SWEN_TourPlanner

von Velid Heldic und Stefan Werner

https://github.com/RxndyOG/SWEN_TourPlanner

Description:

Tour Planner is an application based on the GUI frameworks C# WPF. The user creates (bike-, hike-, running- or vacation-) tours in advance and manages the logs and statistical data of accomplished tours.

Design

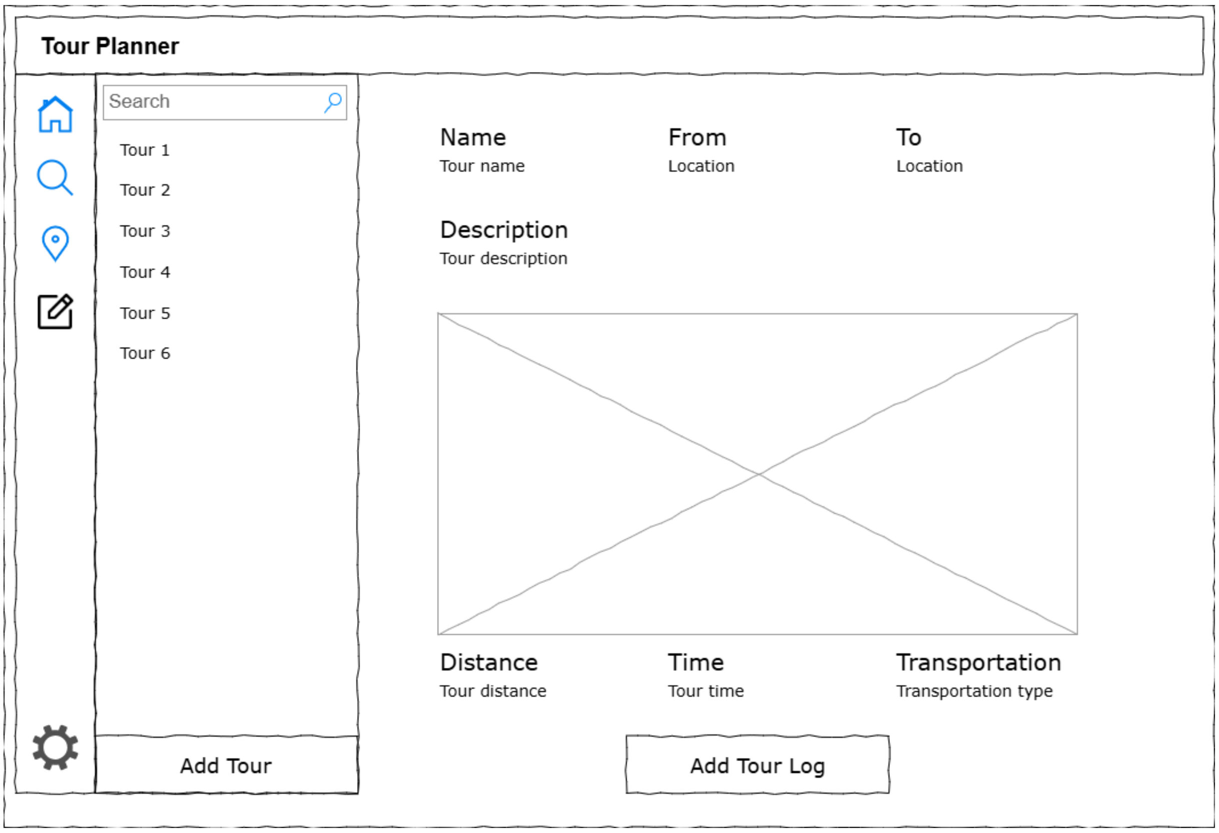
UX (Wireframe)

We divide our application into three sections side-by-side. The left section is the smallest and is responsible for the navigation of the application using dashboard icons. Based on the currently selected icon the content of the other parts changes.

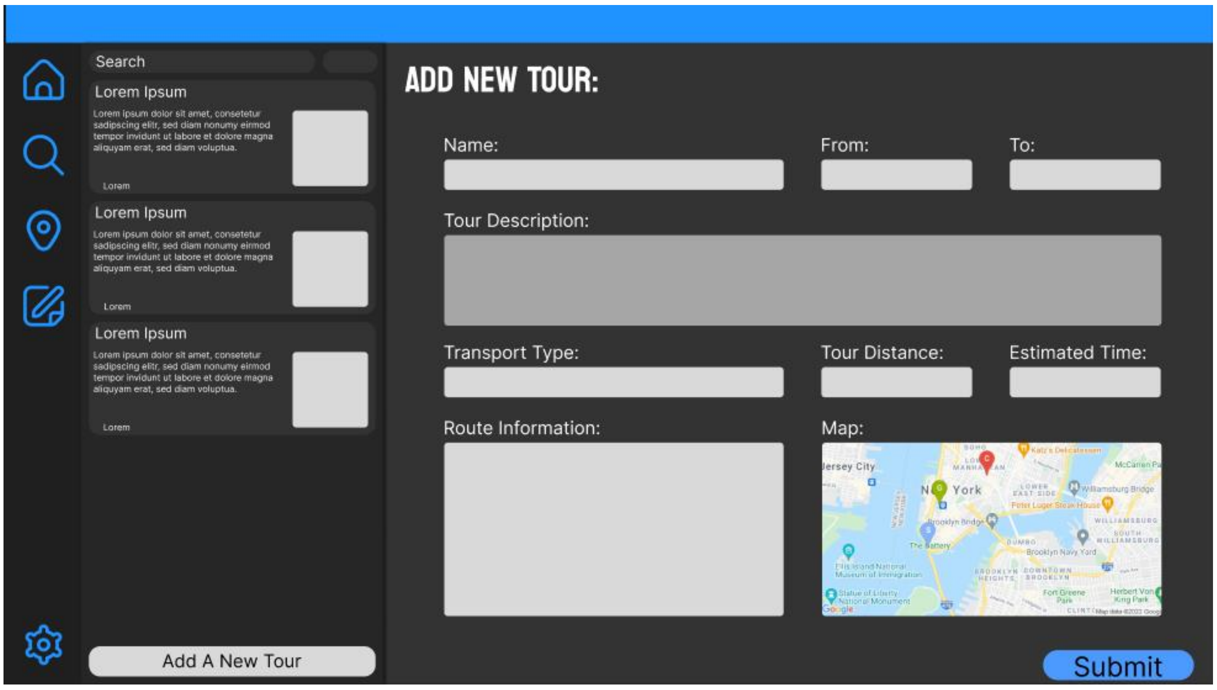
The middle section displays a list of tours the user can choose from. The actual content of the tour and the tour logs are shown in a grid on the right and biggest section. The base stays the same, only the content changes based on the selected tour.

To create a tour or a tour log for a tour, the user can click the specific button to open a form. After submitting the filled in form, the new tour will be listed with the others and the tourlog will be shown in the tour.

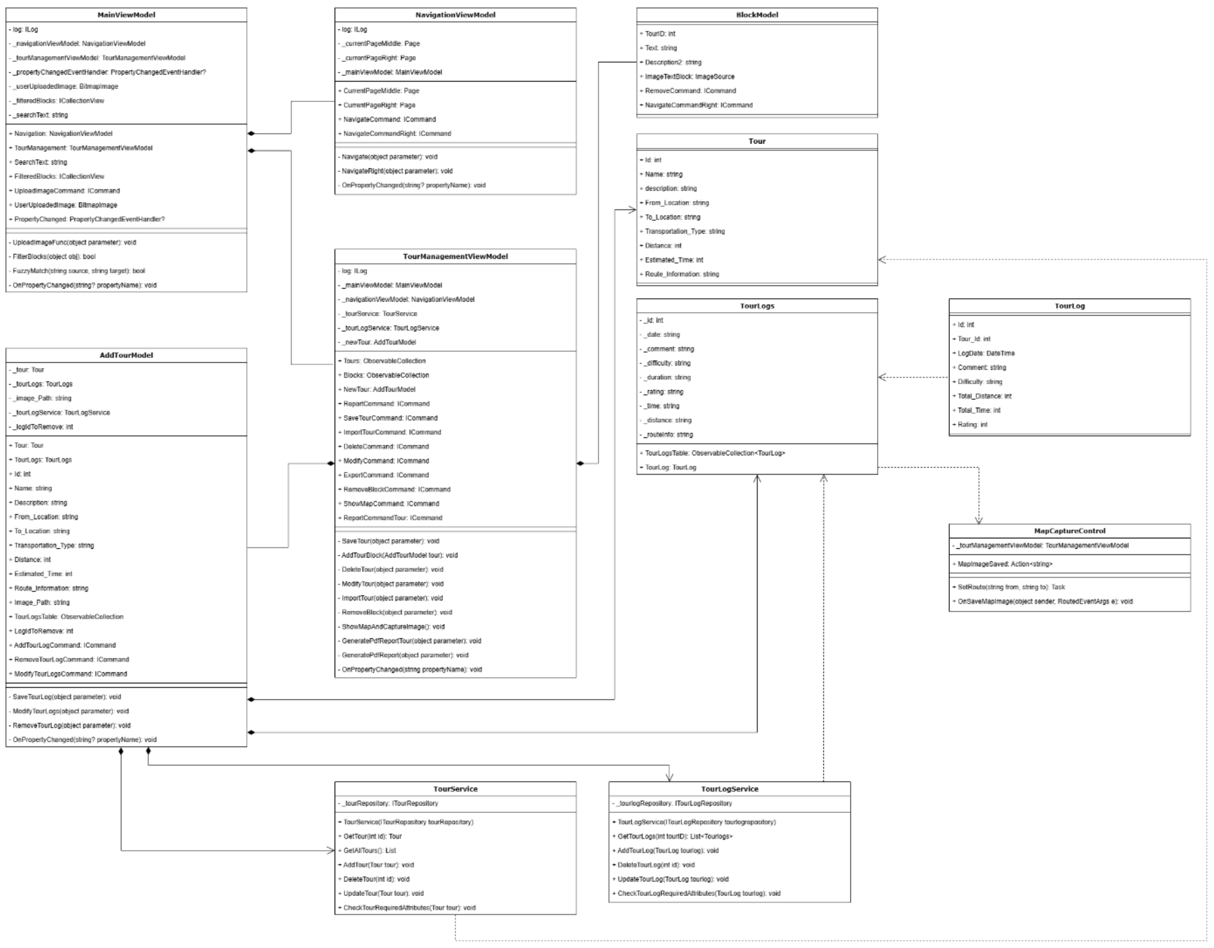
Mockup



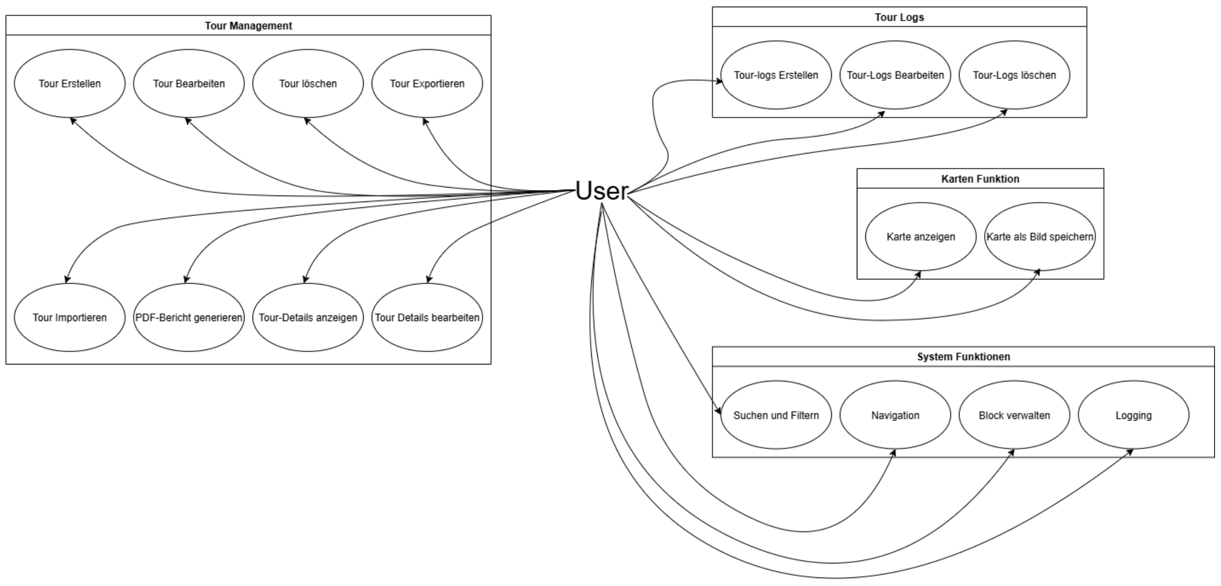
Implementation


Search
Lorem Ipsum
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
Lorem
Lorem Ipsum
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
Lorem
Lorem Ipsum
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
Lorem
Add A New Tour
ADD NEW TOUR:
Name:
From:
To:
Tour Description:
Transport Type:
Tour Distance:
Estimated Time:
Route Information:
Map:
Submit

Class-Diagram


MainViewModel
- log: ILog
- _navigationViewModel: NavigationViewModel
- _tourManagementViewModel: TourManagementViewModel
- _propertyChangedEventHandler: PropertyChangedEventHandler?
- _userUploadedImage: BitmapImage
- _filteredBlocks: ICollectionView
- _searchText: string
+ Navigation: NavigationViewModel
+ TourManagement: TourManagementViewModel
+ SearchText: string
+ FilteredBlocks: ICollectionView
+ UploadImageCommand: ICommand
+ UserUploadedImage: BitmapImage
+ PropertyChanged: PropertyChangedEventHandler?
- UploadImageFunc(object parameter): void
- FilterBlocks(object obj): bool
- FuzzyMatch(string source, string target): bool
- OnPropertyChanged(string? propertyName): void
NavigationViewModel
- log: ILog
- _currentPageMiddle: Page
- _currentPageRight: Page
- _mainViewModel: MainViewModel
+ CurrentPageMiddle: Page
+ CurrentPageRight: Page
+ NavigateCommand: ICommand
+ NavigateCommandRight: ICommand
- Navigate(object parameter): void
- NavigateRight(object parameter): void
- OnPropertyChanged(string? propertyName): void
BlockModel
+ TourID: int
+ Text: string
+ Description2: string
+ ImageTextBlock: ImageSource
+ RemoveCommand: ICommand
+ NavigateCommandRight: ICommand
Tour
+ Id: int
+ Name: string
+ description: string
+ From_Location: string
+ To_Location: string
+ Transportation_Type: string
+ Distance: int
+ Estimated_Time: int
+ Route_Information: string
TourManagementViewModel
- log: ILog
- _mainViewModel: MainViewModel
- _navigationViewModel: NavigationViewModel
- _tourService: TourService
- _tourLogService: TourLogService
- _newTour: AddTourModel
+ Tours: ObservableCollection
+ Blocks: ObservableCollection
+ NewTour: AddTourModel
+ ReportCommand: ICommand
+ SaveTourCommand: ICommand
+ ImportTourCommand: ICommand
+ DeleteCommand: ICommand
+ ModifyCommand: ICommand
+ ExportCommand: ICommand
+ RemoveBlockCommand: ICommand
+ ShowMapCommand: ICommand
+ ReportCommandTour: ICommand
- SaveTour(object parameter): void
- AddTourBlock(AddTourModel tour): void
- DeleteTour(object parameter): void
- ModifyTour(object parameter): void
- ImportTour(object parameter): void
- RemoveBlock(object parameter): void
- ShowMapAndCaptureImage(): void
- GeneratePdfReportTour(object parameter): void
- GeneratePdfReport(object parameter): void
- OnPropertyChanged(string propertyName): void
TourLogs
- _id: int
- _date: string
- _comment: string
- _difficulty: string
- _duration: string
- _rating: string
- _time: string
- _distance: string
- _routeInfo: string
+ TourLogsTable: ObservableCollection<TourLog>
+ TourLog: TourLog
TourLog
+ Id: int
+ Tour_Id: int
+ LogDate: DateTime
+ Comment: string
+ Difficulty: string
+ Total_Distance: int
+ Total_Time: int
+ Rating: int
MapCaptureControl
- _tourManagementViewModel: TourManagementViewModel
+ MapImageSaved: Action<string>
+ SetRoute(string from, string to): Task
+ OnSaveMapImage(object sender, RoutedEventArgs e): void
AddTourModel
- _tour: Tour
- _tourLogs: TourLogs
- _image_Path: string
- _tourLogService: TourLogService
- _logIdToRemove: int
+ Tour: Tour
+ TourLogs: TourLogs
+ Id: int
+ Name: string
+ Description: string
+ From_Location: string
+ To_Location: string
+ Transportation_Type: string
+ Distance: int
+ Estimated_Time: int
+ Route_Information: string
+ Image_Path: string
+ TourLogsTable: ObservableCollection
+ LogIdToRemove: int
+ AddTourLogCommand: ICommand
+ RemoveTourLogCommand: ICommand
+ ModifyTourLogsCommand: ICommand
- SaveTourLog(object parameter): void
- ModifyTourLogs(object parameter): void
- RemoveTourLog(object parameter): void
- OnPropertyChanged(string? propertyName): void
TourService
- _tourRepository: ITourRepository
+ TourService(ITourRepository tourRepository)
+ GetTour(int id): Tour
+ GetAllTours(): List
+ AddTour(Tour tour): void
+ DeleteTour(int id): void
+ UpdateTour(Tour tour): void
+ CheckTourRequiredAttributes(Tour tour): void
TourLogService
- _tourlogRepository: ITourLogRepository
+ TourLogService(ITourLogRepository tourlogrepository)
+ GetTourLogs(int tourID): List<Tourlogs>
+ AddTourLog(TourLog tourlog): void
+ DeleteTourLog(int id): void
+ UpdateTourLog(TourLog tourlog): void
+ CheckTourLogRequiredAttributes(TourLog tourlog): void

Use-Case-Diagram



App architecture

UI Layer

The UI Layer is the frontend of the application and is responsible for the Design. It provides an interactive Programm and handles the input by forwarding it to the Business Layer

Business Layer

The Business Layer acts in between the UI Layer and the DataAccess Layer. It validates the data and hands it to the DataAccess Layer for database related tasks.

DataAccess Layer

The DataAccess Layer is responsible for the database connection and the manipulation of data. It is done using an Object-Relational-Mapper (ORM) to map the objects in the way the database needs it and is devided in the CRUD operations.

Model

The Model defines the data structures that are used throughout the application. It includes classes representing the entities (e.g., Tour, TourLog) and their properties. The model is used by all three layers and ensures consistency in the datatype being used.

Implementation

Design patterns

Repository Pattern

The repository pattern helps us separate the data access logic from the rest of the application. It handles all the details of how data is fetched or saved

In our DataAccess Layer we define Interfaces (`ITourRepository`, `ITourLogRepository`) for the repositories and require them to implement the CRUD Operations. The repositories (`TourRepository`, `TourLogRepository`) than implement the interfaces and use an ORM-Mapper to work with the database

Command Pattern

The project leverages the Command Pattern to encapsulate actions as objects, enabling a clean separation between the UI and business logic. This design pattern is particularly useful in WPF applications, where commands can be bound directly to UI elements, such as buttons, simplifying event handling and improving code maintainability.

Implementation Details:

- **RelayCommand**: A reusable implementation of the `**ICommand**` interface, used extensively across **ViewModels** like `**TourManagementViewModel**` and `**NavigationViewModel**`. It allows the execution of methods and the evaluation of whether a command can be executed.

- **Examples:**

  - `**SaveTourCommand**`: Encapsulates the logic for saving a new tour.

  - `**DeleteCommand**`: Handles the deletion of tours from both the UI and the database.

  - `**NavigateCommand**`: Facilitates navigation between different pages in the application.

- **Benefits:**

  - Decouples UI elements from the underlying logic.

  - Enhances testability by allowing commands to be tested independently of the UI.

  - Promotes reusability and consistency across the application.

Library decisions

O/R Mapper: Dapper, because of its easy use and high ranking in the market

Log Library: log4net, because it was easy to implement and was already used in class

Report-generation library: PDFsharp, because of its easy use and it was recommended in the requirements

Testing

For each layer there is an own testing project where the the most important functions are tested. This keeps the project organized and lets us test layers separately. The unit tests were written in the way to test the behaviour in valid and invalid inputs.

Unique Feature

PlaceholderTextBoxControl

The project introduces a custom WPF control called `PlaceholderTextBoxControl`, which extends the functionality of the standard `TextBox` by adding support for placeholder text. This feature improves user experience by providing contextual hints when the text box is empty.

Key Features:

- Dependency Properties:

  - `**Placeholder**`: A string property that defines the placeholder text displayed when the text box is empty.

  - `**IsEmpty**`: A boolean property that indicates whether the text box is empty, updated dynamically based on user input.

- **Dynamic Behavior**:

  - The placeholder text is automatically shown or hidden depending on the content of the text box.

  - The control overrides `**OnTextChanged**` and `**OnInitialized**` to ensure seamless updates to the `**IsEmpty**` property.

- **Integration**:

  - Can be easily incorporated into WPF layouts using XAML.

  - Supports styling and customization to match the application's design.

Benefits:

- Enhances usability by guiding users with placeholder text.

- Reduces the need for additional labels or instructions in the UI.

- Provides a polished and modern user experience.

Example Usage:

Tracked time

UI: ~ 100 hours

Business: ~ 15 hours

DataAccess: ~ 30 hours

Other (e.g. Structure, Docker, ...): ~ 35 hours

Lessons learned

Clear Separation: Required a clear vision and understanding of the project to separate the code into layers

Usage of Libraries: Implementing Libraries that provide fuctions to help performing complex tasks

Integrating external Services: Using an external APIs like OpenStreetMap to use Services outside of our local project