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

**Layered Architecture:**

The TourPlanner is split up into three (or four, if you count the unit tests) different sub-projects, all of which have their own distinct job to do, thus separating concerns and encapsulating the logic.

The layers are:

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

**Data Access Layer**

The DA Layer concerns itself with the entire access of the database by using the EntityFramework (PostgreSQL), as well as defining the models for the project (Tours and Tourlogs), thus adhering to MVVM Pattern principles.

The TourDbContext class defines the Database layout by the use of the DbSet class, to declare two distinct tables for the Tours and the Tourlogs.

The Tours and Tourlogs have a 1:n relationship, since one Tour can have many Tourlogs. Hence, Tourlogs references the TourId via a foreign key. This relationship is defined in the OnModelCreating method of the database, which is used in the EF Migration process. The database connection string is stored in a dbconfig file, which gets read via a custom JsonReader class, and gets passed to the optionsBuilder of the CreateDbContext method.

**Business Logic Layer**

The Business Logic Layer consists of two Classes, the TourService and TourlogService, both of which handle interactions with the database, such as insertion, deletion and modification, as well as getting distinct Tours / Tourlogs via their Id.

This approach ensures that the Presentation Layer is not able to directly interact with the Data Access Layer, further encapsulating the logic.

**Presentation Layer**

The Presentation Layer makes up the bulk of the code in this hand-in for a few reasons.

This layer solely concerns itself with handling the presentation of the GUI, while only having access to the Business Logic Layer. It consists of several Views (MainWindow, TourListView, TourlogView, etc…), one big ViewModel (TourViewModel) and two helper classes, which aid in minor features.

The MainWindow View consists of a few structures that do not change at all, such as the searchbar or several buttons, and contains two different ContentControl components. These switch the TourListView to AddTourView or EditTourView and the TourLogView to either AddTourlogView or EditTourlogView sub-views via button presses, which activate commands in the TourViewModel, which in turn changes the CurrentView property of the TourViewModel.

In total, the Project contains the following views:

* MainWindow
* TourListBox //displays all available tours solely their name
* TourListView //displays tour details
* TourLogsView //displays all tourlogs for a distinct tour
* AddTourView
* AddTourLogView
* EditTourView
* EditTourLogView
* TourImageView //placeholder image

The main bulk of the project is contained within the TourViewModel file.

This file has most of the logic needed for the GUI and its properties. We attempted to split up this ViewModel into SubViewModels, however, with our limited knowledge and having already written a good chunk of code, the refactoring became a bit cumbersome due to some issues with the DataContext, so we chose to leave it as is for the intermediate hand-in.

The Service classes are available in the TourViewModel by the usage of **Dependency Injection (DI),**which avoids having to “new-up” a service, every time a method is called that requires it, thus removing unnecessary boiler-plate code and improving maintainability.

The TourViewModel contains important properties, such as, the CurrentView being displayed, several ICommands in order to switch views, ObservableCollections for AllTours, TourDetails, SelectedTour, AllTourlogs, SelectedTourlog, NewTour, NewTourlog,… (you get the idea).   
  
Due to having to set the getters and setters for many of these properties, this led to a lot of boiler-plate code, which made the ViewModel become fairly bloated, but it’s still manageable.

Some important methods of the ViewModel include:

* LoadDataAsync – loads all available tours and tourlogs from the database
* UpdateTourDetails – updates the TourDetails property, when a new Tour is selected
* UpdateTourLogDetails – same as above
* SaveTourAsync – passes the user-entered-tour to the TourService, which inserts it into the database
* DeleteTourAsync – passes the selected tour to the TourService, which deletes the tour from the database
* EditTourAsync – passes the edited tour to the TourService, which updates the tour in the database
* SaveLogAsync
* DeleteTourLogAsync
* EditTourLogAsync
* GetSelectedTourId
* ClearInputs – clears the fields of the NewTour and EditTour properties, thus clearing previous user input and avoiding potential bugs

Lastly, the InputValidator Class handles the input validation of the user and contains two methods, one validation method for the tours and tourlogs each.

Both of them check for formatting and data types. If any of the user input is malformed, the methods either return a string with the according error message, or an empty string. If the string is empty, then no error has occurred, and the TourViewModel can call the Services.

This class is used in the TourViewModel via **Dependency Injection** and gets called whenever a user tries to add or edit a tour or tourlog.

**Unit Tests (XUnit)**

The fourth project folder handles all of the unit tests, which use the XUnit testing framework. There are a total of 12 tests, covering 3 main aspects of the code.

ModelsTest – tests the basic aspects of the model’s constructors, ensuring that there are no unforeseen mistakes, as well as testing the relationship between the Tours and Tourlogs (1:n).

InputValidatorTest – handles the InputValidator class and tests both of the validation methods. It tests if correctly structured Tours and Tourlogs return an empty string (OK response), and if malformed requests return the appropriate error message, for example, if the Author field of the tourlog only contains a singular “ “ whitespace, which displays an "Author cannot be empty!" error.

Lastly, ServiceTest handles the Services and their respective methods, ensuring that all of the important methods return the desired output, or manipulate the database correctly without any hiccups. These tests also incorporate an in-memory-database for simple testing, without having to directly work on the actual database, thus simplifying testability.