

Технически университет - Варна

Факултет: ФИТА

Катедра: СИТ

Специалност: СИТ

Дисциплина:

Обектно ориентирано програмиране проект – 2 част

Проект на тема: Библиотека

Изготвен от:

Владимир Стоянов Филчев – 21621640

Светослав Касабов

**Структура на документацията**

**Глава 1. Увод…………………………………………………4**

1.1.Пълно описание на проекта (Условие)………………………………………4

1.2 Интерпретация на условието ...........................................................................9

**Глава 2. Анализ на проблема……………………………….10**

2.1.Функционални изисквания…………………………………………………...5

2.2.Проучване и избор на технологии за реализация съвместими с Java и JavaFx………………………………………………………………………………6

2.3.Структура на проекта (Анализ и избор на архитектурен модел) .……………………………………………………………………………………27

2.4.Дефиниция на модулите на системата……………………………………..30

**Глава 3. Проектиране на системата……………………...33**

3.1.Проектиране на отделните модули (бази данни, бизнес логика, презентационен слой)……………………………………………………………33

3.2.UML(Унифициран език за моделиране) диаграми………………………...70

3.3.Концептуален модел на базата от данни…………………………………...76

**Глава 4. Реализация на системата……………….……….78**

4.1.Реализация на базата от данни (PostGresSQL) – Релационна схема и описание на таблиците…………………………………………………………..78

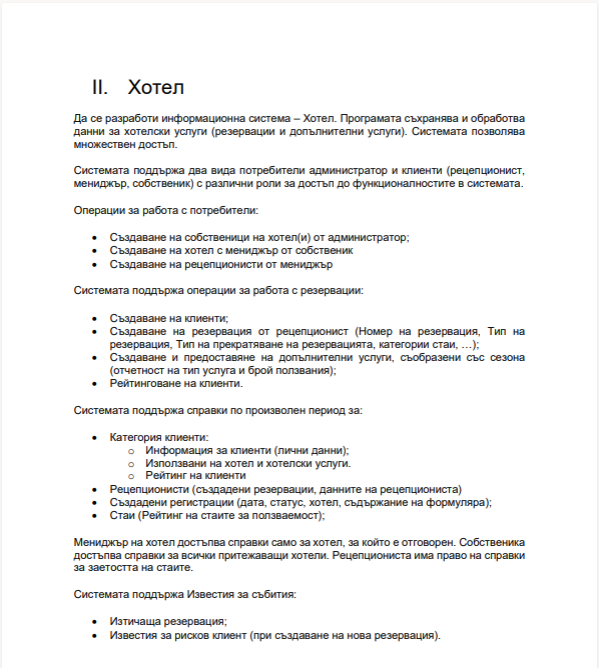
4.2.Реализация на слоя за работа с базата данни (DAO (Data Accessable Object))……………………………………………………………….82

4.3.Реализация на бизнес логика (Services)…………………………………….83

4.4.Реализация на графичен интерфейс (JavaFX)……………………………...85

**Глава 5. Тестови резултати…………………………..........89**

5.2. Тестови резултати на апликацията………………………………………...94



## **Интерпретация на условието (Сценарии на употреба)**

### **Информационна система: Хотел**

### **Актьори:**

* **Собственик** – създава хотели и назначава мениджъри
* **Мениджър** – управлява персонал и справки за конкретен хотел
* **Рецепционист** – създава резервации, обслужва клиенти
* **Клиент** – използва хотелските услуги
* **Система** – автоматизира събития и справки
* **Администратор -** създава собственици на хотели.

## **Use Case 1: Създаване на собственик**

* **Актьор:** Администратор
* **Описание:** Администратора прави регистрация на нов собственик в системата
* **Предусловия:** Администратора да е влязъл в системата.
* **Основен сценарий:**
  1. Администратора избира опция „Създай собственик“.
  2. Въвежда се име, username, password и др.

## **Use Case 2: Създаване на хотел**

* **Актьор:** Собственик
* **Описание:** Собственикът създава запис за нов хотел в системата.
* **Предусловия:** Собственикът е влязъл в системата.
* **Основен сценарий:**
  1. Собственикът избира опция „Създай хотел“.
  2. Въвежда се име, адрес, категории стаи, услуги и др.
  3. Системата валидира и записва информацията.

## **Use Case 3: Назначаване на мениджър**

* **Актьор:** Собственик
* **Описание:** Собственикът назначава мениджър на съществуващ хотел.
* **Основен сценарий:**
  1. Собственикът избира хотел.
  2. Въвежда данни за мениджъра.
  3. Системата предоставя права на мениджъра.

## **Use Case 4: Назначаване на рецепционист**

* **Актьор:** Мениджър
* **Описание:** Мениджърът назначава рецепционисти за своя хотел.
* **Основен сценарий:**
  1. Мениджърът избира „Създай рецепционист“.
  2. Въвежда лични данни и роля.
  3. Рецепционистът получава достъп до функционалности.

## **Use Case 5: Регистрация на клиент**

* **Актьор:** Рецепционист
* **Описание:** Създава се клиентски профил при резервация или на рецепция.
* **Сценарий:**
  1. Въвеждат се лични данни на клиента.
  2. Системата създава профил и го използва при резервации.

## **Use Case 6: Създаване на резервация**

* **Актьор:** Рецепционист
* **Описание:** Създаване на резервация за клиент.
* **Основен сценарий:**
  1. Избира се клиент или се създава нов.
  2. Въвеждат се данни за престоя – дати, стая, тип резервация.
  3. Системата създава резервацията и изчислява цена.

## **Use Case 7: Добавяне на допълнителни услуги**

* **Актьор:** Рецепционист
* **Описание:** Добавя се допълнителна услуга (СПА, храна и др.) към съществуваща резервация.
* **Сценарий:**
  1. Избира се активна резервация.
  2. Избира се услуга и брой ползвания.
  3. Системата актуализира цената и детайлите на резервацията.

## **Use Case 8: Генериране на справки**

* **Актьори:** Мениджър, Собственик, Рецепционист
* **Описание:** Извличане на данни според права:  
  + **Собственик:** Всички хотели, мениджъри, заетост
  + **Мениджър:** Хотел, стаи, служители
  + **Рецепционист:** Резервации, клиенти
* **Примери за справки:**
  + Използване на услуги по клиент
  + Рейтинг на клиенти и стаи
  + История на резервации

## **Use Case 9: Системни известия**

* **Актьор:** Система
* **Описание:** Автоматично известяване при събития.
* **Сценарий:**
  + При наближаващ край на резервация – известие към рецепционист.
  + При създаване на резервация от рисков клиент – предупреждение.

## **Анализ на проблема**

### **2.1 Цел на проекта**

Да се разработи информационна система за управление на хотелски услуги, която да:

* Поддържа множество роли и нива на достъп;
* Управлява хотели, персонал, клиенти, резервации и допълнителни услуги;
* Осигурява автоматизирани справки и известия.

### **Съществуващи проблеми, които трябва да бъдат решени:**

1. **Ръчно и разпокъсано управление на данни:**
   * В реална среда много хотели използват отделни системи за резервации, клиенти и услуги.
   * Това води до дублиране на информация, загуба на данни и липса на синхронизация.
2. **Ниска проследимост и контрол върху служителите:**
   * Липсва ясна йерархия и права на достъп.
   * Рецепционисти могат да извършват действия, без надзор от мениджър или собственик.
3. **Невъзможност за централизирано управление на повече от един хотел:**
   * За собственици с няколко хотела е трудно да наблюдават заетост, приходи и рейтинги от едно място.
4. **Липса на персонализирани услуги и клиентска история:**
   * Не се проследява история на резервации и предпочитания на клиентите.
   * Това възпрепятства лоялността и персонализираното обслужване.
5. **Няма автоматични известия и предупреждения:**
   * Служителите не получават напомняния за важни събития, като изтичане на резервации или рискови клиенти.
   * Това може да доведе до грешки, пропуски и недоволство.

### **Изисквания към системата (обобщени):**

#### **Функционални:**

* Управление на потребители и роля-достъп (администратор, собственик, мениджър, рецепционист);
* Създаване и управление на хотели;
* Назначаване на служители;
* Регистрация на клиенти и резервации;
* Добавяне на допълнителни услуги;
* Генериране на справки;
* Известия при ключови събития.

#### **Нефункционални:**

* Сигурност на данните и достъпа;
* Удобен и интуитивен интерфейс;
* Възможност за мащабиране при управление на повече от един хотел;
* Висока надеждност и автоматизация на процесите.

### **Участващи страни и нуждите им:**

| **Роля** | **Нужди** |
| --- | --- |
| **Собственик** | Централен контрол върху хотелите, достъп до ключови справки и права за мениджърите |
| **Мениджър** | Управление на служители и достъп до статистики за конкретния хотел |
| **Рецепционист** | Лесна работа с резервации и клиенти, добавяне на услуги |
| **Клиент** | Получаване на качествено обслужване и персонализирани услуги |

**2.2.** Проучване и избор на технологии за реализация съвместими с Java и JavaFX

В проектът са включени множество външни библиотеки, които предоставят вече съществуващи функционалности или такива, които биха направили работата по проекта по-лесна, те са:

* Lombok – Lombok библиотеката за Java, която предоставя анотации, които автоматично генерират код за общи операции като гетъри, сетъри, методи за toString(), equals() и hashCode(), както и други.

Целта на Lombok е да опрости и улесни писането на по-кратък и по-четлив код.

Ето някои от основните анотации, предоставяни от Lombok и това, което те постигат:

import lombok.Getter;  
import lombok.Setter;  
  
@Getter @Setter  
public class MyClass {  
 private int myField;  
}

public class MyClass {  
 private int myField;  
  
 public int getMyField() {  
 return myField;  
 }  
  
 public void setMyField(int myField) {  
 this.myField = myField;  
 }  
}

-----------------------------------------------------------------------------

import lombok.ToString;  
  
@ToString  
public class MyClass {  
 private int myField;  
}

public class MyClass {  
 private int myField;  
  
 @Override  
 public String toString() {  
 return "MyClass{" +  
 "myField=" + myField +  
 '}';  
 }  
}

-----------------------------------------------------------------------------

import lombok.EqualsAndHashCode;  
  
@EqualsAndHashCode  
public class MyClass {  
 private int myField;  
}

public class MyClass {  
 private int myField;  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 MyClass myClass = (MyClass) o;  
 return myField == myClass.myField;  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(myField);  
 }  
}

-----------------------------------------------------------------------------

import lombok.NoArgsConstructor;  
  
@NoArgsConstructor  
public class MyClass {  
 private int myField;  
}

public class MyClass {  
 private int myField;  
  
 public MyClass() {  
 }  
}

-----------------------------------------------------------------------------

import lombok.RequiredArgsConstructor;  
  
@RequiredArgsConstructor  
public class MyClass {  
 private final int myField;  
}

public class MyClass {  
 private final int myField;  
  
 public MyClass(int myField) {  
 this.myField = myField;  
 }  
}

-----------------------------------------------------------------------------

import lombok.AllArgsConstructor;  
  
@AllArgsConstructor  
public class MyClass {  
 private int field1;  
 private String field2;  
}

public class MyClass {  
 private int field1;  
 private String field2;  
  
 public MyClass(int field1, String field2) {  
 this.field1 = field1;  
 this.field2 = field2;  
 }  
}

-----------------------------------------------------------------------------

import lombok.Data;  
  
@Data  
public class MyClass {  
 private int myField;  
}

import java.util.Objects;  
  
public class MyClass {  
 private int myField;  
  
 public int getMyField() {  
 return myField;  
 }  
  
 public void setMyField(int myField) {  
 this.myField = myField;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 MyClass myClass = (MyClass) o;  
 return myField == myClass.myField;  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(myField);  
 }  
  
 @Override  
 public String toString() {  
 return "MyClass{" +  
 "myField=" + myField +  
 '}';  
 }  
}

-----------------------------------------------------------------------------

import lombok.Builder;  
  
@Builder  
public class MyClass {  
 private int field1;  
 private String field2;  
}

public class MyClass {  
 private int field1;  
 private String field2;  
  
 private MyClass(Builder builder) {  
 this.field1 = builder.field1;  
 this.field2 = builder.field2;  
 }  
  
 public int getField1() {  
 return field1;  
 }  
  
 public String getField2() {  
 return field2;  
 }  
  
 public static Builder builder() {  
 return new Builder();  
 }  
  
 public static class Builder {  
 private int field1;  
 private String field2;  
  
 private Builder() {  
 }  
  
 public Builder field1(int field1) {  
 this.field1 = field1;  
 return this;  
 }  
  
 public Builder field2(String field2) {  
 this.field2 = field2;  
 return this;  
 }  
  
 public MyClass build() {  
 return new MyClass(this);  
 }  
 }  
}

-----------------------------------------------------------------------------

@Slf4j  
public class MyClass {  
 public void exampleMethod() {  
 *log*.info("Logging with Lombok!");  
 }  
}

import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
public class MyClass {  
 private static final Logger *log* = LoggerFactory.*getLogger*(MyClass.class);  
  
 public void exampleMethod() {  
 *log*.info("Logging without Lombok!");  
 }  
}

* PostGreSQL - Библиотеката org.postgresql:postgresql представлява JDBC (Java Database Connectivity) драйвер за PostgreSQL, една от популярните релационни бази от данни. JDBC драйверите се използват в Java приложенията, за да установят връзка и да взаимодействат с релационни бази данни, като например PostgreSQL.

Ето някои ключови аспекти на тази библиотека:

1. **Установяване на Връзка (Establishing Connection):** JDBC драйверите позволяват на Java приложенията да се свързват с бази от данни. Този драйвер специално е предназначен за връзка с PostgreSQL бази от данни.

import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;  
  
public class ExampleClass {  
 public static void main(String[] args) throws SQLException {  
 String jdbcUrl = "jdbc:postgresql://localhost:5432/mydatabase";  
 String username = "myuser";  
 String password = "mypassword";  
  
 Connection connection = DriverManager.*getConnection*(jdbcUrl, username, password);  
 }  
}

1. **Изпълнение на Заявки (Executing Queries):** След като е установена връзка, драйверът позволява на приложението да изпраща SQL заявки към базата данни.

import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.sql.Statement;  
  
public class ExampleClass {  
 public static void main(String[] args) throws SQLException {  
 String sql = "SELECT \* FROM mytable";  
 try (Statement statement = connection.createStatement();  
 ResultSet resultSet = statement.executeQuery(sql)) {  
 *// Обработка на резултатите* }  
 }  
}

1. **Транзакции и Управление на Базата Данни (Transactions and Database Management):** JDBC драйверът позволява управление на транзакции и изпълнение на операции за управление на базата данни.

import java.sql.SQLException;  
  
public class ExampleClass {  
 public static void main(String[] args) {  
 try {  
 connection.setAutoCommit(false);  
 *// Изпълнение на операции* connection.commit();  
 } catch (SQLException e) {  
 connection.rollback();  
 *// Обработка на грешката* } finally {  
 connection.setAutoCommit(true);  
 }  
  
 }  
}

1. **Използване на Prepared Statements:** За подобряване на ефективността и безопасността, драйверът поддържа използването на подготвени заявки.

import java.sql.PreparedStatement;  
import java.sql.SQLException;  
  
public class ExampleClass {  
 public static void main(String[] args) throws SQLException {  
 String sql = "INSERT INTO mytable (column1, column2) VALUES (?, ?)";  
 try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {  
 preparedStatement.setString(1, "value1");  
 preparedStatement.setInt(2, 42);  
 preparedStatement.executeUpdate();  
 }  
 }  
}

Тази библиотека предоставя основните средства, необходими за свързване и взаимодействие с PostgreSQL база данни в Java приложения. Използването на JDBC драйвери е стандартен начин за комуникация между Java и релационни бази данни.

* JavaFX - Библиотеката **org.controlsfx:controlsfx** е библиотека за JavaFX, която предоставя допълнителни и напълно настраиваеми контроли и компоненти за създаване на графични потребителски интерфейси (GUI). Тя е предназначена да разшири стандартните възможности на JavaFX, като предоставя допълнителни контроли и инструменти за създаване на по-богати и сложни потребителски интерфейси.

Ето някои от възможностите и компонентите, предоставени от ControlsFX:

1. **Диалогови Прозорци (Dialogs):**

Позволява лесно създаване на диалогови прозорци с различни стилове и ефекти. Предоставя готови диалогови прозорци за потвърждение, предупреждение, информация и др.

import org.controlsfx.dialog.Dialogs;  
  
public class ExampleClass {  
 public static void main(String[] args) {  
  
 Dialogs.create()  
 .title("Information Dialog")  
 .masthead("Look, an Information Dialog")  
 .message("I have a great message for you!")  
 .showInformation();  
 }  
}

1. **Календар (CalendarPicker):**

Позволява избор на дата от календар с интуитивен и удобен начин.

import org.controlsfx.control.CalendarPicker;  
  
public class ExampleClass {  
 CalendarPicker calendarPicker = new CalendarPicker();  
}

1. **Autocomplete TextField:**

Поле за текст, което предлага автоматично допълване на въведените данни.

import org.controlsfx.control.textfield.AutoCompletionTextFieldBinding;  
import org.controlsfx.control.textfield.TextFields;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class ExampleClass {  
 List<String> data = Arrays.*asList*("Java", "JavaScript", "Python", "C++");  
 TextFields.bindAutoCompletion(textField, data);  
}

1. **Task Progress View:**

Контрол за визуализация на прогреса на фонови задачи.

import org.controlsfx.control.TaskProgressView;  
  
public class ExampleClass {  
  
 TaskProgressView<Task<?>> taskProgressView = new TaskProgressView<>();  
taskProgressView.getTasks().addAll(task1, task2, task3);  
  
}

1. **Master-Detail View:**

Предоставя контрол за лесно създаване на интерфейси с master-detail структура.

import org.controlsfx.control.MasterDetailPane;  
  
  
public class ExampleClass {  
 public static void main(String[] args) {  
 MasterDetailPane masterDetailPane = new MasterDetailPane();  
 masterDetailPane.setMasterNode(masterNode);  
 masterDetailPane.setDetailNode(detailNode);  
 }  
}

1. **CheckComboBox:**

Позволява на потребителя да избира от списък със селекция на отметки.

import javafx.collections.FXCollections;  
import org.controlsfx.control.CheckComboBox;  
  
import java.util.Arrays;  
import java.util.List;  
  
  
public class ExampleClass {  
 public static void main(String[] args) {  
 List<String> items = Arrays.*asList*("Option 1", "Option 2", "Option 3");  
 CheckComboBox<String> checkComboBox = new CheckComboBox<>(FXCollections.*observableArrayList*(items));  
 }  
}

1. **InfoOverlay:**

Контрол за показване на информация върху друг контрол.

import org.controlsfx.control.InfoOverlay;  
  
import javax.swing.text.html.ImageView;  
  
public class ExampleClass {  
 public static void main(String[] args) {  
  
 InfoOverlay infoOverlay = new InfoOverlay();  
 infoOverlay.setGraphic(new ImageView(image));  
 infoOverlay.setText("This is additional information");  
 }  
}

Това са само някои от примерите за използване на компоненти от библиотеката ControlsFX. В реалния код на вашето приложение бихте добавили необходимите импорти и интегрирали тези компоненти във вашия потребителски интерфейс.

2.3. Структура на проекта (Анализ и избор на архитектурен модел)

**Описание на Архитектурния Модел - MVC**

**MVC (Model-View-Controller)** е архитектурен модел, който има за цел да раздели компонентите на софтуерната система на три основни части, които изпълняват различни роли в системата: Model (Модел), View (Изглед) и Controller (Контролер).

1. Model (Модел):

* **Описание:**
  + Моделът представлява данните и бизнес логиката на приложението. Той съдържа структури от данни, методи и бизнес логика, отговарящи за обработката на информацията.

2. View (Изглед):

* **Описание:**
  + Изгледът е отговорен за представянето на информацията на потребителя. Това включва графични елементи, форми, таблици и всичко, което потребителят ви вижда.
  + Използвайки JavaFX, вие бихте създали графичен интерфейс, който представя информацията от модела на потребителя.

3. Controller (Контролер):

* **Описание:**
  + Контролерът действа като посредник между модела и изгледа. Той приема вход от потребителя, обработва го и актуализира модела или изгледа според необходимостта.
  + В JavaFX, контролерът ще съдържа логика за обработка на събития (например бутони, текстови полета) и актуализация на модела и изгледа.

Обща Работа и Поток на Данни:

* **Взаимодействие:**
  + Контролерът получава вход от потребителя и извиква съответните методи в модела за обработка на данни.
  + Моделът може да уведомява контролера за промени чрез шаблона Observer или по друг начин.
  + Контролерът обновява изгледа, който отразява промените в модела.
* **Поток на Данни:**
  + **Изглед → Контролер → Модел:** Получаване на вход от потребителя, обработка и пренасяне на необходимите данни към модела.
  + **Модел → Контролер → Изглед:** Проактуализиране на изгледа чрез предоставяне на актуална информация от модела.

**Обосновка на Избора на MVC Архитектурен Модел**

Изборът на MVC (Model-View-Controller) архитектурен модел за проекта ни е основан на редица предимства и аспекти, които подобряват ефективността, поддръжката и разширяемостта на приложението.

1. **Разделяне на Отговорности:**

MVC разделя компонентите на проекта ви на три отделни, но взаимодействащи части - модел, изглед и контролер. Този принцип на разделяне на отговорности ви дава следните предимства:

* **Поддържане на Чист Код:**
  + Всяка част на приложението има своя четимост и лесно разбираем код, който е поддържаем и подходящ за промени.
* **Лесна Разработка на Екип:**
  + Разработчиците могат да работят паралелно върху различни части от системата без да се препъват по драските.

2. **Лесна Промяна на Изгледа:**

Един от ключовите аспекти на архитектурата MVC е възможността за промяна в графичния интерфейс (изгледа) без да се нарушава бизнес логиката в модела. Това включва:

* **Гъвкавост при Дизайна:**
  + Възможността за да променяте и подобрявате графичния интерфейс без да се засяга кодът, отговарящ за обработката на данни и бизнес логиката.

3. **Повторна Употреба на Компоненти:**

MVC позволява лесна повторна употреба на компоненти, като:

* **Модулярност:**
  + Моделът и контролерът могат да се използват повторно в различни части на приложението или дори в бъдещи проекти, което подобрява ефективността и бързината на разработката

5. **Съвместимост с JavaFX и PostGreSQL:**

MVC се интегрира лесно с технологии като JavaFX, Hibernate и PostGreSQL:

* **JavaFX за Интерфейса:**
  + Използването на JavaFX за създаване на графичен интерфейс отговаря на модерните стандарти за UI разработка в Java.

* + **PostGreSQL за Базата Данни:**
  + Изборът на PostGreSQL осигурява стабилна и мощна релационна база от данни.

6. **Контрол върху Потока на Данни:**

С MVC имате по-голям контрол върху потока на данни в приложението:

* **Централизиран Контрол:**
  + Контролерът централизира логиката за обработка на входящите данни и решава как да се актуализира моделът и изгледа.

2.4. Дефиниция на модулите на системата

**Модули и компоненти**

Нашата апликация е организирана в три основни модула: **backend**, **frontend**, и **database**. Всеки от тези модули е разделен на различни компоненти, които изпълняват специфични роли. По-долу е подробно обяснение на структурата и целите на всяка част.

1. **Backend:**

* **Services:**
  + **Описание:** Този модул съдържа бизнес логика, предоставена чрез различни услуги (services).
  + **Цел:** Осигурява високо ниво на абстракция и обединява логиката, необходима за взаимодействие с базата данни и обработка на данни от frontend.

2. **Database:**

* **Object:**
  + **Описание:** Модели на апликацията.
  + **Цел:** Предоставяне на структура и дефиниране на обекти, които отразяват бизнес обекти в базата данни.
* **Data:**
  + **Описание:** DAO (Data Accessable Object) класове, съдържащи операции за работа с моделите и базата данни.
  + **Цел:** Предоставя удобен интерфейс за достъп и манипулиране на данни в базата данни.
* **Enums:**
  + **Описание:** Колекция от константи, представляващи различните състояния на entity-тата.
  + **Цел:** Централизира и документира различните стойности, които могат да бъдат асоциирани със състоянията на моделите.

3. **Frontend:**

* **Controllers:**
  + **Описание:** Контролери, свързани със съответните FXML файлове, които управляват визуалния интерфейс на потребителя.
  + **Цел:** Изолира бизнес логиката от потребителския интерфейс и осигурява лесен начин за свързване на различни UI елементи.
* **Screens:**
  + **Описание:** Помощни класове и методи, които обслужват различни части от frontend.
  + **Цел:** Поддържа обща логика, която се използва от различни компоненти на frontend.

Този модулен и структуриран подход осигурява четимост, лесна поддръжка и разширяемост на нашата апликация. Всеки модул изпълнява специфични функции, като гарантира отделност на отговорностите и подобрява преизползваемостта на кода.

**Глава. 3 Проектиране на системата**

3.1. Проектиране на отделните модули (бази данни, бизнес логика, презентационен слой)

**База данни:**

Пакетът "object" предоставя основните класове или ентитети, които представят основните обекти в системата. Тези класове отразяват структурата и характеристиките на данните, които се използват в приложението. Класовете в този пакет играят ключова роля в представянето и съхранението на данни в базата данни.

Пакетът "enums" съдържа перечисления (enum-и), които представят различни стойности с ограничен набор. Тези enum-и се използват за категоризиране и идентифициране на различни аспекти в системния модел. Включват се 6 enum класа: **ReservationEndType, ReservationType, RoomStatus , RoomType , UserType, Ratings**

1. **ReservationEndType - Начина, по който клиент е приключил резервацията. Спрямо него се намаля или вдига рейтинга на даден гост.**

**Enum Константи:**

***GOOD*, +0.5**

***MID*, +0.0**

***BAD -0.5***

1. **ReservationType - Типовете на резервация: all Inclusive, Ultra Inclusive, Non Inclusive. Спрямо тях се добавя стойност към крайната цена.**

***ALL*, +50$**

***ULTRA*, +100$**

***NON +0$***

3.**RoomStatus - Дали стаята е заета, свободна или блокирана (поради почистване или др.)**

***OCCUPIED*,**

***UNOCCUPIED*,**

***BLOCKED***

**4.RoomType - Тип на стаи, места, лукс. Всеки тип има различна цена на вечер.**

***SINGLE*(70),**

***DOUBLE*(100),**

***FAMILY*(150),**

***PENTHOUSE*(300);**

Полета:

private final int pricePerNight;

// Constructor to accept price

RoomType(int pricePerNight) {

this.pricePerNight = pricePerNight;

}

5.**UserType - Определя ролята на потребител в приложението.**

***OWNER*,**

***MANAGER*,**

***RECEPTIONIST*,**

***ADMIN***

6. Ratings - Енумерация за рейтингите на клиентите:

*NONE*,

*ONE*,

*TWO*,

*THREE*,

*FOUR*,

*MAX*

Пакетът “data” съдържа всички DAO класове и обекти. В неговите класове се извършват всички заявки към базата данни. Класове:

**AdminUserDAO, ClientDAO, DatabaseConnection, HotelDAO, ManagerDAO, OwnerDAO, ReceptionistDAO, ReservationDAO, RoomDAO.**

1. **AdminUserDAO**.

Методи:

1. authenticateUser(String username, String password) - Автентикация при логване в системата
2. getLoggedInUser(String username) - събиране на данни за логнат потребител.

2. **ClientDAO**

Методи:

1. saveClientToDb - Прави нов запис за клиент в базата данни.

3. **DatabaseConnection**

Полета:

private static final String *DB\_URL* = "jdbc:postgresql://localhost:\*\*\*/hmsbase";

private static final String *USER* = "p\*\*\*\*";

private static final String *PASSWORD* = "\*\*\*";

Методи:

1. connect() - Осъществява конекция с базата данни.

**HotelDAO** **Методи:**

* insertHotel(Hotel hotel) – Добавя нов хотел в базата данни. (Методът е дефиниран, но все още няма имплементация)
* authenticateHotel(String username) – Проверява дали съществува хотел с дадено име (обикновено за избягване на дублиране при регистрация).

**ManagerDAO** **Методи:**

* authenticateUser(String username, String password) – Автентикация на мениджър при логване в системата. Проверява дали съществува мениджър с дадените потребителско име и парола.
* insertManager(Manager manager) – Добавя нов мениджър в базата данни, използвайки данните от обект от тип Manager.
* getLoggedInUser(String username) – Извлича информацията за логнат мениджър по подадено потребителско име и я връща като обект от тип Manager.

**OwnerDAO** **Методи:**

* authenticateUser(String username, String password) – Автентикация на собственик (Owner) при логване в системата. Проверява дали съществува запис с подадените потребителско име и парола в таблицата owners.
* insertOwner(Owner owner) – Добавя нов собственик в базата данни, използвайки информацията от обекта Owner.
* getLoggedInUser(String username) – Извлича данните за логнат собственик по подадено потребителско име и ги връща като обект от тип Owner.

**ReceptionistDAO** **Методи:**

* authenticateUser(String username, String password) – Автентикация на рецепционист (Receptionist) при логване в системата. Проверява дали съществува запис с подадените потребителско име и парола в таблицата receptionists.
* getLoggedInUser(String username) – Извлича данните за логнат рецепционист по подадено потребителско име и ги връща като обект от тип Receptionist.

**ReservationsDAO**

**Методи (примерни):**

* insertReservation(Reservation reservation) – Добавя нова резервация в базата данни.
* getReservationsByHotelId(int hotelId) – Връща списък с резервации за даден хотел по неговото ID.
* getReservationsByReceptionistId(int receptionistId) – Връща списък с резервации, направени от конкретен рецепционист.
* cancelReservation(int reservationId) – Отменя съществуваща резервация по нейното ID.
* updateReservation(Reservation reservation) – Актуализира данните за съществуваща резервация.
* getReservationById(int reservationId) – Връща конкретна резервация по ID.

### **RoomDAO**

**Методи :**

* insertRoom(Room room)  
   → Добавя нова стая в базата данни.
* getRoomsByHotelId(int hotelId)  
   → Връща списък със стаи, свързани с даден хотел по ID.
* getAvailableRooms(int hotelId, LocalDate checkIn, LocalDate checkOut)  
   → Връща списък със свободни стаи в хотела за определен период.
* deleteRoom(int roomId)  
   → Изтрива стая от базата данни по ID.
* updateRoom(Room room)  
   → Актуализира данните на стая (тип, цена, заетост и т.н.).
* getRoomById(int roomId)  
   → Връща информация за конкретна стая по нейното ID.

Пакетът Services служи за обединяване на различни DAO класове и свързва тях с дадените контролери, чрез логика.

1.AdminControllerServices

* getMenuScene() - Метод за показване на нов екран/сцена.

2. HotelService

* saveHotel(Hotel hotel) - Изпраща цял обект от “Хотел” Към базата данни за запазване, присвоява Hotel\_Id на стаите създадени към дадения хотел.

3. LoginControllerServices

* Полета:

private OwnerDAO ownerDAO = new OwnerDAO();

private ManagerDAO managerDAO = new ManagerDAO();

private ReceptionistDAO receptionistDAO = new ReceptionistDAO();

private AdminUserDAO adminUserDAO = new AdminUserDAO();

private final String ownerMenu = "/com/example/hotelmanager/screens/OwnerMenuScreen.fxml";

private final String managerMenu = "/com/example/hotelmanager/screens/ManagerMenuScreen.fxml";

private final String receptionistMenu = "/com/example/hotelmanager/screens/ReceptionistMenuScreen.fxml";

private final String adminMenu = "/com/example/hotelmanager/screens/AdminMenuScreen.fxml";

* getMainScene(UserType userType) - Срямо типа на логнатия USER, се избира начален прозорец за зареждане.
* verifyLogin(String username, String password, UserType userType) - Проверява типа на логнатия USER и прави справка с базата данни дали съществува в таблиците.
* fetchUser(String username, UserType userType) - Взима данни за влезлия USER.

4. OwnerControllerServices

* returnScene()
* getHotelCreationScene()
* getManagerCreationScene()

5. RoomService

* saveRoom(Room room, int hotelId) - Запазва стая към създадения хотел.

6. UserService

* getAllManagers() - Взима всички мениджъри в базата данни и ги добавя в List<Manager>.

Пакет Controllers, в него са всички контролери за FXML сцените. Използва се за контролиране на визуалните елементи и обекти.

1. AdminMenuScreenController

* openCreateOwner(ActionEvent event) - Отваря ново меню, за създаване на собственици

2. HotelCreationController

* initialize() - Пълни страницата с данни.
* addRoomDialog() - Създава диалогов прозорец за въвеждане на стаи към хотел.
* createHotel() - Създава хотел с мениджър и собственик

3. LoginScreenController

* initialize()
* handleLogin(ActionEvent actionEvent) - Събира информация от полетата и препраща за обработка.

4. ManagerCreationController

* initialize()
* validateFields() - Проверка за попълнени полета

private boolean isValidName(String name) {

return name.matches("[a-zA-Z]+");

}

private boolean isValidDateOfBirth(LocalDate dob) {

return dob != null && dob.isBefore(LocalDate.*now*());

}

private boolean isValidPhoneNumber(String phoneNumber) {

return phoneNumber.matches("\\d{10}");

}

private boolean isValidUsername(String username) {

return username.matches("[a-zA-Z0-9]{1,12}");

}

private boolean isValidPassword(String password) {

return password.length() > 5;

}

* showFeedback(String message, Color color) - Обратна връзка
* clearFields() - Чисти полетата
* onClick(ActionEvent actionEvent) - Събира информация от полетата, след това валидира информацията и създава нов обект от Мениджър, който добавя към базата данни.

5. ManagerMenuController

* openReceptionistRegistration(ActionEvent event) - Зарежда сцената за регистрация на рецепционист.

6. OwnerCreationController

* initialize()
* validateFields() - Валидация на полета

private boolean isValidName(String name) {

return name.matches("[a-zA-Z]+");

}

private boolean isValidDateOfBirth(LocalDate dob) {

return dob != null && dob.isBefore(LocalDate.*now*());

}

private boolean isValidPhoneNumber(String phoneNumber) {

return phoneNumber.matches("\\d{10}");

}

private boolean isValidUsername(String username) {

return username.matches("[a-zA-Z0-9]{1,12}");

}

private boolean isValidPassword(String password) {

return password.length() > 5;

}

* showFeedback(String message, Color color) - Обратна връзка
* clearFields() - Чисти полетата
* onClick(ActionEvent actionEvent) - Прави валидация на полетата и след това създава обект от Owner, който пълни в базата данни.

7. OwnerMenuScreenController

* openCreateManager(ActionEvent event) - Зарежда меню за създаване на мениджър.
* openCreateHotel(ActionEvent event) - Зарежда меню за създаване на хотел и стаи.
* setScene(Scene scene, ActionEvent event) - Изпълнява роля на Loader.

8. PersonFormController

* add() - Запазва клиент в базата данни
* isValidInput() - Валидиране на данни
* showErrorMessage(String message) - нз
* cancel() - Връща назад

9. ReceptionistMenuController

10. RoomTileController

Пакет Screens - Всички прозорци и сцени използвани в програмата:

1. **AdminMenuScreen.fxml**
2. **HotelCreationScreen.fxml**
3. **LoginScreen.fxml**
4. **ManagerCreationScreen.fxml**
5. **ManagerMenuScreen.fxml**
6. **OwnerCreationScreen.fxml**
7. **OwnerMenuScreen.fxml**
8. **PersonForm.fxml**
9. **ReceptionistMenuScreen.fxml**
10. **RoomTile.fxml**

Пакет CSS - Стилизирането на екраните и визуалните елементи:

**AdminMenuCss.css**

**hotel\_creation.css**

**LoginCss.css**

**manager\_creation.css**

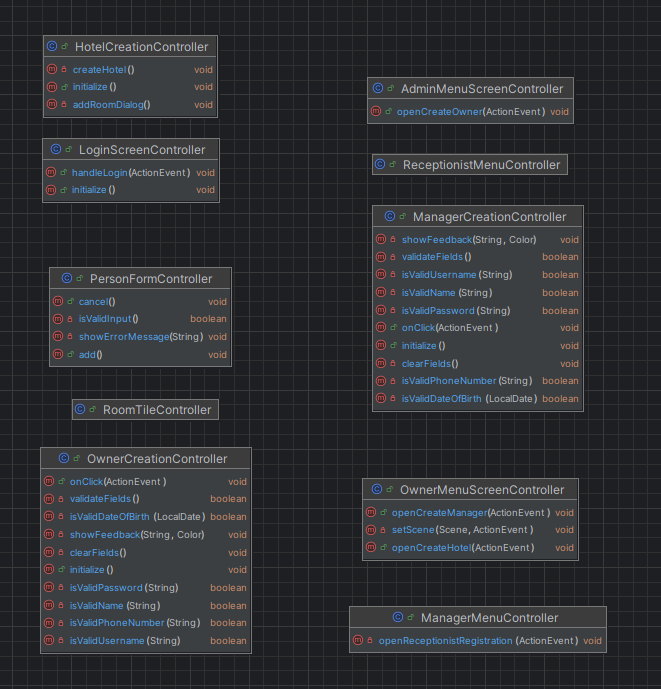
**ManagerMenu.css**

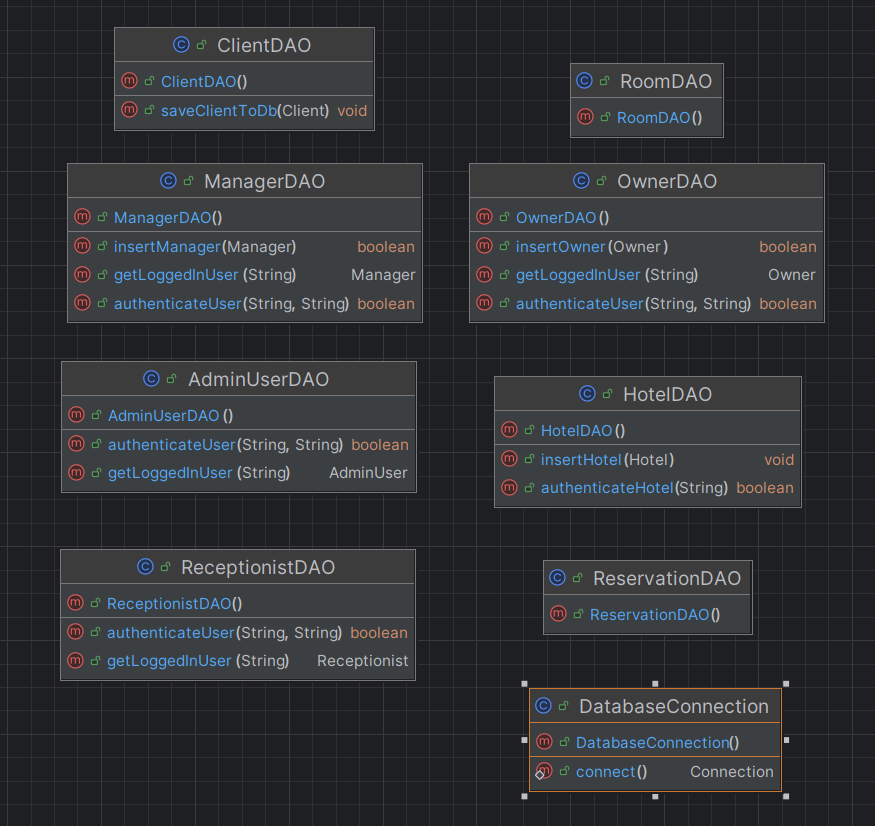
**owner\_creation.css**

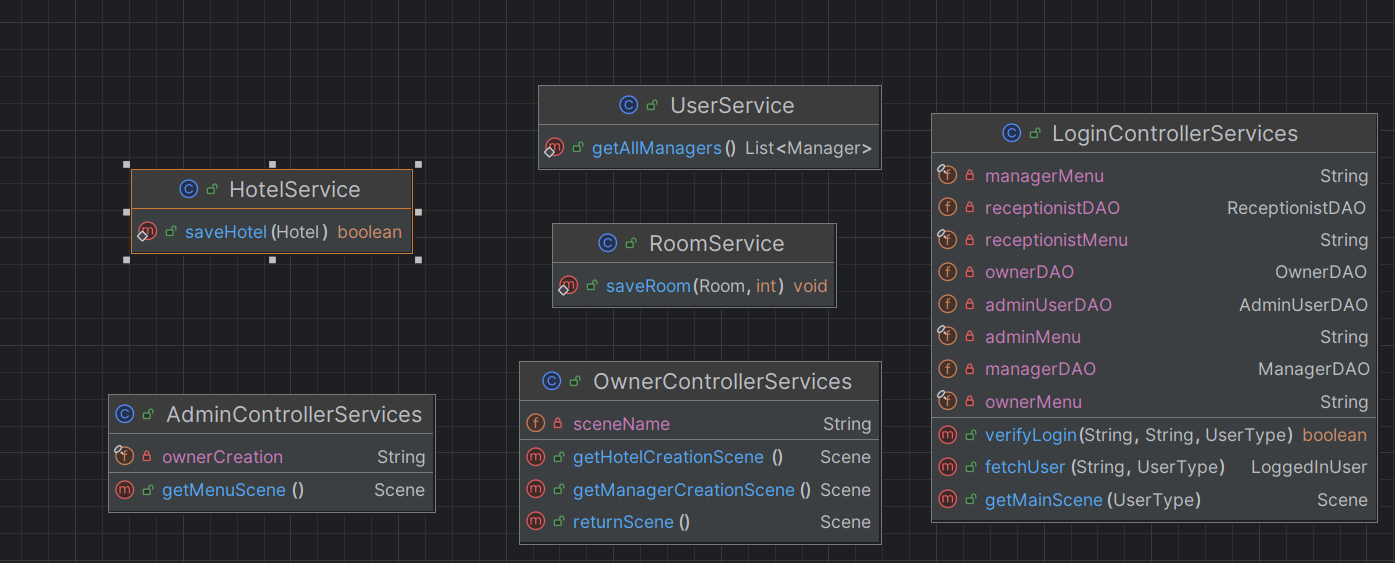
**OwnerMenuCss.css**

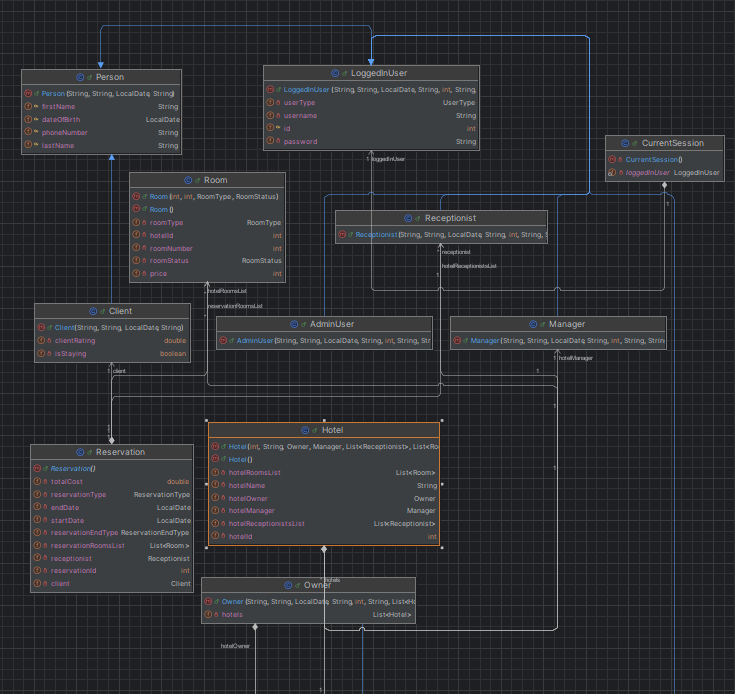
**ReceptionistMenuCss.css**

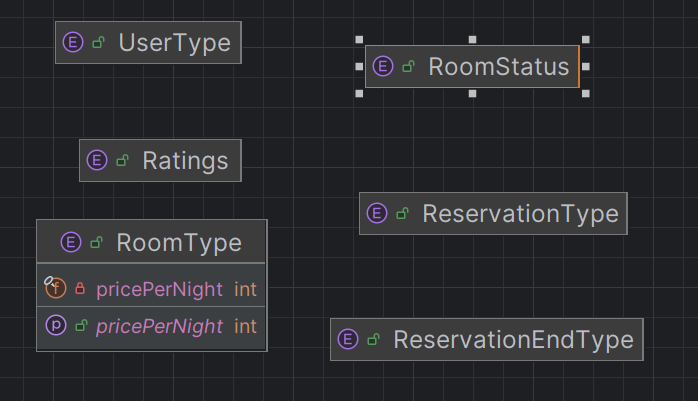
3.2 Diagrams:

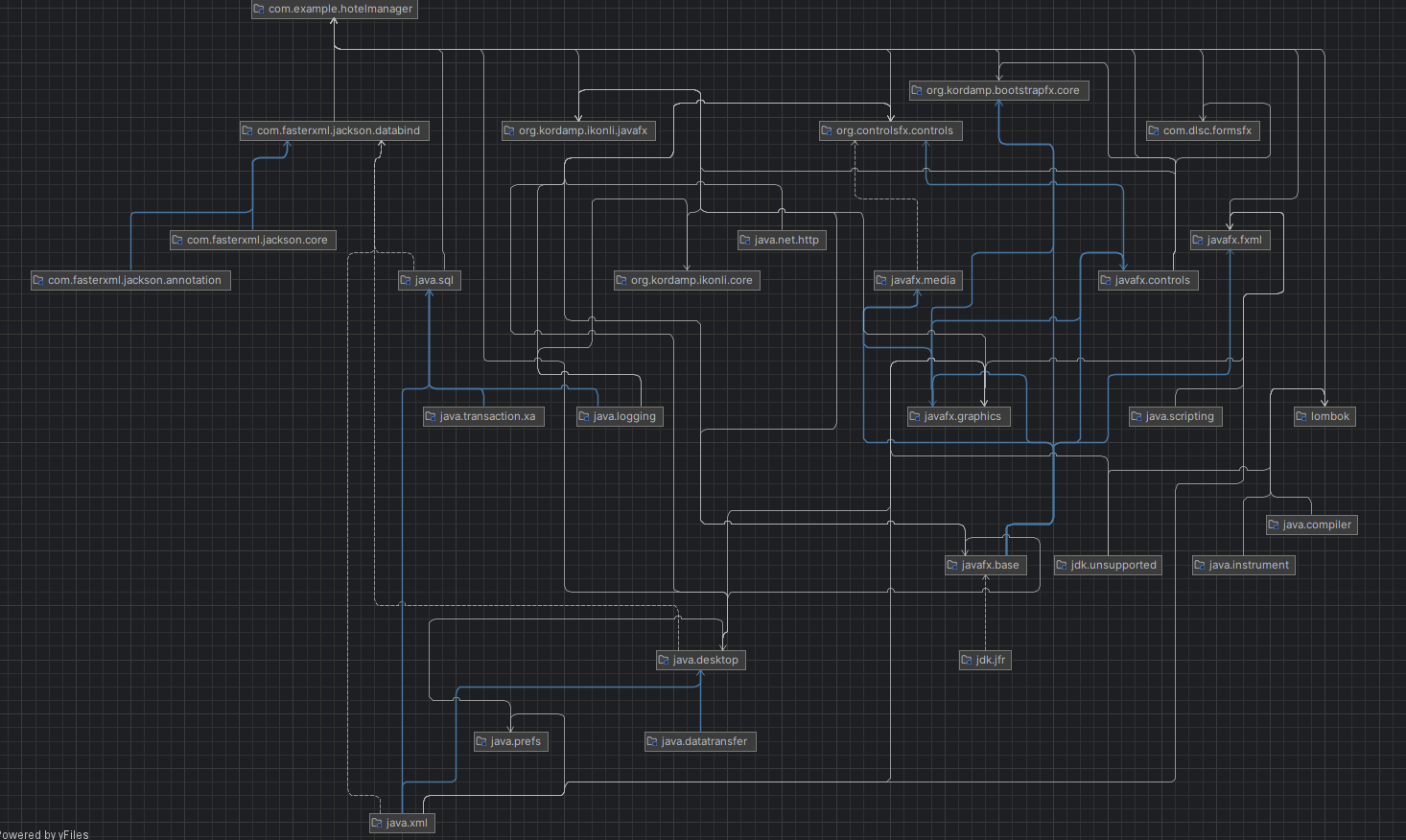






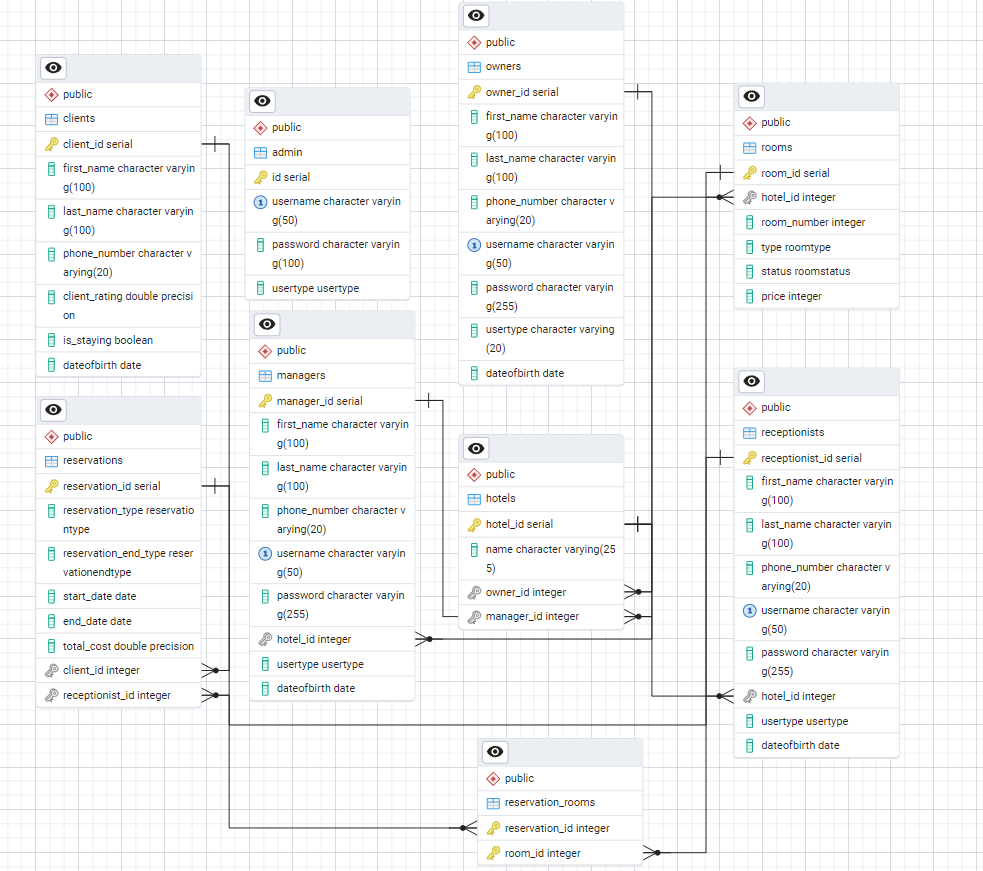






**Глава 4. Реализация на системата**

**4.1.Реализация на базата от данни (PostGresSQL) – Релационна схема и описание на таблиците**



### **Entity: Client**

Описва клиент, който прави резервации.

* **Атрибути:**
  + client\_id (PK)
  + first\_name
  + last\_name
  + phone\_number
  + client\_rating
  + is\_staying
  + dateofbirth

### **Entity: Admin**

Представлява администраторски потребител.

* **Атрибути:**
  + id (PK)
  + username
  + password
  + usertype

### **Entity: Owner**

Собственик на един или повече хотели.

* **Атрибути:**
  + owner\_id (PK)
  + first\_name
  + last\_name
  + phone\_number
  + username
  + password
  + usertype
  + dateofbirth

### **Entity: Manager**

Управител на хотел.

* **Атрибути:**
  + manager\_id (PK)
  + first\_name
  + last\_name
  + phone\_number
  + username
  + password
  + hotel\_id (FK → Hotel)
  + usertype
  + dateofbirth

### **Entity: Receptionist**

Рецепционист, назначен в хотел.

* **Атрибути:**
  + receptionist\_id (PK)
  + first\_name
  + last\_name
  + phone\_number
  + username
  + password
  + hotel\_id (FK → Hotel)
  + usertype
  + dateofbirth

### **Entity: Hotel**

Хотел, който има собственик, мениджър, рецепционисти, стаи.

* **Атрибути:**
  + hotel\_id (PK)
  + name
  + owner\_id (FK → Owner)
  + manager\_id (FK → Manager)

### **Entity: Room**

Стая в хотел.

* **Атрибути:**
  + room\_id (PK)
  + hotel\_id (FK → Hotel)
  + room\_number
  + type
  + status
  + price

### **Entity: Reservation**

Резервация, направена от клиент, обработена от рецепционист.

* **Атрибути:**
  + reservation\_id (PK)
  + reservation\_type
  + reservation\_end\_type
  + start\_date
  + end\_date
  + total\_cost
  + client\_id (FK → Client)
  + receptionist\_id (FK → Receptionist)

### **Entity: Reservation\_Rooms**

Свързва резервации със стаи (M:N).

* **Атрибути:**
  + reservation\_id (FK → Reservation)
  + room\_id (FK → Room)

## **Връзки между ентитетите**

* **Owner (1) — (N) Hotel** Един собственик може да притежава няколко хотела.
* **Manager (1) — (1) Hotel** Всеки хотел има един мениджър (по manager\_id). Един мениджър отговаря за един хотел.
* **Receptionist (1) — (1) Hotel** Всеки рецепционист е назначен към един хотел.
* **Hotel (1) — (N) Room** Един хотел има много стаи.
* **Client (1) — (N) Reservation** Един клиент може да направи множество резервации.
* **Receptionist (1) — (N) Reservation** Една резервация се обработва от един рецепционист.
* **Reservation (M) — (N) Room** Една резервация може да включва много стаи, и една стая може да участва в много резервации (чрез reservation\_rooms).

**Глава 5. Тестове**

## **Заключение**

Системата **Hotel Management System** е цялостно софтуерно решение, създадено да автоматизира и оптимизира управлението на хотелския бизнес. Проектирана с ясна структура, модулност и гъвкавост, тя осигурява ефективна координация между различните типове потребители – собственици, мениджъри, рецепционисти и клиенти. Чрез интуитивен потребителски интерфейс и стабилна връзка с база данни, системата предоставя надеждни функционалности като регистрация на потребители, управление на хотели, резервации и стаи, както и проследяване на клиентската активност.

Използването на технологии като JavaFX, PostgreSQL и добре организирана обектно-ориентирана архитектура позволява системата да бъде лесно разширяема и поддържаща се. Предвидената роля-базирана достъпност гарантира сигурност и контрол върху действията на всеки потребител.

Разработката на **Hotel Management System** не само демонстрира прилагането на добри софтуерни практики, но и предоставя стабилна основа за бъдещо надграждане, като включване на допълнителни модули – онлайн резервации, плащания, отчети и анализи.

С тази система хотелите могат да подобрят обслужването на клиентите си, да повишат оперативната ефективност и да отговорят адекватно на динамичните изисквания на съвременния туристически пазар.