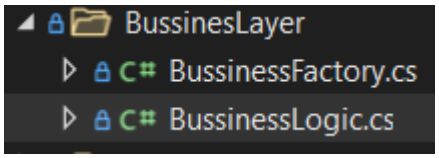
Protokoll

1. Design and Architecture

We have created an application based on the GUI frameworks C# (WPF). Our TourPlanner is structured in more layers: BussinessLayer, DataAccessLayer and UILayer.

1. **BussinessLayer**

   In the Bussiness layer, we implemented two classes: *BussinessFactory* and *BussinessLogic*.



   In the *BussinessFactory* we are reading all information from the config file (*Resources\Settings.xml*) with *XMLReader* class, which we implemented in *Util* folder in our Solution. In the *BussinessFactory* we are putting all information in the objects, we implemented in *DTO* folder, so it will be easier for use to use it, later in the project. We also created an Instance for the *BussinessFactory*, so we can easily use the data, so we used *Singleton pattern*.

   Credentials for the Database connection and HttpRequests are located in the *Settings.xml* and in *BussinessFactory* used and stored in *DatabaseDTO* and in *HttpDTO*. We also implemented the *Crypto* class in the *Util* folder and used it to hash all credentials.

   All Sql Commands we also implemented in *Settings.xml* and stored them in *SqlDTO*. And we used *Settings* file to define all paths we are going to use in the application.

```
1 reference
public BussinessFactory()
{
    try
    {
        string? path = System.IO.Path.GetDirectoryName(System.Reflection.Assembly.GetExecutingAssembly().GetName().CodeBase);
        string file = new Uri(path + @"\Resources\" + "Settings.xml").LocalPath;
        xmlReader = new XMLReader(file);
        ReadValues();
    }
    catch (Exception e)
    {
        Console.WriteLine("Fehler beim Initialisieren.\n" + e.Message);
        Environment.Exit(-1);
    }
}
```

In the *BussinessLogic* we implemented all the functions, we will need for this application to work. We are getting the objects from the *BussinessFactory* and we are using them in the BussinessLogic.

We are in almost every function using just string *routeId* as parameter. And most of the functions, have a Boolean as return type, so we can use this in the UI for the interaction with the user.

Also, we implemented some functions in the *Documents* folder, which we are using for the report creation and import and export.

For the import and export function, we used .xml format.

For the unique feature, we implemented the function to make the route favorite. User should click on the button and in database the column for the favorite route will be updated.

```
16 references
public class BussinessLogic
{
    private static BussinessLogic logicInstance = new BussinessLogic();
    1 reference
    public bool CreateRoute(string name, string from, string to, string transport, string comment)...
    1 reference
    public bool ModifyRoute(string from, string to, string name, string transport, string comment, string routeId)...
    1 reference
    public void DeleteRoute(string routeId)...
    1 reference
    public bool CreateLog(string comment, string difficulty, string totalTime, string rating, string routeId)...
    1 reference
    public bool ModifyLog(string comment, string difficulty, string totalTime, string rating, string logId)...
    1 reference
    public void Deletelog(string logId)...
    3 references
    public List<TourPreview> SelectTourNameId()...
    1 reference
    public bool CreateRouteReport(string routeId)...
    1 reference
    public bool CreateSummarizeReport(string routeId)...
    1 reference
    public string ImportRouteFromFile(string filename)...
    3 references
    public HttpResponseDTO SelectAllFromRoute(string routeId)...
    1 reference
    public bool ExportRouteToFile(string filename, string routeId)...
    0 references
    public void MakeRouteFavorite(string routeId)...
    1 reference
    public string CheckRoutePopularity(string routeId)...
    1 reference
    public string CheckRouteChildFriendliness(string routeId)...
    0 references
    public List<string> PrepareTableRouteForSearch(string searchText)...
    1 reference
    public List<TourLogDTO> SelectLogForRoute(string routeId)...
    13 references
    public static BussinessLogic LogicInstance...
}
```

2. DataAccessLayer

In the *DataAccessLayer* we implemented two classes: *HttpRequest* and *DatabaseConnection*.

We also used *Singleton pattern* here, so that will be easier to use it later on.

In the *HttpRequest* class we created two functions for the requests. To the first one, we are sending just the *HttpDTO* and we are receiving a new object with the response from the API.

```
3 references
public HttpResponseDTO GetRoutes(HttpDTO httpDTO)
{
    HttpResponseDTO responseDTO = new HttpResponseDTO();

    try
    {
        using (var client = new HttpClient(new HttpClientHandler { AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate }))
        {
            client.BaseAddress = new Uri(httpDTO.Url);
            HttpResponseMessage response = client.GetAsync("?from=" + httpDTO.From + "&to=" + httpDTO.To + "&key=" + httpDTO.Key).Result;
            response.EnsureSuccessStatusCode();
            string result = response.Content.ReadAsStringAsync().Result;
            responseDTO = JsonConverter.ConvertFromJson<HttpResponseDTO>(result);
            return responseDTO;
        }
    }
    catch (Exception e)
    {
        LoggerToFile.LogError(e.Message + "\n" + e.StackTrace);
        return responseDTO;
    }
}
```

To the second function we are sending the object we already filled with the response from API, to get the image. We are saving the image on the local disc and its path to the database.

```
3 references
public string GetRouteImage(HttpDTO httpDTO)
{
    try
    {
        using (var client = new HttpClient(new HttpClientHandler { AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate }))
        {
            client.BaseAddress = new Uri(httpDTO.MapUrl);
            HttpResponseMessage response = client.GetAsync("?start=" + httpDTO.From + "&end=" + httpDTO.To + "&size=600,400&key=" + httpDTO.Key).Result;
            response.EnsureSuccessStatusCode();

            byte[] result = response.Content.ReadAsByteArrayAsync().Result;
            string image = BussinessFactory.Instance.DirectoryDTO.ImagesPath + DateTime.Now.ToString("hhmmssffffff") + ".jpg";
            File.WriteAllBytes(image, result);

            return image;
        }
    }
    catch (Exception e)
    {
        LoggerToFile.LogError(e.Message + "\n" + e.StackTrace);
        return null;
    }
}
```

In the DatabaseConnection class we implemented all the functions we are using for the communication with database and also the connection to database.

Here we used *Singleton pattern* as well.

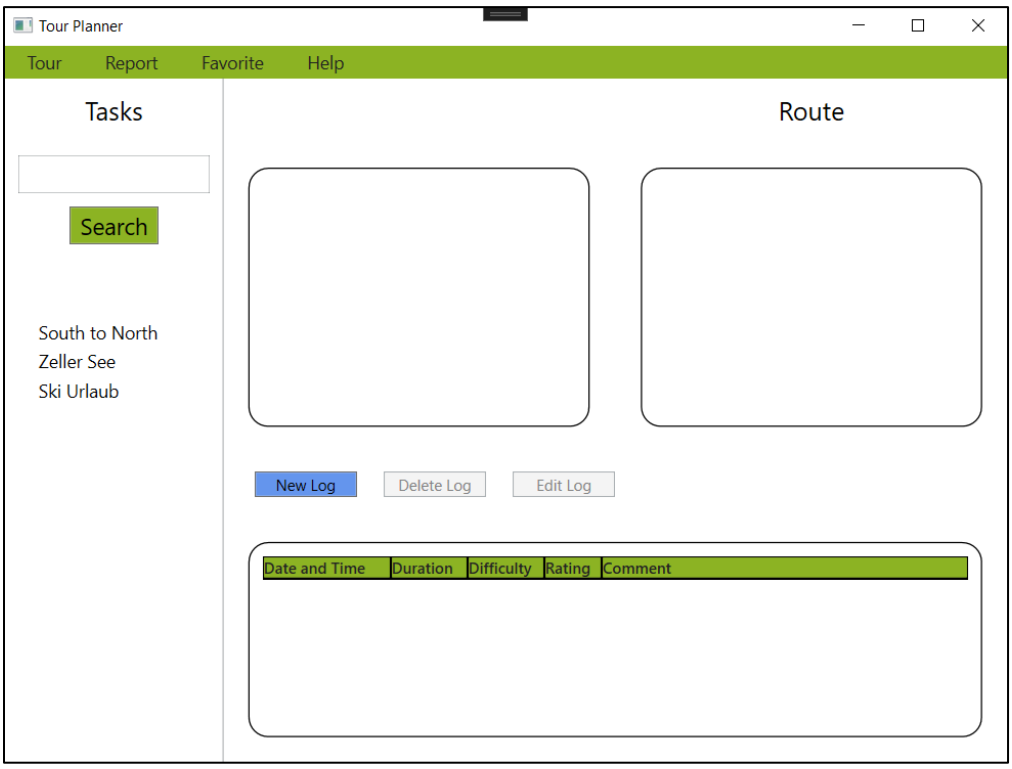
We also created *LoggerToFile* class in the *Util* folder and we used it to not allow SQL injections. LoggerToFile we used in every function to write the exception to the log file.

```
20 references
public class DatabaseConnection
{
    private static DatabaseConnection instance = new DatabaseConnection();
    private SqlConnection connection;

    19 references
    public static DatabaseConnection Instance...

    //CONNECTION
    1 reference
    public SqlConnection Connect()...

    public SqlConnection Connection()...
    0 references
    public void closeConnection()...

    //FUNCTIONS
    2 references
    public void ExecuteInsertRoute(string query, HttpResponseDTO HttpResponseDTO)...
    1 reference
    public void ExecuteUpdateRoute(string query, HttpResponseDTO HttpResponseDTO)...
    1 reference
    public void ExecuteDeleteRoute(string query, string tourId)...
    1 reference
    public void ExecuteInsertLog(string query, TourLogDTO tourLogDTO, string routeId)...
    1 reference
    public void ExecuteModifyLog(string query, TourLogDTO tourLogDTO)...
    1 reference
    public void ExecuteDeleteLog(string query, string logId)...
    1 reference
    public DataTable ExecuteSelectAllRoutes(string query)...
    8 references
    public DataTable ExecuteSelect(string query, string id)...
    1 reference
    public void ExecuteFavorite(string query, string routeId)...
}
```

3. UILayer

This is the top-most layer of the application where the user performs their activity. Let's take the example of any application where the user needs to fill up a form. This form is nothing but the Presentation Layer. In Windows applications Windows Forms are the Presentation Layer and in web applications the web form belongs to the Presentation Layer. Basically, the user's input validation and rule processing is done in this layer.

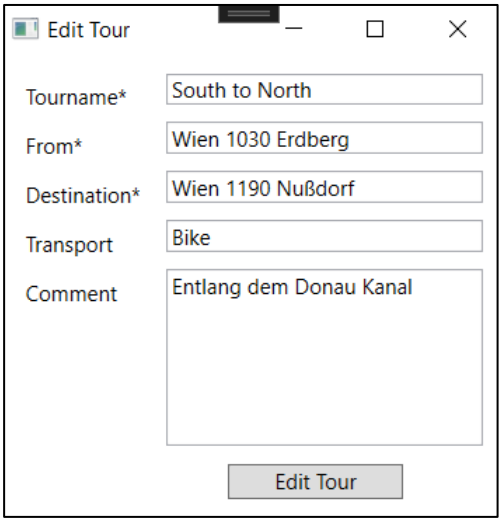
This is the top-most layer where the user interacts with the system. We will create simple windows as in the following:



The form contains one Menu with several Menu Items, Borders, Labels, Buttons, one List and two DataGrids. We also used Style Definitions for most of the Elements. In the load event of the form, we will pull all data and it will show in the DataGrid. There is another operation, where can the user fetch a specific tour by providing a text in the search box.

New and Edit Tour Windows

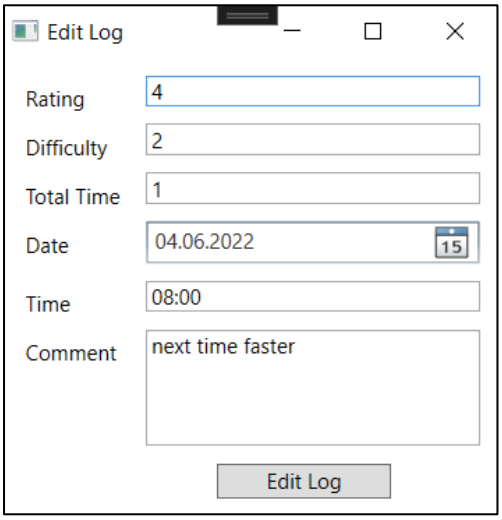
There are also four other windows that the user can call up. NewTourWindow.xaml and EditTourWindow are two of them and are quite similar but have a separate structure and look like this (see right).



The form consists of a grid containing labels, text boxes and a button. When editing a tour, the data from the ViewModel is inserted into the window to be able to edit existing data easily and not have to rewrite it. However, the NewTourWindow must remain empty when opened.

New and Edit Log Windows

The two remaining windows are CreateLogWindow.xaml and EditLogWindow.xaml, which are also quite similar.



The form consists of a grid containing labels, text boxes, a datepicker and a button. Here, too, the data is loaded from the ViewModel and inserted into the Edit Window.

2. NUnit tests

We created 20 NUnit tests.

17 of them are used to Test BusinessLogic and the functions we created there. Almost all of them are using a database Connection and all of them are returning values, so it was perfect designed to implement the tests.

We also created one test for the HttpRequest and two tests for DocumentCreation.

3. Lessons learned

We learned to work in the team on the same project and to implement the solution together. Git was very helpful for the teamwork.

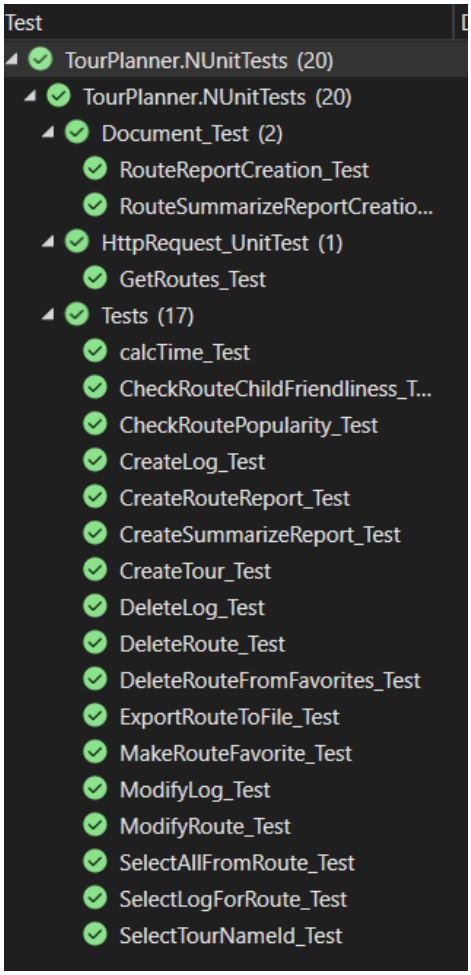
We also learned to structure the code in the layers and to use the config file. Using the config file was very easy to make some changes, for example for paths. We needed just to change it in config file and not in the logic itself.

MVVM is the maybe the most important thing we learned to use. Also, we learned to use the WPF.

We learned to create the HttpRequest in C# and to serialize and deserialize json in the objects.

We learned to use the report-generation library.

We learned that the logging in the file is very useful, because it is much easier to find an error there.



4. Time spent and link to the git

We spent together 112 hours to finish this project.

Link to our git repository: **https://github.com/PavleTomanovic/TourPlanner.**