TourPlanner – Protocol

Inhaltsverzeichnis

App Architecture

The TourPlanner application is structured using a three-layer architecture:

- Presentation Layer: This layer is responsible for user interaction and interface elements. It includes components like Views and ViewModels. Views handle the rendering of UI elements, while ViewModels manage the data binding and command execution, enabling a responsive and interactive user experience.

- Business Logic Layer: This layer contains the core functionalities and business rules of the application. It consists of Controllers and Services. Controllers manage the application flow by processing user inputs and orchestrating business operations. Services encapsulate the business logic, ensuring the separation of concerns and facilitating easier maintenance.

- Data Access Layer: This layer is responsible for data storage and retrieval. It includes Repositories and Data Models. Repositories provide an abstraction for data access, allowing the business logic to be decoupled from data storage details. Data Models represent the application's data structures, ensuring consistency and integrity.

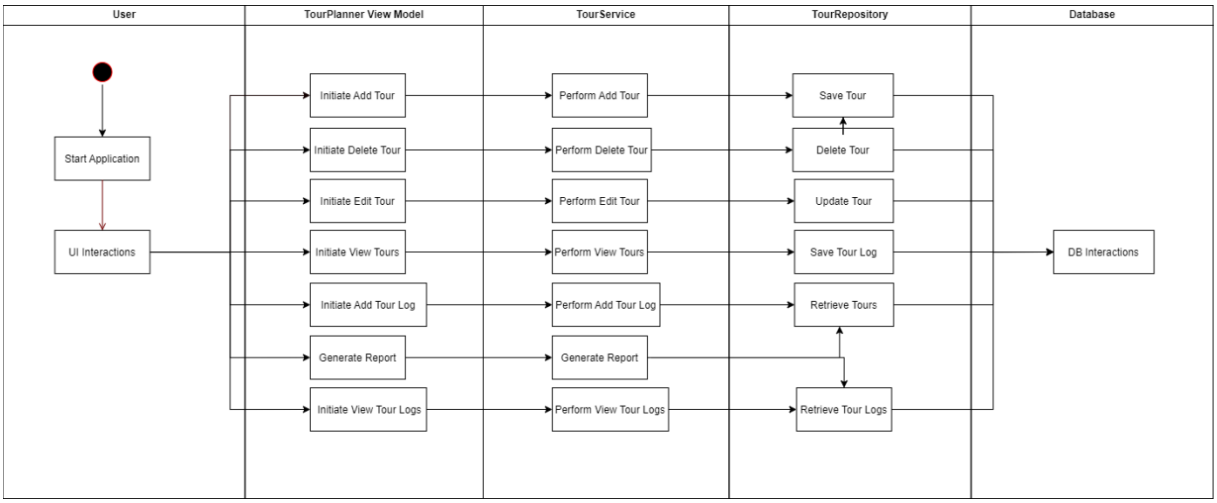
The Models define the data structures used throughout the application, with Tour.cs and TourLog.cs being the primary model classes. Configuration and resource management are handled by the Configuration and Resources components, which include files such as appsettings.json and log4net.config. These files store configuration settings and resources used by different parts of the application. The files in the Resources folder are needed for the embedding of the Leaflet-Map.

Use-Cases

Use-Case-Diagram

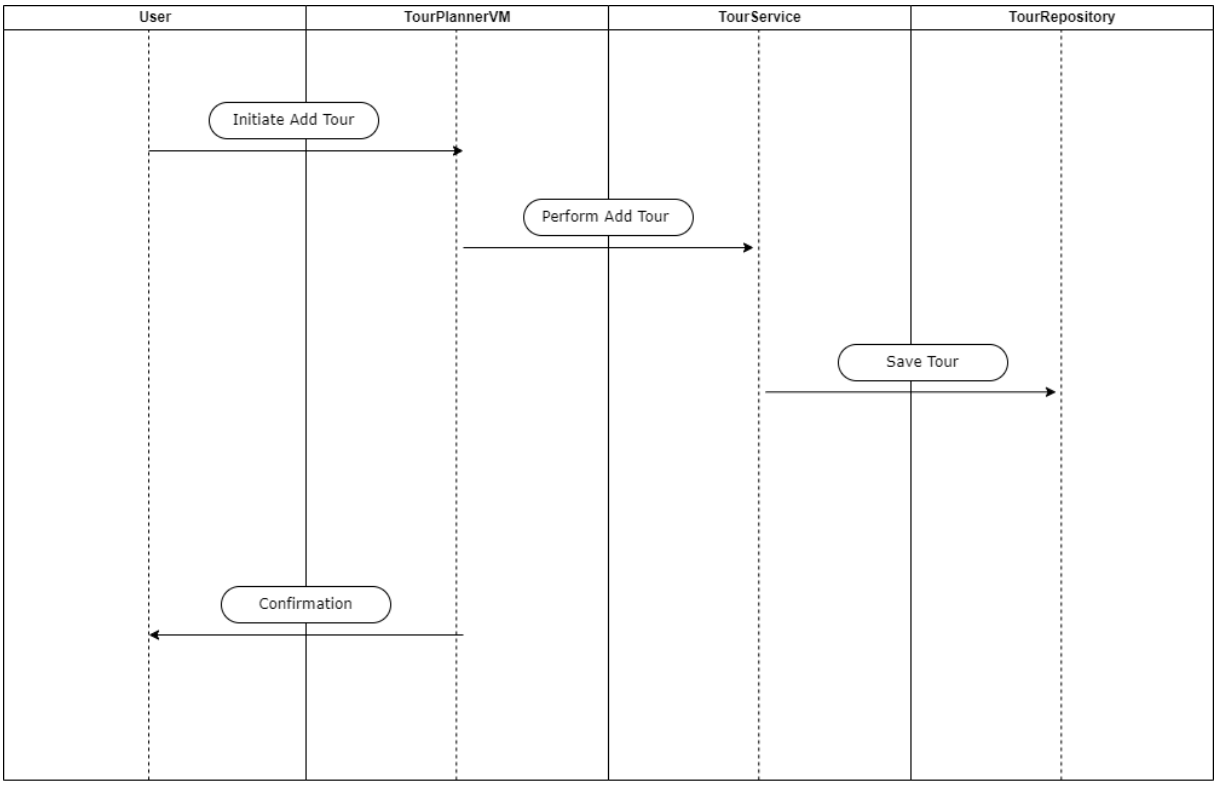
The App supports several use cases. The primary use cases include adding, viewing, editing, and deleting tours and tourlogs, as well as generating reports and comprehend the route between start and end-address via the WebView2-Map.

For instance, the "Add Tour" use case begins when a user initiates the process through the interface. This action triggers a request handled by TourPlannerVM, which then calls the TourService to perform the necessary operations. The TourService interacts with TourRepository to save the new tour to the database, and a confirmation is sent back through the TourPlannerVM to the user interface.

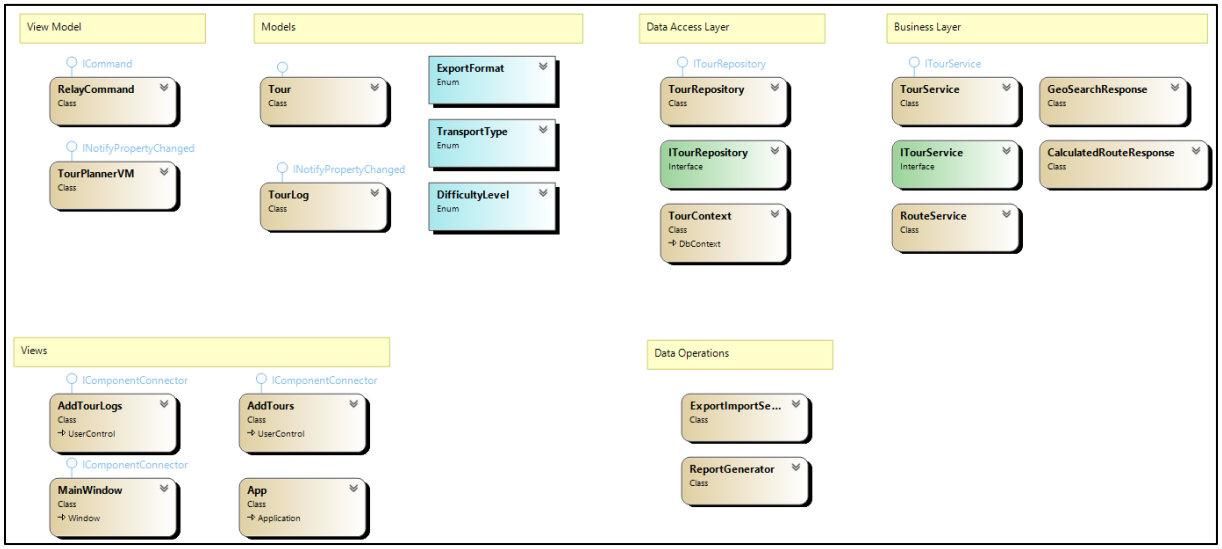


Sequence-Diagram

Sequence diagrams provide a detailed view of the interactions between different components during these use cases. For the "Add Tour" sequence, the diagram illustrates the flow from the user's initial action, through the various service calls, and back to the user with a confirmation of the added tour.



Class-Diagram



UX

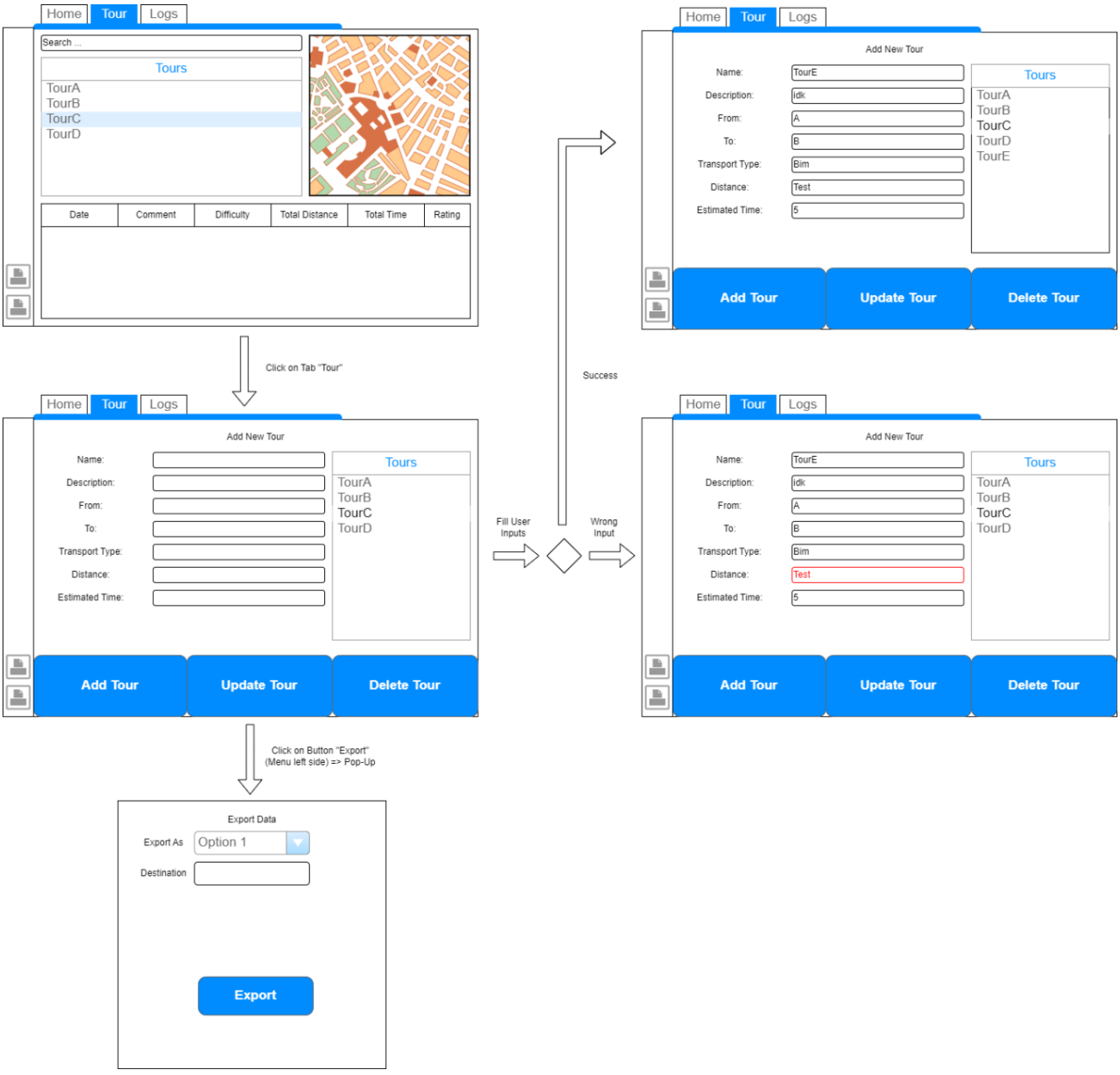
In the Home Tab, there is a navigation bar at the top of the window featuring three sections: Menu, Tours, and TourLogs. On the left side, there is a hamburger menu that can be toggled open or closed by clicking a button (☰). This menu offers functions such as downloading tour reports, exporting and importing tour data, and exporting data as CSV files.

Beneath the navigation bar, there is a search bar where users can input search terms to filter tours. In the middle section of the window, a list of all tours is displayed. Users can search for tours by name, description and all other attributes. Selecting a tour displays details such as distance, estimated time, popularity, and child-friendliness. On the right side, a view is provided using the WebView2 control, offering a visual representation of the tours. At the bottom of the window, there is a section for tour logs. A table displays various log details, including ID, date, comment, difficulty, total distance, total time, and rating.

The Tour Tab is designed for adding new tours. It includes fields for the tour name, description, start point, destination, transport type (with options such as walking, bicycling, and driving), distance, and estimated time. A list on the right side shows existing tours. At the bottom, there are buttons for clearing the input fields, adding new tours, updating existing tours, and deleting tours.

In the TourLogs Tab, users can add logs to existing tours. This view features dropdown menus and text fields for selecting the tour, date and time, comments, difficulty level (easy, medium, hard), total distance, total time, and rating. At the bottom, there are buttons for adding, updating, and deleting tour logs.

The user experience design of TourPlanner is documented with wireframes that outline the application's layout and user interaction flow. These wireframes serve as blueprints for the interface design, ensuring a consistent and intuitive user experience:



Library Decisions

CsvHelper

We used the library CsvHelper to simplify the process of reading from and writing to CSV files, which is essential for handling data import and export within our application.

Extended.Wpf.Toolkit

The Extended.Wpf.Toolkit was utilized to enhance the user interface by providing additional controls and utilities that are not available in the standard WPF toolkit, such as a Color Picker and Calculator.

iTextSharp

We incorporated iTextSharp to handle PDF document creation and manipulation, allowing our application to generate, modify, and manage PDF files efficiently.

log4net

For logging purposes, we relied on log4net to facilitate logging of information, errors, and debugging messages to various outputs, thereby improving the monitoring and troubleshooting processes.

Microsoft.AspNetCore.Hosting

Microsoft.AspNetCore.Hosting was used to provide hosting capabilities for web applications, which is useful in scenarios where our WPF application needs to integrate or host web services.

Microsoft.AspNetCore.Http.Abstractions

We used Microsoft.AspNetCore.Http.Abstractions to simplify the handling of HTTP requests and responses, making it easier to manage web-based communication within our application.

Microsoft.EntityFrameworkCore

To handle database interactions, we utilized Microsoft.EntityFrameworkCore, an Object-Relational Mapper (ORM) that allows us to query and manipulate data using .NET objects.

Microsoft.Extensions.Configuration

The Microsoft.Extensions.Configuration library was employed to manage application settings and configuration data from various sources, ensuring a flexible and maintainable configuration system.

Microsoft.Extensions.Configuration.Binder

We used Microsoft.Extensions.Configuration.Binder to bind configuration settings to strongly-typed objects, facilitating easier and more reliable configuration management.

Microsoft.Extensions.Configuration.Json

Microsoft.Extensions.Configuration.Json was chosen to read configuration settings from JSON files, providing a straightforward and widely-used format for configuration data.

Microsoft.Extensions.DependencyInjection

We incorporated Microsoft.Extensions.DependencyInjection to manage service lifetimes and dependencies within our application, promoting loose coupling and enhancing testability

Microsoft.Extensions.Hosting

The Microsoft.Extensions.Hosting library was used to handle application lifetime management, offering features for starting and stopping services seamlessly.

Microsoft.Web.WebView2

We used Microsoft.Web.WebView2 to embed web content within our WPF application, leveraging the capabilities of Microsoft Edge (Chromium) to display HTML and JavaScript content.

MSTest.TestAdapter

To integrate unit testing, we used MSTest.TestAdapter, which works with Visual Studio Test Explorer to discover and execute tests written using the MSTest framework.

MSTest.TestFramework

MSTest.TestFramework provided the necessary tools and attributes for writing unit tests in .NET, helping us ensure the reliability and correctness of our code.

Newtonsoft.Json

For JSON serialization and deserialization, we relied on Newtonsoft.Json to handle the conversion between .NET objects and JSON, which is essential for data interchange.

Npgsql.EntityFrameworkCore.PostgreSQL

Lastly, we used Npgsql.EntityFrameworkCore.PostgreSQL to enable Entity Framework Core to interact with PostgreSQL databases, facilitating CRUD operations and efficient database management within our application.

Lessons Learned

Kastl: Same as in the last SWEN-project, it might have helped if I started earlier with the actual implementation of the code. My partner Flo had more discipline and started before me, but I guess this was the trigger for me to help him. But in that case, at least I had “fun” tryharding over 10 hours per day for the last 5 days with another SWEN-group on discord (for moral support). Wouldn’t have wanted it any other way. I can also say that I regret working with OpenStreetMap-Tilemaps; I don’t know how much time I wasted on them – without even using them in the end. The basic knowledge about WPF from the HTL was also very useful for this project.

Poppinger: Initially, we had strong motivation for the project, which was positively influenced by the intermediate hand-in. However, after taking a long pause, it was challenging to get back into the programming mindset. By the end, things became confusing, especially with the One VM class. I wanted to split it up but couldn’t figure out how to do it.

Another issue was the misleading errors from WPF. This was particularly frustrating because I spent a lot of time trying to solve them, only to realize that all it needed was to rebuild the application.

Implemented Design Pattern

We used the **Repository Pattern** to manage data access logic within our application. By abstracting data storage and retrieval, the Repository Pattern allows us to decouple the data layer from the business logic layer, ensuring a clear separation of concerns. This makes the codebase more maintainable, testable, and scalable. It provides a consistent way to interact with data sources, whether they are databases, web services, or other data stores, allowing us to switch or modify data sources with minimal changes to the business logic.

Additionally, we employed the **MVVM Pattern**, which is a design pattern commonly used in WPF applications. MVVM helps to separate the user interface (View) from the business logic and data (Model) through an intermediate component (ViewModel). This separation facilitates a cleaner architecture, enabling developers to work on the user interface and business logic independently. MVVM enhances testability by allowing the ViewModel to be tested without requiring the actual UI, and it improves maintainability by clearly defining the responsibilities of each component. This pattern also supports data binding, making it easier to synchronize the UI with the underlying data model.

Unit-Testing Decisions

The TourPlannerVMTests class focuses on testing the functionalities of the TourPlannerVM ViewModel in the Tour Planner application. The AddTour method verifies that adding a new tour increases the count of tours in the ViewModel, ensuring that tours are correctly added. The UpdateTour method tests the functionality of updating an existing tour by checking if the ViewModel updates the tour details correctly. The DeleteTour method ensures that a tour can be successfully removed from the ViewModel. Similarly, the AddTourLog and UpdateTourLog methods test the addition and updating of tour logs within a selected tour. Finally, the DeleteTourLog method confirms that a tour log can be correctly removed from a tour's log list.

The ICommandTests class is dedicated to testing the commands in the TourPlannerVM ViewModel, which implement the ICommand interface. The AddTourCommand_CanExecute method tests whether the AddTourCommand can execute given valid input, ensuring the command is enabled only when necessary inputs are provided. The AddTourCommand_Execute method confirms that executing the command adds a new tour to the ViewModel. The UpdateTourCommand_CanExecute and UpdateTourCommand_Execute methods verify that the update command is enabled when a tour is selected and correctly updates the selected tour.

The DeleteTourCommand_CanExecute and DeleteTourCommand_Execute methods ensure that the delete command is enabled when a tour is selected and successfully removes the selected tour. Additionally, the AddTourLogCommand and DeleteTourLogCommand methods test the execution and enabling of commands related to tour logs, ensuring logs can be added and removed as expected. The SaveTourLogCommand methods confirm that tour logs can be updated correctly.

The ServiceTests class evaluates the functionality of the TourService class, which handles business logic and data interactions. The AddTour method tests that a tour can be added to the database through the service, ensuring data is persisted correctly. The UpdateTour method verifies that updates to tour details are correctly reflected in the database. The DeleteTour method ensures that a tour can be removed from the database using the service. Finally, the GetAllTours method tests that the service retrieves all tours from the database, confirming that the service can correctly access and return stored data. These tests collectively ensure that the TourService class functions as intended, providing reliable data operations for the application.

Unique Feature

For our unique feature, we chose to implement more ways of exporting our Tour-data in the formats CSV, XML and JSON:

CSV Exporter: This method constructs a CSV file by organizing tour details into rows and columns. The first line specifies column headers such as "Name," "Description," "From," "To," "Distance," "Estimated Time," "Popularity," and "Child Friendliness," separated by semicolons. It then iterates through each tour in the collection, appending the relevant data to each line. The file is saved with a timestamp in the user's Documents folder for convenient access.

XML Exporter: With this exporter, tour data is serialized into XML format, facilitating easy storage and transfer. The method employs XmlSerializer to convert the list of tours into structured XML data. Once serialized, the XML content is saved to a file in the user's Documents folder, named with a timestamp for clarity.

JSON Exporter: Utilizing the capabilities of the Newtonsoft.Json library, this exporter converts tour data into JSON format. The method serializes the tour list, ensuring that complex data structures are accurately represented. The resulting JSON content is written to a file in the user's Documents folder, incorporating a timestamp to differentiate files.

Tracked Time

Kastl Tobias

Date	Time	Description
< 30.05.2024	?h	Tasks for Intermediate-Submission
30.05.2024	11h	OpenRouteservice (Select Tour, Refresh-Btn), Report Generator Update, WebView2 for showing route, more Seeding-Data
31.05.2024	12h	Started with Protocol, Input-Validation-Fix, Report-Generator-Formatting, Distance & Estimated-Time Fix (UTC formatting), Text-Search-Filter
01.06.2024	10h	Create UnitTests, Create Unique-Feature, Computed Values added, Code-Cleanup, Protocol
02.06.2024	4h	Protocol-editing, Readme-Guide, Submission-rdy

Poppinger Florian

Date	Time	Description
< 25.05.2024	?h	Tasks for Intermediate-Submission
25.05.2024	11	DB Connection, Postgres Container, Layers, DAL, BL
28.05.2024	6	DB Functions
29.05.2024	8	Log4net, Export Import Service
31.05.2024	3	String to Enum Changes, Code Fixes
01.06.2024	6	Bug Fixing, Code Cleaning, Protocol
02.06.2024	2	Protocol-editing, Readme-Guide, Submission-rdy

Link to GIT

https://github.com/FloberPoP/Swen2-Tour_Planner.git