

SWE 2 Tour Planner Dokumentation

GitHub

<https://github.com/martinadodmasejj/TourPlanner.git>

App Architecture

Der TourPlanner App ist nach die MVVM Pattern entwickelt, d.h es gibt ein DataAccessLayer, BusinessLayer, MainViewModel und Views.

Key Architectural Decisions

Maven

Maven wurde als Libraryverwaltung Tool verwendet, um neue externe Libraries hinzuzufügen.

JSON Jackson

JSON Jackson wurde verwendet, um die Daten aus der JSON Datenbank Config File auszulesen.

JSON in Java

JSON in Java wurde verwendet, um den http JSON Response Daten vom String Format im JSON Objekte umzuwandeln damit sie weiterverwendet werden können.

iText PDF

Die iTextPDF Library wurde verwendet, um die TourReports und TourReportSummaries zu generieren.

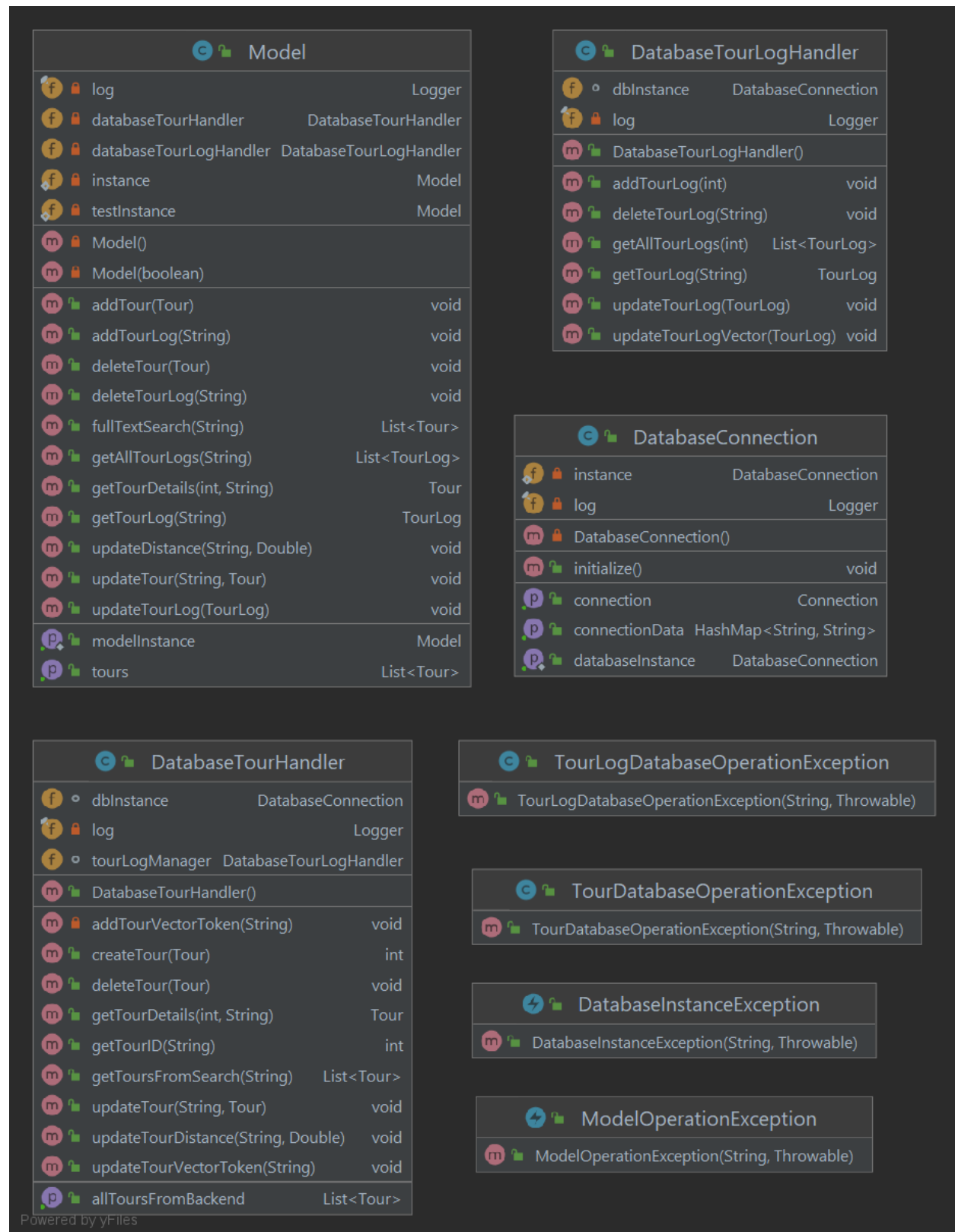
JUNIT

Die JUNIT Library wird verwendet, um Unit Tests umzusetzen.

Tour Images

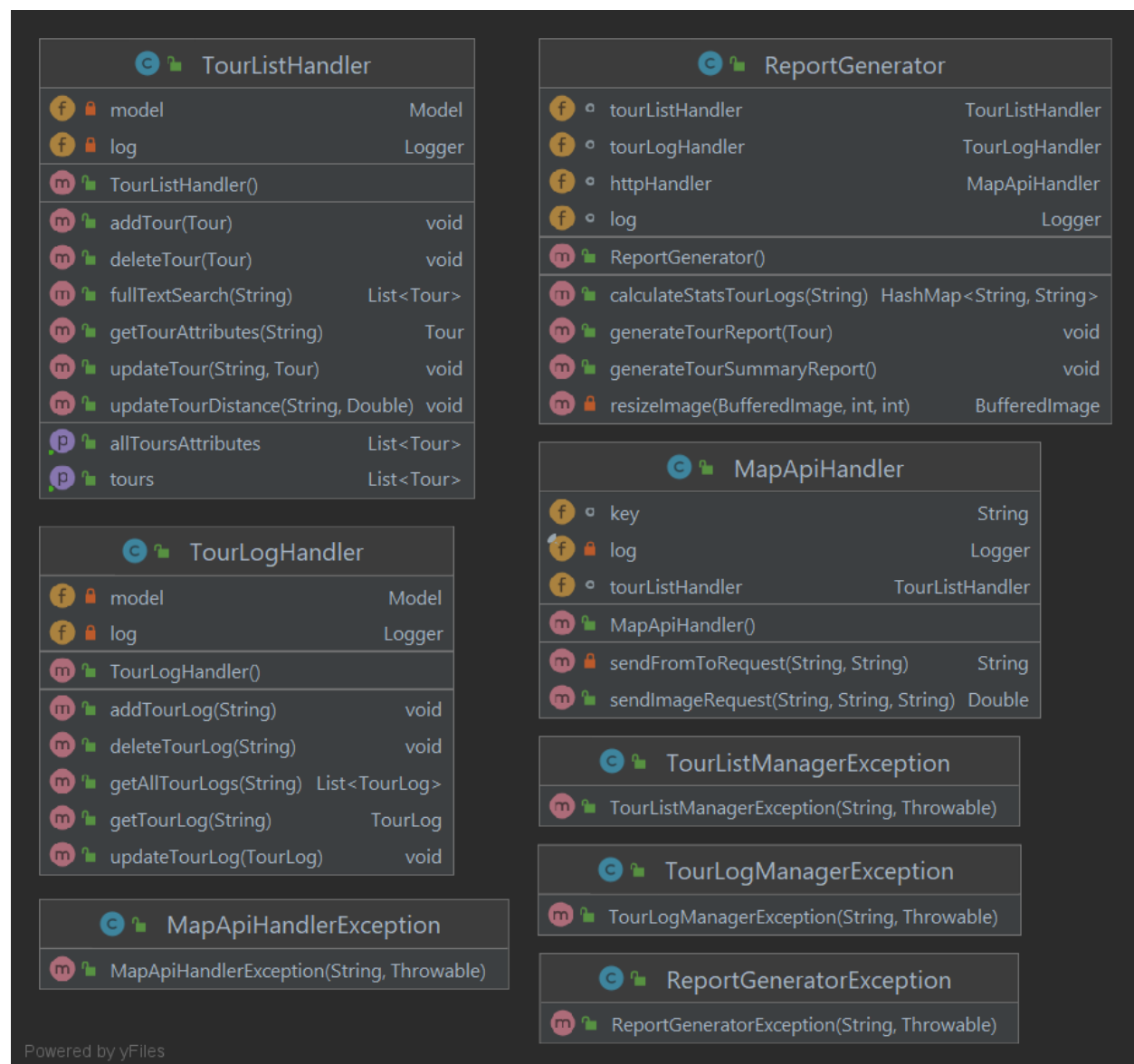
Tour Images werden in eine lokale Verzeichnis im Projektverzeichnis gespeichert. Wenn ein Tour neu erstellt oder aktualisiert wird, dann wird das neue Tour Image in dem Verzeichnis gespeichert und das alte Tour Image gelöscht. Somit werden http Requests und Performance erspart, und anstatt jedes Mal eine neue Request zu schicken wird das gespeicherte Bild dafür verwendet.

DAL – DataAccessLayer



Der DataAccessLayer ist von mehreren Packages aufgebaut. Die Database Package kümmert sich um die Tour und Tour Logs Datenbankoperationen wie z.b Inserts,Update,Delete,Search. Die Klassen können einen DatabaseInstance von die DatabaseConnection Klasse anfordern, um Operationen durchzuführen. Die Connectiondaten werden aus die config.json File gelesen. Die Operationen auf Datenbankebene sind auf zwei Klassen geteilt DatabaseTourHandler und DatabaseTourLogHandler um den single-responsibility principle zu erfüllen. Als Interface dient die Model Klasse, die alle Funktionalitäten an die andere Layer zur Verfügung stellt. Zusätzlich hat jede Klasse seine eigene Exceptions, um detaillierte Auskunft über Errors möglich zu machen.

BusinessLayer



Der BusinessLayer kümmert sich um das Verwalten und Bearbeiten der Daten, die von Model ausgeliefert werden. Diese Layer entsteht aus der ReportGenerator, MapApiHandler, und die zuständige TourList und TourLog Handlers. Die TourList und TourLog Handlers kommunizieren mit den Model, um Daten von Backend rauszuholen. Die Daten werden dann bearbeitet und stehen bereit für den ViewModel zum Holen. Der Report Generator verwendet die Handler, um Daten von Backend zu lesen und um Pdfs entsprechend zu generieren. Zusätzlich werden Statistiken berechnet für den gesamten TourSummaryReport. Der MapApiHandler kümmert sich um das Schicken und Empfangen von http Requests and den MapApi. Dort werden die JSON Daten behandelt und weitergeleitet. Jede Klasse definiert seine eigene Exceptions zusätzlich.

View / ViewModel

MainViewModel	Controller	App
<ul style="list-style-type: none"> tourListHandler TourListHandler tourLogHandler TourLogHandler log Logger reportGenerator ReportGenerator tourLogs ObservableList<TourLog> mapApiHandler MapApiHandler tourName StringProperty tourDistance StringProperty tourDescription StringProperty routeInformation StringProperty fromDestination StringProperty toDestination StringProperty searchField StringProperty tourImage ObjectProperty<Image> tourLogsTable ObjectProperty<ObservableList<TourLog>> selectedListitem Tour selectedLog TourLog 	<ul style="list-style-type: none"> log Logger viewModel MainViewModel tourList ListView attributesTab Tab tourName TextField tourDistance TextField searchField JFXTextArea tourDescription TextArea routeInformation TextArea fromDestination TextField toDestination TextField tourImage ImageView selectionTab TabPane logTab Tab tourLogsTable TableView dateColumn TableColumn reportColumn TableColumn distanceColumn TableColumn timeColumn TableColumn ratingColumn TableColumn avgSpeedColumn TableColumn remarksColumn TableColumn weatherColumn TableColumn jouleColumn TableColumn authorColumn TableColumn 	<ul style="list-style-type: none"> scene Scene App() App loadFXML(String) Parent main(String[]) void start(Stage) void root String
<ul style="list-style-type: none"> MainViewModel() addTour() void addTourLog() void deleteTour() void deleteTourLog() void deleteTourPicture() void displayTourAttributes() void displayTourPicture(String) void fromDestinationProperty() StringProperty generateTourReport() void generateTourSummaryReport() void getAllTourLogs() void routeInformationProperty() StringProperty saveTourPicture(Tour) void searchFieldProperty() StringProperty searchTours() void toDestinationProperty() StringProperty tourDescriptionProperty() StringProperty tourDistanceProperty() StringProperty tourImageProperty() ObjectProperty<Image> tourLogsTableProperty() ObjectProperty<ObservableList<TourLog>> tourNameProperty() StringProperty updateTour() void updateTourLogAuthor(String) void updateTourLogAvgSpeed(Double) void updateTourLogDistance(Double) void updateTourLogJoule(Integer) void updateTourLogRating(Integer) void updateTourLogRemarks(String) void updateTourLogReport(String) void updateTourLogTime(Double) void updateTourLogWeather(String) void 	<ul style="list-style-type: none"> Controller() addTour(ActionEvent) void addTourLog(ActionEvent) void deleteTour(ActionEvent) void deleteTourLog(ActionEvent) void displayTourDetails(Event) void displayTourInfo(Event) void generateSummaryReport(ActionEvent) void generateTourReport(ActionEvent) void getAllTourLogs(Event) void initialize(URL, ResourceBundle) void initializeEditableTable() void initializeLogTable() void searchTours(ActionEvent) void updateTour(ActionEvent) void updateTourLogAuthor(CellEditEvent<TourLog, String>) void updateTourLogAvgSpeed(CellEditEvent<TourLog, Double>) void updateTourLogDistance(CellEditEvent<TourLog, Double>) void updateTourLogJoule(CellEditEvent<TourLog, Integer>) void updateTourLogRating(CellEditEvent<TourLog, Integer>) void updateTourLogRemarks(CellEditEvent<TourLog, String>) void updateTourLogReport(CellEditEvent<TourLog, String>) void updateTourLogTime(CellEditEvent<TourLog, Double>) void updateTourLogWeather(CellEditEvent<TourLog, String>) void 	<ul style="list-style-type: none"> CustomIntegerStringConverter converter IntegerStringConverter CustomIntegerStringConverter() fromString(String) Integer showAlert(Exception) void toString(Integer) String
<ul style="list-style-type: none"> ChangeListener<Tour> ChangeListener<TourLog> ObservableList<Tour> 		<ul style="list-style-type: none"> CustomDoubleStringConverter converter DoubleStringConverter CustomDoubleStringConverter() fromString(String) Double showAlert(Exception) void toString(Double) String
		<ul style="list-style-type: none"> CustomTourConverter cell ListCell<Tour> CustomTourConverter() CustomTourConverter(ListCell<Tour>) fromString(String) Tour toString(Tour) String

Die View Layer kümmert sich um das Darstellen aller Daten um UI und um das Lesen aller Benutzer Inputs und Anforderungen. Um den MVVM Pattern zu erfüllen gibt's ein MainViewModel Klasse, die mit den JavaFx Controller verbunden ist. Die MainViewModel Klasse ist bidirektional mit dem Controller verbunden, d.h. bidirektional gebundene Werte werden auch beim MainViewModel automatisch erneuert. Zusätzlich gibt es 3 ConverterKlassen: CustomIntegerConverter, CustomDoubleConverter, CustomTourConverter. Die dienen, um Objekte in dem UI darstellen zu können beispielsweise: in dem TourList wird nicht der TourObjekt selbst dargestellt, sondern nur der TourName.

Custom Objects

Es gibt 3 neu definierten Datentypen, um Daten einzupacken und durch die Layer korrekt zu übertragen.

Tour	
from	String
to	String
Tour()	
Tour(String)	
Tour(int, double, String, String, String)	
Tour(int, double, String, String, String, String, String)	
toString()	String
routeInformation	String
tourDescription	String
tourDistance	double
tourFrom	String
tourID	int
tourName	String
tourTo	String

TourLog	
TourLog()	
TourLog(String, double, double, String, int, double, String, String, int, String)	
author	String
avgSpeed	double
date	String
duration	double
joule	int
logReport	String
rating	int
remarks	String
timestamp	String
traveledDistance	double
weather	String

Tour Log Properties

Ein TourLog beinhaltet fünf neue Attribute zusätzlich zum Spezifikation

TourLog	
TourLog()	
TourLog(String, double, double, String, int, double, String, String, int, String)	
author	String
avgSpeed	double
date	String
duration	double
joule	int
logReport	String
rating	int
remarks	String
timestamp	String
traveledDistance	double
weather	String

Design Patterns

Singleton Pattern

```
public static DatabaseConnection getInstance() throws DatabaseInstanceException {  
    if(instance==null){  
        instance=new DatabaseConnection();  
    }  
    return instance;  
}  
  
private DatabaseConnection() throws DatabaseInstanceException {  
  
    log = LogManager.getLogger(DatabaseConnection.class);  
    connectionData = new HashMap<String, String>();  
    ObjectMapper mapper = new ObjectMapper();  
    Map<?, ?> readValues = null;  
  
    try {  
        readValues = mapper.readValue(Paths.get( first: "config.json").toFile(), Map.class);  
        for (Map.Entry<?, ?> entry : readValues.entrySet()) {  
            connectionData.put(entry.getKey().toString(), entry.getValue().toString());  
        }  
        log.info("Read Connection Attributes from config file");  
        initialize();  
    } catch (JsonMappingException jsonMappingException) {  
        throw new DatabaseInstanceException("could not convert JsonMapping into Java Logic",jsonMappingException);  
    } catch (JsonParseException jsonParseException) {  
        throw new DatabaseInstanceException("could not parse JsonData into Java",jsonParseException);  
    } catch (IOException ioException) {  
        throw new DatabaseInstanceException("could not read config file",ioException);  
    }  
}
```

Die DatabaseConnection Klasse ist eine Singleton. D.h es gibt nur eine einzige globale Instanz von die DatabaseConnection Klasse. Durch das Aufrufen von getInstance() wird eine neue Instanz erzeugt, wenn es keines gibt oder die existierende Instanz zurückgegeben. Die Instanz liest dann die Daten aus der config.json Datei und baut die einzelne Datenbankverbindung ein.

Testing Decisions

The screenshot displays several test classes in a code editor, each with its methods and return types listed. The classes are:

- ModelTest**
 - dataModelBackend: Model
 - testTour: Tour
 - ModelTest(): void
 - testAddTourLog(): void
 - testFullTextSearch(): void
 - testGetTourDetails(): void
 - testTourDistanceUpdate(): void
 - testTourUpdate(): void
 - testToursCreation(): void
 - testUpdateTourLog(): void
- TourLogTest**
 - logReport: String
 - traveledDistance: double
 - rating: int
 - duration: double
 - avgSpeed: double
 - author: String
 - remarks: String
 - joule: int
 - weather: String
 - timestamp: String
 - tourLog: TourLog
 - TourLogTest(): void
 - testAuthor(): void
 - testAvgSpeed(): void
 - testDuration(): void
 - testJoule(): void
 - testLogReport(): void
 - testRating(): void
 - testRemarks(): void
 - testTimestamp(): void
 - testTraveledDistance(): void
 - testWeather(): void
- TourTest**
 - tourID: int
 - tourName: String
 - tourDistance: double
 - tourDescription: String
 - routeInformation: String
 - from: String
 - to: String
 - tour: Tour
 - TourTest(): void
 - createTourDescriptionTest(): void
 - createTourIDTest(): void
 - createTourNameTest(): void
 - createTourRouteFrom(): void
 - createTourRouteInfoTest(): void
 - createTourRouteTo(): void
- ReportGeneratorTest**
 - ReportGeneratorTest(): void
 - calculateStatsTourLogs(): void
 - calculateStatsTourLogsEmpty(): void
- CustomRatingStringConverterTest**
 - converter: CustomRatingStringConverter
 - CustomRatingStringConverterTest(): void
 - testNegativeRating(): void
 - testRatingOver5(): void
 - testWorkingRating(): void
- CustomIntegerStringConverterTest**
 - converter: CustomIntegerStringConverter
 - CustomIntegerStringConverterTest(): void
 - ConvertIncorrectStringToInteger(): void
 - ConvertIntegerToString(): void
 - ConvertStringToInteger(): void
- CustomDoubleStringConverterTest**
 - converter: CustomDoubleStringConverter
 - CustomDoubleStringConverterTest(): void
 - ConvertDoubleToString(): void
 - ConvertIncorrectStringToDouble(): void
 - ConvertStringToDouble(): void
- CustomTourStringConverterTest**
 - converter: CustomTourConverter
 - tourObservableList: ObservableList<Tour>
 - tourList: List<Tour>
 - CustomTourStringConverterTest(): void
 - testTourToStringConversion(): void
- MapApiHandlerTest**
 - testResponse: String
 - MapApiHandlerTest(): void
 - testDistanceConversion(): void
 - testLatLngConversion(): void

Powered by yFiles

CustomInteger/Double/TourStringConverterTest/RatingConverterTest

Die Klassen dienen, um zu testen ob die Objekte vom Backend korrekt in ein passendes Format für den UI umgewandelt werden. Daher ist es wichtig zu testen, ob das der Fall ist damit es nicht rohe Objektdaten in den UI dargestellt werden.

ReportGeneratorTest

Die ReportGeneratorTest Klasse testet, ob der ReportGenerator Tourdaten richtig einsammelt und ob die Daten für die Reports korrekt behandelt werden. Also hier wird nicht der iText Library getestet, sondern nur die Datensammlung für einzelne Tourreports und die Berechnung von die Averages für den Tour Summary Report. Daher ist es wichtig zu bestimmen, ob die Daten für die PDF-Generierung korrekt gesammelt werden

MapApiHandlerTest

In dieser Instanz wird getestet, ob die JSON String Daten korrekt im JSON Objekte umgewandelt werden. In den MapApiHandler werden 2 http Requests nacheinander geschickt und die Daten dazwischen müssen korrekt behandelt werden. Die Klasse testet die Datenbehandlung von die MapApi Klasse und nicht die MapApi selbst.

ModelTest

Die ModelTest ist besonders wichtig, weil die Modelklasse als Hauptinterface für die Datenmanipulation dient. D.H es muss sichergestellt werden, dass Daten richtig gespeichert und geliefert werden. Die Klasse führt Test auf mögliche Datenbankoperationen.

TourTest / TourLogTest

Daher die Tour und TourLog Klassen als Datentypen verwendet werden, ist es kritisch, dass die Attribute korrekt gespeichert werden.

Lessons Learned

Architecture Planning

Durch ein genaueres Plan und UML Diagramm könnte ein bisschen mehr Zeit sparen. Also das Trennung von die Layers musste neu gemacht werden, um die MVVM Pattern zu halten. Durch besseres Planung könnte man die Klassen sauber trennen und diese Refactoring vermeiden.

Mehr Git Branches

Es war öfters der Fall, dass manche Git Branchen übersprungen wurden und das war natürlich nicht der beste Ansatz, wenn es zu Fehler kam. Wenn da mehr Branches erstellt wurden, wurde die Arbeit beim Fehlerkorrektur und Finden viel erleichtert, weil da mehrere genauere Kontrollpunkte gibt's, um zurückzuspringen beim Fehler.

Time Tracking

Singleton database 15 min

UI Creation 5 Std

Reading from config file 1 Std

Show data from tour database in list 1std

Custom Converters for the UI Lists and Tables 3 Std

Custom Datatypes for Lists and Tables 1 Std

Selecting tour from list 1std

Create new Tour 2 Std

Delete Tour 0.5 Std

Update Tour 1 Std

Image file management 1 Std

singleton Model 10 min

MAP API Images -- 5 Std

FROM TO in tour database -- 2Std

check from to loading when they dont exists 1Std

Logging - 3std

Reporting - PDF of Tour generation (Single Tour) - 4std

Reporting – PDF Summary Report – 2 Std

Fulltextsearch 3 Std

http requests tune up 0.5 Std

Distance von MapQuest - pointers 1 Std

Unit Testing done – 10 Std

Tour Logs add - delete - update - select 7Std

Search for Tour Log Details - 2Std

Exception development -- 5 Std

Total: ~ 44 Std