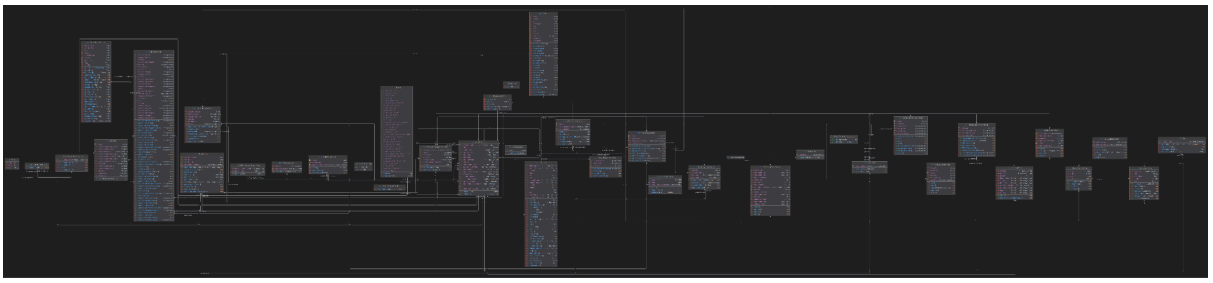
Protocol Tourplanner SWEN2:

Leitner Dominik, if22b170@technikum-wien.at
Widhalm Florian, if22b251@technikum-wien.at

App architecture:



We decided to split our application into several layers (according to the MVVM pattern):

- view
- viewmodel
- service
- repository
- entities

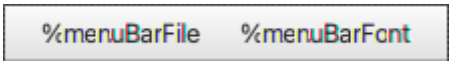
Design Pattern:

- Publisher/Subscriber

The view layer is only in direct communication with its corresponding viewmodel. Here the abstract view objects are bound via data binding to java objects. These bindings are either unidirectional or bidirectional. The viewmodel defines the next step in the process. Here comes the event publisher into play. We decided to use a publisher/subscriber design pattern to transfer information between the different viewmodels. Further are the services that contain the business logic. Here is most of the logic to be found. In some cases the services call the repositories which handle requests to the database.

Use cases:

MenuBar: Here the users can choose between different options. It is possible to report a pdf report or exit the program. It is also possible to change the font of the application.

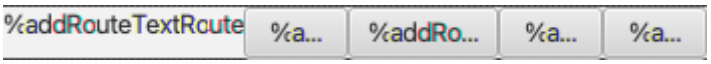


SearchBar: After inserting a searchterm the user can search the database after that term. Sadly we weren't able to finish this element.

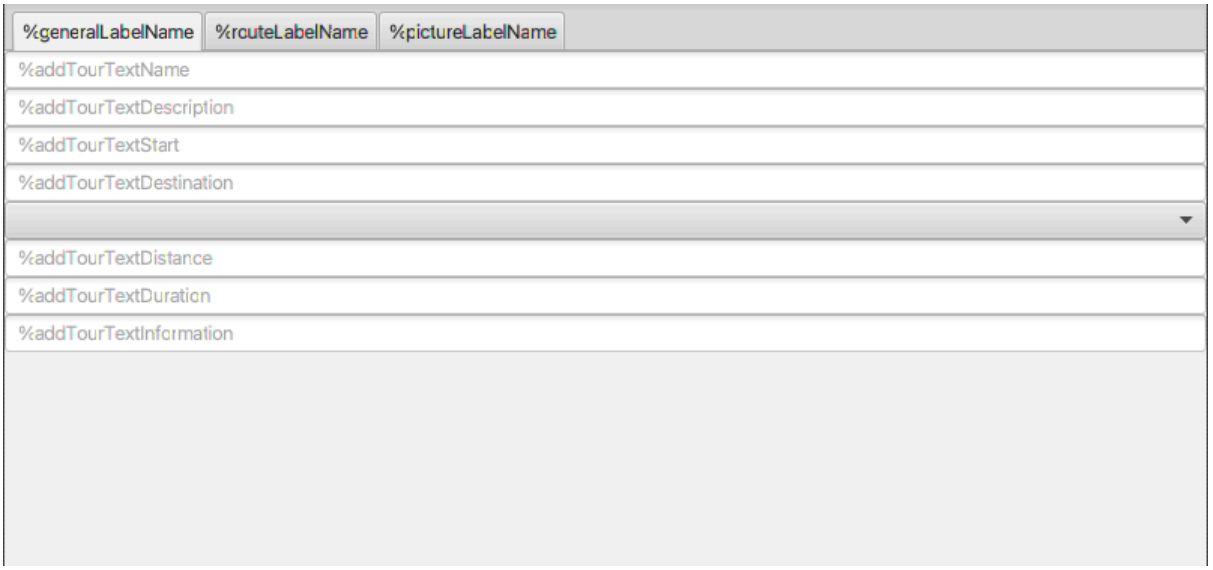
TourList: On the left side of the screen there is a list with all tours inside the database. If the user selects a tour, the values of that tour are displayed on the right side inside the text

fields. This is more or less the heart of the application, as the selection of a tour triggers the display on the right and the corresponding logs on the bottom of the application.

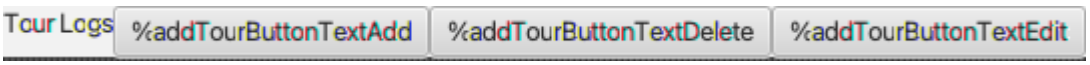
RouteButtons: These buttons are triggered by different events. To enable the add button all of the needed information needs to be inserted in the text fields on the right hand side. If the input is valid, the button is enabled and by pressing it the new tour is saved in the database. The remove and edit button become enabled if a tour is selected. Remove does what it implies and edit lets the user change values of the current selected tour. Save becomes only enabled when the edit button has been pressed and serves to save the newly edited tour.



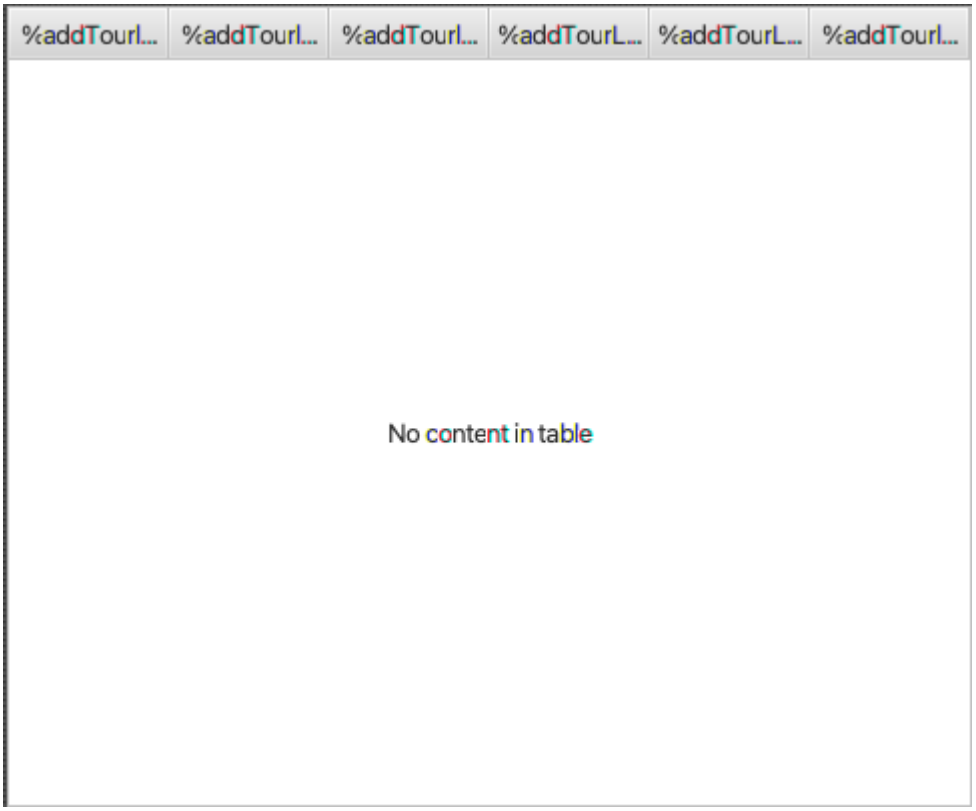
TabView: Here all data for a tour is displayed. The text fields allow user input to insert new data for a tour. After entering values for the start, destination and the type of transportation, the openrouteservice is called and calculates the distance and duration of the tour. If all fields are filled in the add button of the RouteButtons becomes enabled. As written before, if a tour is selected in the TourList, the corresponding data is displayed.



TourLogButtons: These buttons offer their implied uses after a tour is selected in the TourList. Add allows the user to add a new log entry; delete allows the user after selecting a log to delete that log; edit allows the user to edit the values of the log.

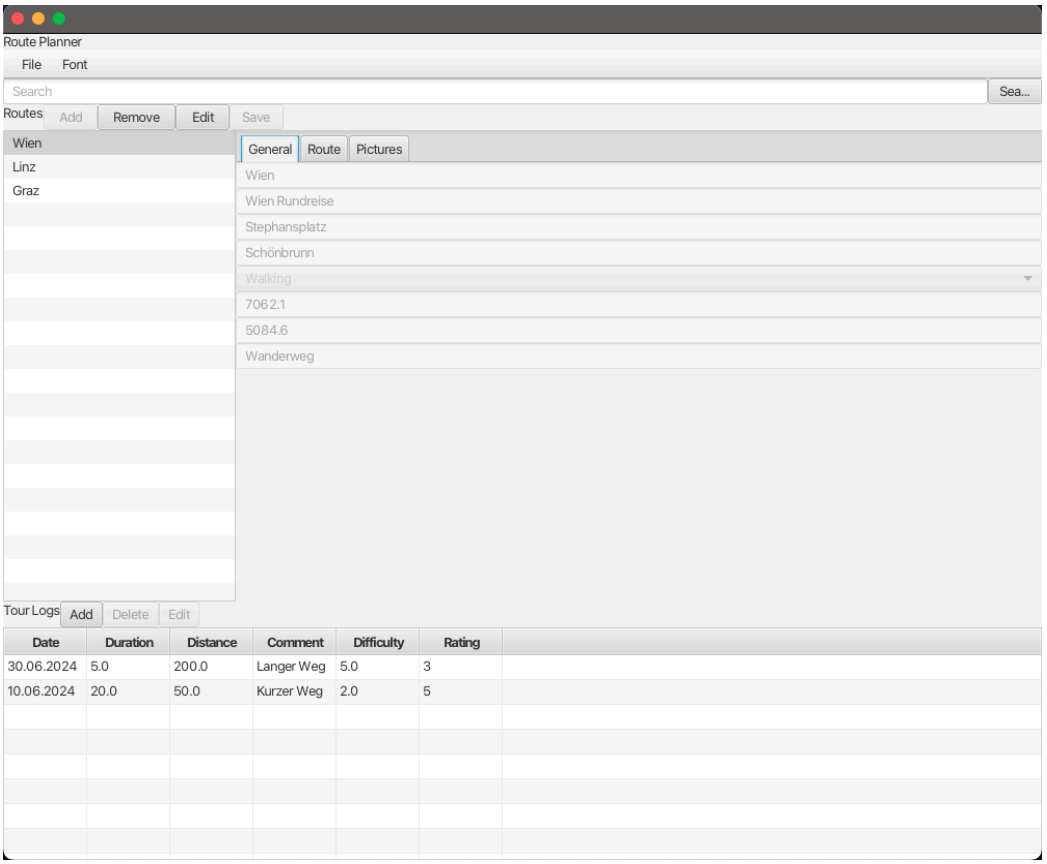


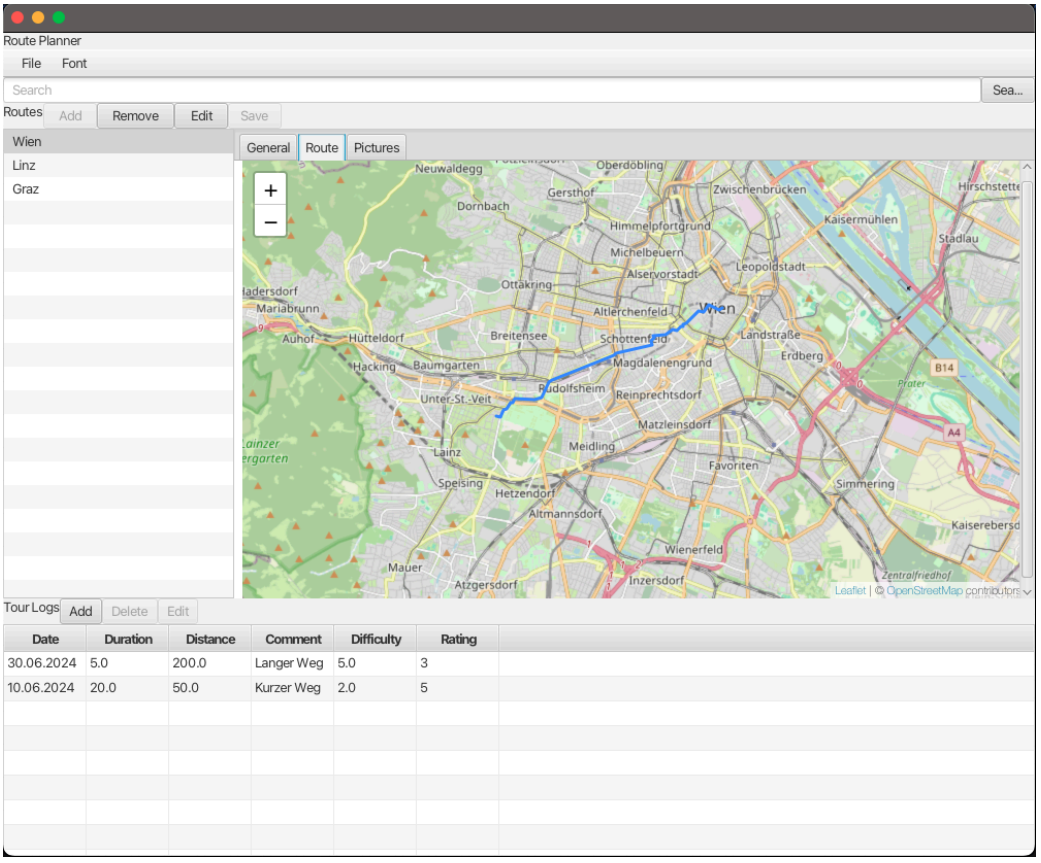
TourLogView: Here are all logs for a selected tour displayed.



UX:

The two screenshots show the current UI. The functionality has been described in the previous paragraphs.





Libraries:

For our project we used the following libraries:

- OpenJFX v.21: as the basis of the project
- Jakarta Persistence v.3.1.0: library for ORM and persistence
- Hibernate v.6.5.0: library for ORM and persistence
- Postgresql v.42.7.3: database
- Apache httpcomponents v.4.5.14: library for http handling
- Jackson v.2.17.1: json mapping
- Pdfbox v.3.0.2: library to create pdf report
- Log4j v.2.23.1: logging library

Testing:

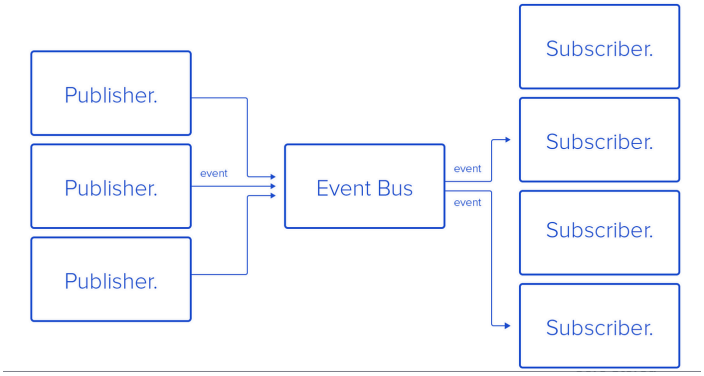
- Junit-Jupiter
- Mockito

Lessons learned:

Both of us will not use OpenJFX for frontend applications in the future. The reasons for that are that there is only limited or outdated documentation and there are many modern solutions to frontend development.
Furthermore we learned that unit tests should be done immediately and not in the end. In our application, the services and repositories are very nested and therefore hard to test.

Design pattern:

We used the publisher/subscriber design pattern to allow the communication between the different viewmodels. The publisher shares an event which is forwarded through the event bus and received via a subscriber, which triggers an action at the new destination. Therefore we have no direct communication between the different viewmodels.



Unit tests:

We decided to test our Services and some ViewModels. As mentioned earlier, our classes are somewhat nested, which makes it really hard to write unit tests for the repository or some other classes. That is something we will consider for the next projects and aim to do better the next time.

Unique feature:

For the unique feature we decided to implement the possibility to change the font of the application. For that we implemented a function that gets the current scene and the chosen font from a MenuItem in the MenuBarView and changes it in the corresponding MenuBarViewModel.

```
@FXML  1 usage   CruzeFW
public void fontCourierNew(ActionEvent event){
    currentScene = getCurrentScene(event);
    viewModel.changeFont( font: "CourierNew", currentScene);
}

private Scene getCurrentScene(ActionEvent event) {  3 usages   CruzeFW
    return menuBar.getScene();
}
```

```
public void changeFont(String font, Scene currentScene){  3 usages   CruzeFW
    if (currentScene != null) {
        logger.info( message: "changing font to {}", font);
        currentScene.getRoot().setStyle("-fx-font-family: '" + font + "'; -fx-font-size: 12px;");
    }
}
```

Tracked time:

~140 hours per student.

GIT:

https://github.com/Shayvin/04-SWEN-Tourplanner