TourPlanner

https://github.com/ang3lika1/TourPlanner.git

About the Project:

Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8
Maven Application based on JavaFX GUI framework.
The user can create/manage tours, associated logs and statistical data of tours.

All data is persisted in a database.

Structure

Database runs in Docker Container

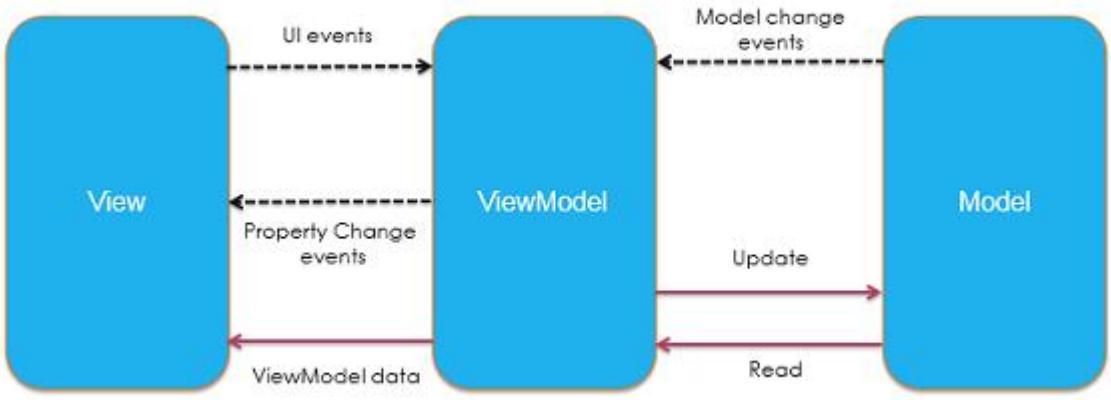
Tables:

- tour
- tourlog

DAO class for each table in the database encapsulate the logic required to the access data sources.

MVVM pattern:

Service classes call the methods of the DAO and define the functionality provided by the service. The services are used in the corresponding ViewModel class whose methods in turn are called by the responsible Controller. Finally the methods are triggered by user interaction with the GUI and the Controller handles particular actions (e.g. event on button click).

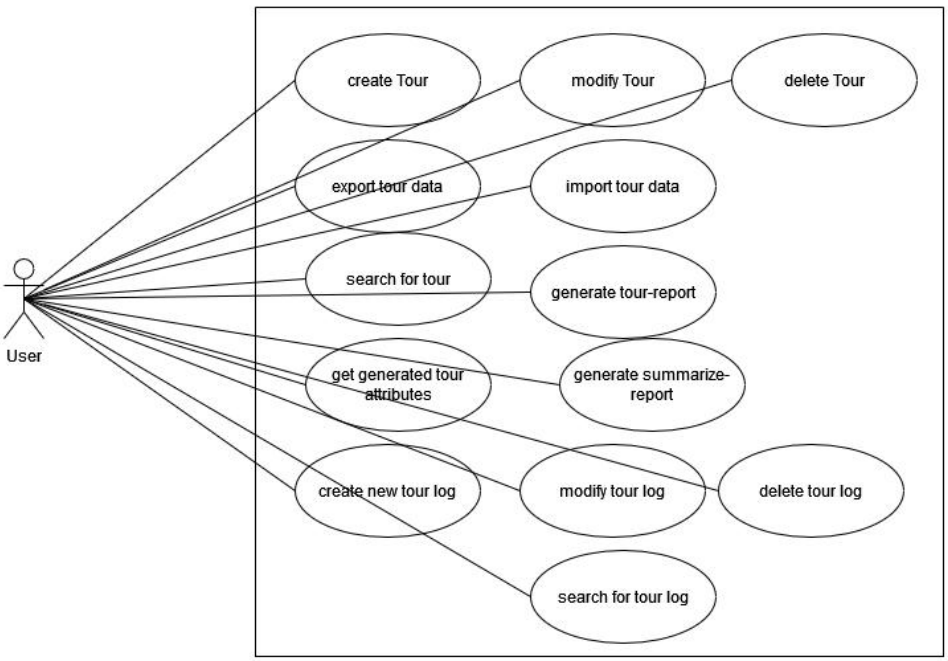


source:
https://www.codeguru.com/dotnet/differences-among-mvc-mvp-and-mvvm-design-patterns/

Use Cases

User can create, modify, delete tour and tourlogs whereat logs always among to one tour but one tour can hold multiple logs.
Tour data can be exported and imported (file format: .txt).
User can perform full-text search for tour and tourlogs.
A tour-report, containing all details of one tour can be generated in form of a pdf file.
A summarize-report for statistical analysis, which for each tour provides the the average time, -distance and rating over all associated tour-logs, can be generated in

form of a pdf file.



Unique Feature

The user can look at the direction maneuvers in a tab besides the Details and Map tab. Furthermore these directions can be downloaded in a text file. The maneuvers are received by the MapQuest API "route"->"legs"->"maneuvers"->(key="narrative").

```
JsonNode arrayNode =
objectMapper.readTree(json).get("route").get("legs");
     if (arrayNode.isArray()) {
         for (JsonNode jsonNode : arrayNode) {
            JsonNode maneuversNode = jsonNode.get("maneuvers");
            for (JsonNode narNode : maneuversNode) {
               String narrativeFieldNode =
narNode.get("narrative").asText();
               narratives.add(narrativeFieldNode);
            }
         }
     }
```

```
"legs": [
  {
    "hasTollRoad": true,
    "hasBridge": false,
    "destNarrative": "Proceed to SPITAL AM SEMMERING, STYRIA.",
    "distance": 63.703,
    "hasTimedRestriction": false,
    "hasTunnel": true,
    "hasHighway": true,
    "index": 0,
    "formattedTime": "01:04:41",
    "origIndex": -1,
    "hasAccessRestriction": false,
    "hasSeasonalClosure": false,
    "hasCountryCross": false,
    "roadGradeStrategy": [
      []
    ],
    "destIndex": 16,
    "time": 3881,
    "hasUnpaved": false,
    "origNarrative": "",
    "maneuvers": [
      {
        "distance": 0.082,
        "streets": [
          "B14",
          "Handelskai"
        ],
        "narrative": "Start out going northwest on B14/Handelskai.",
        "turnType": 0,
        "startPoint": {
          "lng": 16.406752,
          "lat": 48.224434
        },
```


Design Pattern

Singleton pattern and DAO pattern are used.

Singleton

Singleton is a creational design pattern which ensures that a class has only one instance, while providing a global access point to this instance. In this application the class "Database.java", responsible for the connection with the postgres database, is designed as Singleton. Since the database is a shared resource, the access should be controlled and therefore the number of instances is restricted.

The instance of the database is needed by every DAO and by using the Singleton Pattern, this instance is easily accessed globally.

```
public class Database {
   private static Database instance = null;
   protected String connectionString;
   protected Connection connection;

   private Database() {
      try {
         this.connectionString =
ConfigHelper.getIniString(ConfigHelper.getConfigIni(), "db", "url");
         this.connection =
DriverManager.getConnection(connectionString,
ConfigHelper.getIniString(ConfigHelper.getConfigIni(), "db", "user"),
ConfigHelper.getIniString(ConfigHelper.getConfigIni(), "db", "pw"));
      } catch (Exception e) {
         e.printStackTrace();
      }
   }

   public static Database getInstance() {
      if(instance==null){
         instance= new Database();
      }
      return instance;
   }

   public Connection getConnection() {
      return connection;
   }
```

DAO

```
public interface DAO<T> {

    Optional<T> get(int id);

    List<T> getAll(@Nullable Integer id);


    T create(T t) throws SQLException;

    T update(T t);

    void delete(T t);
}
```

The DAO pattern is used to isolate the business layer from the persistence layer(database).
"TourDAO" and "TourLogDAO" both implement the interface "DAO"

Unit Tests:

- Unit Tests for testing usage of MapQuest API:
  directions, calculated time, calculated distance.
- Unit Tests for database/DAO testing:
  Test Classes for Services (Tour/ TourLog) with mocks for DAO to avoid writing actual data to the real database. MockitoExtension used.

Tracked Time

subject	time in hours
MVVM structure	6
Database with Docker	2
DB connection and DAO	3
CRUD tour	5
CRUD tourlog	5
data binding, properties	10
unique feature	2
mapQuest API, json	12
Unit Tests	5
Debugging and creating hand over	4