `TourRepo`, `TourLogsRepo`
  - **Rationale**:
    - The repository layer handles data retrieval and persistence. Testing this layer ensures that data operations are performed correctly, such as saving, updating, deleting, and fetching tours and logs.
    - Validating data access logic helps prevent data integrity issues and ensures the smooth functioning of CRUD operations.

- **ViewModel Layer (Presentation Logic)**
  - **Tested Components**: `MainWindowViewModel`, `AddTourViewModel`
  - **Rationale**:
    - The ViewModel layer acts as an intermediary between the UI and the business logic. Testing this layer ensures that the data binding, command execution, and UI updates are working correctly.
    - By testing ViewModels, we verify that the user interactions are correctly processed and reflected in the application state.

- **External API Interactions**
  - **Tested Components**: Methods in `OpenRouteServiceClient`
  - **Rationale**:
    - Interacting with external APIs, such as the OpenRouteService API for route generation, is a critical functionality of the application. Testing these interactions ensures that API calls are made correctly and responses are handled appropriately.
    - Validating API interactions helps catch potential issues related to network communication, data parsing, and error handling.

- **Edge Cases and Input Validation**
  - **Tested Scenarios**: Invalid inputs, boundary conditions, and error handling
  - **Rationale**:
    - Testing edge cases and input validation ensures that the application can handle unexpected or erroneous inputs gracefully. This includes scenarios such as invalid tour names, incorrect addresses, and API errors.

- Ensuring robust input validation and error handling improves the application's stability and user experience.

Describes unique feature

The "Favorite Tours" feature enables users to mark specific tours as favorites, making them easily distinguishable from other tours. This can be particularly useful for users who manage a large number of tours and want to quickly access their preferred ones.

Tracked time

Period	Hours	Notes
Feb 15-21	7	
Feb 22-28	7	
Mar 1-7	10	
Mar 8-14	12	
Mar 15-21	12	
Mar 22-28	12	
Mar 29-Apr 4	20	
Apr 5-11	20	
Apr 12-18	15	
Apr 19-25	15	
Apr 26-May 2	10	
May 3-9	10	
May 10-16	10	
May 17-23	10	
May 24-30	10	
May 31-Jun 6	10	
Jun 7-15	10	
Total Hours	**200**	

Github

https://github.com/schamori/Tour-planner/blob/main/README.md