

Технически университет - Варна

Факултет: ФИТА

Катедра: СИТ

Специалност: СИТ

Дисциплина:

Обектно ориентирано програмиране проект – 2 част

Проект на тема: Библиотека

Изготвен от:

Алекс Михайлов Орозов – 21621586

Богомил Георгиев Донков – 21621611

**Структура на документацията**

**Глава 1. Увод…………………………………………………4**

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

**Глава 2. Анализ на проблема……………………………….5**

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 (Hibernate и DAO (Data Accessable Object))……………………………………………………………….82

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

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

4.5.Реализация на модул за тестване на събития в системата (Logback и SLF4J)…………………………………………………………………………….88

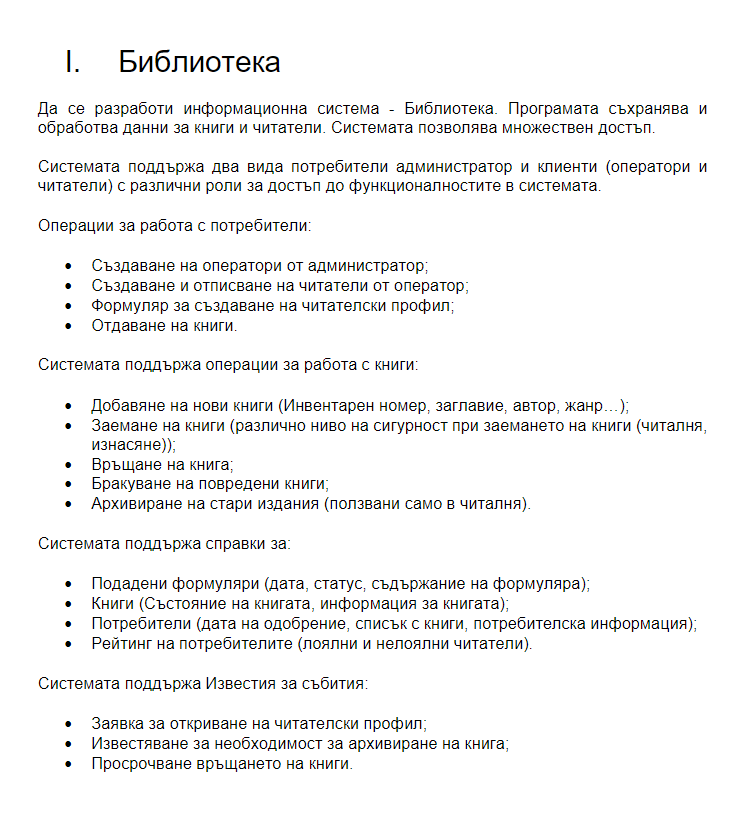
**Глава 5. Тестови резултати…………………………..........89**

5.1. Автоматизирани тестове……………………………………………………89

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

**Глава 1. Увод**

* 1. Пълно описание на проекта (Условие)



**Глава 2. Анализ на проблема**

2.1. Функционални изисквания

Системата трябва да поддържа като потребители администратори, оператори и читатели.

Администраторите са отговорни за регистрирането на нови книги в библиотеката, създаване и премахване на оператори, преглед на всички съществуващи потребители в системата, да архивира книги по негова преценка и да премахва книги със щети по тях.

Операторите трябва да бъдат способни да откриват и закриват читателски профили, да внасят книги в определено от тях количество, да преглеждат всички съществуващи потребители в системата, да отдават книги за различна употреба, да отговарят за връщане на книгите от даден читателски формуляр.

Читателите нямат никаква отговорност в системата. Те се явяват на гише при работник(оператора) и просто заявяват какви книги желаят да върнат или вземат. Операторът извършва цялата работа за тях.

Системата поддържа методи за известяване – изпращане на имейл на потребителка електронна поща.

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!");  
 }  
}

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

* **Hibernate** – org.hibernate:hibernate-core е основната библиотека за обектно-релационно съпоставяне (ORM) в Java еко системата, която предоставя инструменти за работа с релационни бази данни чрез обекти. Hibernate улеснява извличането, обновяването и вмъкването на данни в релационни бази данни, като предоставя високо ниво на абстракция на SQL.

Ключови функционалности в Hibernate:

1. **ORM(Object-Relation Mapping):** Hibernate позволява работата с обекти в Java, които съответстват на записи в релационни бази данни. Това осигурява по-лесен и интуитивен начин за манипулиране на данни.
2. **Автоматично генериране на SQL:** Hibernate автоматично генерира SQL заявки за създаване, обновяване, извличане и изтриване на данни от базата данни, което намалява необходимостта от писане на ръчен SQL код.
3. **Кеширане:** Hibernate предоставя възможности за кеширане на данни, което подобрява производителността при честите заявки.
4. **Transaction Management:** Hibernate управлява транзакциите в рамките на приложението, което поддържа консистентността на данните.
5. **Хибриден модел на връзката между таблиците:** Hibernate поддържа както единична, така и множествена връзка между таблиците, като предоставя гъвкавост и лесна настройка на връзките.
6. **Поддръжка на наследяване:** Hibernate поддържа различни стратегии за наследяване, които позволяват лесно създаване на йерархии от класове.
7. **Поддръжка на HQL (Hibernate Query Language):** HQL предоставя алтернативен начин за извършване на заявки върху обекти, които се използват вместо SQL.
8. **Разширяемост:** Hibernate е високо разширяем и дава възможност за дефиниране на собствени стратегии и обекти.

С implementation ('org.hibernate:hibernate-core:6.2.9.Final') в проекта се добавя конкретната версия на Hibernate като зависимост. Този артефакт съдържа основния код на Hibernate, необходим за изпълнението на ORM функционалността и други ключови възможности, споменати по-горе. Този артефакт е необходим, за да може приложението ви да използва Hibernate за връзка с релационна база данни.

Примери с анотации:

* @Entity - Употреба: Анотацията се поставя над клас, за да му каже Hibernate, че този клас трябва да бъде представен като таблица в базата данни.
* @Id – Употреба: Посочва полето като основен ключ на таблицата.

import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
  
@Entity  
public class Book {  
 @Id  
 private Long id;  
 private String title;  
 private String author;  
}

* @Table – Употреба: С тази анотация може да се указва името и други свойства на таблицата, към която класът ще се съпостави.
* import jakarta.persistence.Entity;  
  import jakarta.persistence.Id;  
  import jakarta.persistence.Table;  
    
  @Entity  
  @Table(name = "library\_books")  
  public class Book {  
   @Id  
   private Long id;  
   private String title;  
   private String author;  
  }
* @GeneratedValue – Употреба: Анотацията дефинира начина, по който Hibernate генерира стойностите за основния ключ (ID).

import jakarta.persistence.\*;  
  
@Entity  
@Table(name = "library\_books")  
public class Book {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 private String title;  
 private String author;  
}

* @Column – Употреба: Позволява насрочването на свойства на колоната в базата данни.

import jakarta.persistence.\*;  
  
@Entity  
@Table(name = "library\_books")  
public class Book {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 @Column(name = "title")  
 private String title;  
 @Column(name = "book\_author")  
 private String author;  
}

Тези са само някои от основните анотации, които можете да използвате с Hibernate. Зависимо от конкретните изисквания на вашето приложение, може да бъдат използвани и други анотации.

* **Bcrypt** - "bcrypt" библиотека за Java. Това е библиотека за хеширане на пароли, която използва bcrypt алгоритъма. Bcrypt е криптографски сигурен алгоритъм за хеширане на пароли, който е проектиран да бъде забавен и използва сол (salt), което го прави устойчив на атаки срещу радуга (rainbow table attacks).

Ето кратко описание на някои от ключовите характеристики на тази библиотека:

1. **Безопастност** - Bcrypt е считан за един от най-сигурните методи за хеширане на пароли в настоящия момент. Той използва сол и включва параметър за работа (work factor), който определя колко време отнема хеширането и го прави изключително трудно за атаки.
2. **Salt(Сол)** - Bcrypt автоматично включва сол във всеки хеш, който създава. Това предотвратява успешни атаки срещу радуга, където злоумишленникът зареди предварително изчислени хешове за голям списък от обичайни пароли.
3. **Work Factor** - Параметърът за работа (work factor) е настройка, която определя колко време се изисква за създаване на хеш. Този параметър може да бъде настроен, за да се усложни атаките срещу пароли чрез използването на brute-force атаки.
4. **Проста интеграция** - Библиотеката предоставя лесен за използване API за хеширане на пароли, което я прави подходяща за интеграция в приложения.
5. **Активно развитие** - Ако версията, която използвате, е 0.10.2, това може да съобщи за активно развитие и поддръжка на библиотеката.

Пример на използване на библиотеката може да изглежда по следния начин:

import at.favre.lib.crypto.bcrypt.BCrypt;  
  
public class PasswordHashingExample {  
 public static void main(String[] args) {  
 *// Генериране на сол* String salt = BCrypt.*withDefaults*().hashToString(12, "userPassword".toCharArray());  
  
 *// Хеширане на парола със създадената сол* String hashedPassword = BCrypt.*withDefaults*().hashToString(12, "userPassword".toCharArray());  
  
 *// Проверка на парола* boolean isPasswordCorrect = BCrypt.*verifyer*().verify("userPassword".toCharArray(), hashedPassword).verified;  
  
 System.*out*.println("Salt: " + salt);  
 System.*out*.println("Hashed Password: " + hashedPassword);  
 System.*out*.println("Is Password Correct: " + isPasswordCorrect);  
 }  
}

* **Javax.mail** - Библиотеката javax.mail предоставя Java API за работа с електронна поща (email). Тя е стандартна библиотека в рамките на Java EE (Enterprise Edition) и предоставя функционалности за изпращане, получаване и обработка на електронни съобщения.

Ето някои от ключовите функционалности и ползи на библиотеката:

1. **Създаване на Електронни Съобщения (Emails):** javax.mail позволява създаването на електронни съобщения с различни свойства като тема, получатели, изпращач, прикачени файлове и текстово съдържание.
2. **Изпращане на Електронни Съобщения:** Библиотеката предоставя функционалности за изпращане на съобщения по различни протоколи като SMTP (Simple Mail Transfer Protocol).
3. **Получаване на Електронни Съобщения:** javax.mail поддържа получаването на електронни съобщения от пощенски сървъри по протоколи като POP3 (Post Office Protocol) и IMAP (Internet Message Access Protocol).
4. **Обработка на Електронни Съобщения:** Библиотеката предоставя функционалности за обработка на съдържанието на електронни съобщения, включително работа с прикачени файлове, извличане на текст и работа с хедърите на съобщението.
5. **Конфигуриране и Управление на Сървъри:** javax.mail позволява конфигуриране и управление на настройките на пощенски сървъри, където можете да зададете настройките за изпращане и получаване на съобщения.

Библиотеката javax.mail е изключително полезна за Java приложения, които имат нужда от взаимодействие с електронни съобщения, като например имейл уведомления, обработка на имейли и други сценарии, свързани с електронна поща.

Пример на употреба:

import javax.mail.\*;  
import javax.mail.internet.InternetAddress;  
import javax.mail.internet.MimeMessage;  
import java.util.Properties;  
  
public class SimpleMailSender {  
 private final String username;  
 private final String password;  
 private final Properties properties;  
  
 public SimpleMailSender(String username, String password) {  
 this.username = username;  
 this.password = password;  
  
 *// Настройка на свойствата за използване на SMTP* properties = new Properties();  
 properties.put("mail.smtp.auth", "true");  
 properties.put("mail.smtp.starttls.enable", "true");  
 properties.put("mail.smtp.host", "smtp.example.com"); *// Подменете с реалния хост* properties.put("mail.smtp.port", "587"); *// Подменете с реалния порт* }  
  
 public void sendEmail(String to, String subject, String content) throws MessagingException {  
 Session session = Session.*getInstance*(properties, new Authenticator() {  
 protected PasswordAuthentication getPasswordAuthentication() {  
 return new PasswordAuthentication(username, password);  
 }  
 });  
 Message message = new MimeMessage(session);  
 message.setFrom(new InternetAddress(username));  
 message.setRecipients(Message.RecipientType.*TO*, InternetAddress.*parse*(to));  
 message.setSubject(subject);  
 message.setText(content);  
  
 Transport.*send*(message);  
 System.*out*.println("Email sent successfully.");  
 }  
  
 public static void main(String[] args) {  
 *// Подменете със свои реални данни* SimpleMailSender mailSender = new SimpleMailSender("your\_email@example.com", "your\_password");  
  
 try {  
 mailSender.sendEmail("recipient@example.com", "Тестово съобщение", "Здравей, това е тестово съобщение!");  
 } catch (MessagingException e) {  
 e.printStackTrace();  
 }  
 }  
}

* **Google Guava API -** Guava е библиотека на Java, предоставена от Google, която предлага много полезни класове и методи, които допълват стандартната библиотека на Java. В следващите редове ще разгледаме някои от ключовите характеристики и функции на Guava:

1. **Колекции (com.google.common.collect):**

Guava предоставя много допълнителни класове за работа с колекции, които са по-удобни и мощни от тези в стандартната библиотека. Например, ImmutableList, ImmutableSet, Multimap и други.

import com.google.common.collect.ImmutableList;  
import com.google.common.collect.ImmutableSet;  
  
import java.util.List;  
import java.util.Set;  
  
public class ShowCaseClass {  
 List<String> immutableList = ImmutableList.*of*("one", "two", "three");  
 Set<Integer> immutableSet = ImmutableSet.*of*(1, 2, 3);  
  
}

1. **Работа със символни низове (com.google.common.base):**

Guava предоставя удобни методи за работа със символни низове, като например методи за проверка на null стойности и извличане на стойности.

import com.google.common.base.MoreObjects;  
  
public class ShowCaseClass {  
 String result = MoreObjects.*firstNonNull*(input, "default");  
}

1. **Функционално програмиране (com.google.common.base):**

Guava предоставя инструменти за функционално програмиране, като Function, Predicate, Supplier и други.

import java.util.function.Function;  
  
public class ShowCaseClass {  
 Function<Integer, String> squareFunction = input -> "Square: " + (input \* input);  
}

1. **Функционални операции върху колекции (Functional Operations on Collections):** Guava предоставя методи за филтриране, трансформиране и обединяване на колекции.

import com.google.common.collect.Lists;  
  
import java.util.List;  
import java.util.stream.Collectors;  
  
public class ExampleClass {  
 List<String> filteredList = Lists.*newArrayList*("apple", "banana", "orange")  
 .stream()  
 .filter(fruit -> fruit.startsWith("a"))  
 .collect(Collectors.*toList*());  
}

1. **Задачи за изпълнение (Concurrency):** Guava предоставя инструменти за управление на конкурентността.

import com.google.common.util.concurrent.ListenableFuture;  
import com.google.common.util.concurrent.ListeningExecutorService;  
import com.google.common.util.concurrent.MoreExecutors;  
  
import java.util.concurrent.Executors;  
  
public class ExampleClass {  
 ListeningExecutorService executorService = MoreExecutors.*listeningDecorator*(Executors.*newFixedThreadPool*(10));  
 ListenableFuture<String> future = executorService.submit(() -> "Hello, Guava!");  
}

1. **Разширени структури от данни (Additional Data Structures):** Guava включва структури като Multimap, Table и други, които предоставят по-богати възможности от стандартните Java колекции.

import com.google.common.collect.ArrayListMultimap;  
import com.google.common.collect.Multimap;  
  
public class ExampleClass {  
 public static void main(String[] args) {  
 Multimap<String, String> multimap = ArrayListMultimap.*create*();  
 multimap.put("Fruits", "Apple");  
 multimap.put("Fruits", "Banana");   
 }  
}

* 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.

* **SLF4J (Simple Logging Facade for Java) –** SLF4J е прост интерфейс за логиране в Java, който предоставя общ API за различни системи за логиране. Той цели да улесни прехода между различни системи за логиране и да предостави гъвкавост при избора на конкретна имплементация.
* **Logback -** Logback е конкретна имплементация на SLF4J API и предоставя мощен и ефективен механизъм за логиране. Той включва две основни компонента - logback-classic и logback-core.
* **logback-classic:** Този модул предоставя статичен бекенд за логиране, което използва SLF4J API. Той е наследник на популярния log4j проект и предоставя подобен API за конфигуриране и употреба.
* **logback-core:** Този модул предоставя ядрена функционалност за Logback, като асинхронно логиране и филтриране на съобщения.

Функционалности:

* **Лесен за използване API:** Предоставя прост и лесен за използване API за логиране в Java приложения.
* **Бърз и ефективен:** Logback е известен със своята висока производителност и ефективност.
* **Гъвкавост:** Позволява конфигуриране на различни аспекти на логирането чрез конфигурационни файлове.
* **Поддръжка на различни нива на логиране:** Поддържа различни нива на логиране, като DEBUG, INFO, WARN, ERROR и др.
* **Способност за асинхронно логиране:** Предоставя възможност за асинхронно логиране, което може да подобри производителността.

Примерна употреба:

import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
public class MyClass {  
 private static final Logger *logger* = LoggerFactory.*getLogger*(MyClass.class);  
  
 public void exampleMethod() {  
 *logger*.info("Logging with SLF4J and Logback!");  
 }  
}

Този код демонстрира как да се използва SLF4J API за логиране в Java клас. Когато конфигурацията е направена правилно, съобщението "Logging with SLF4J and Logback!" ще бъде записано в лог файла чрез Logback.

* **JUnit 5:** JUnit е популярна библиотека за тестване в Java, предоставяща функционалност за създаване и изпълнение на тестове. Версията 5 на JUnit (JUnit Jupiter) предлага нови възможности и подобрения в сравнение с предишните версии.

Функционалности:

* **Декларативен синтаксис:** JUnit 5 предлага декларативен синтаксис за написване на тестове с анотации, като **@Test**, **@BeforeEach**, **@AfterEach** и други.
* **Rerunnable Tests:** Тестовете могат да бъдат маркирани като **@RepeatedTest** или **@ParameterizedTest** за повторно изпълнение с различни аргументи.
* **Extension Model:** Разширенията (Extensions) предоставят начин за добавяне на допълнителна функционалност към тестовете.
* **Поддръжка на Java 8+:** Използва функционалности като ламбда изрази и потоци, които са налични в Java 8 и по-нови версии.
* **Поддръжка на паралелно тестване:** Възможност за паралелно изпълнение на тестове, което подобрява ефективността на тестовите суити.

Примерен код:

import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*assertEquals*;  
  
public class MyTest {  
 @Test  
 void testAddition() {  
 *assertEquals*(4, 2 + 2, "2 + 2 should equal 4");  
 }  
}

* **TestFX -** TestFX е библиотека за функционално тестване на графични потребителски интерфейси (GUI) в JavaFX приложения. Това позволява на разработчиците да пишат тестове, които симулират действия на потребителя върху GUI и проверяват резултатите.

Примерен код:

import org.junit.jupiter.api.Test;  
import org.testfx.api.FxRobot;  
import org.testfx.framework.junit5.ApplicationTest;  
  
public class MyGuiTest extends ApplicationTest {  
 @Test  
 void testButtonClick(FxRobot robot) {  
 clickOn("#myButton");  
 *// Проверка за очакван резултат след натискане на бутон* }  
}

* **Mockito -** Mockito е библиотека за създаване и използване на мок обекти (mock objects) в Java тестове. Това позволява на разработчиците да изолират тестовете от външни зависимости и да създават контролирани среди за тестване.

Примерен код:

import org.junit.jupiter.api.Test;  
import static org.mockito.Mockito.\*;  
  
public class MyMockTest {  
 @Test  
 void testMockBehavior() {  
 *// Създаване на мок обект* MyService myServiceMock = *mock*(MyService.class);  
  
 *// Дефиниране на поведение на мок обекта  
 when*(myServiceMock.getResult()).thenReturn("Mocked Result");  
  
 *// Проверка на извикване на метод с мок обект* myServiceMock.getResult();  
  
 *// Проверка на очакван резултат  
 verify*(myServiceMock, *times*(1)).getResult();  
 }  
}

Тези библиотеки се използват в комбинация за покриване на различни аспекти на тестовете, като модулни, функционални и мок тестове. Те предоставят инструменти за написване и изпълнение на тестове с цел поддържане на качеството и надеждността на софтуера.

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:

* **Модуларност:**
  + 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.

4. **По-Лесно Тестване:**

Чрез разделянето на системата на три отделни компонента, MVC архитектурата улеснява процеса на тестване:

* **Паралелни Тестове:**
  + Всеки компонент (модел, изглед, контролер) може да бъде тестван поотделно, което улеснява откриването и коригирането на проблеми в кода.

5. **Съвместимост с JavaFX, Hibernate 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.
* **Hibernate за ORM:**
  + Хибернейт се интегрира като модел в MVC, предоставяйки лесен начин за взаимодействие с базата от данни.
* **PostGreSQL за Базата Данни:**
  + Изборът на PostGreSQL осигурява стабилна и мощна релационна база от данни, съвместима с Hibernate.

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.Top of Form

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.
* **Exceptions:**
  + **Описание:** Съдържа използвани изключения в програмата.
  + **Цел:** Добавя детайли и документира възможни грешки и изключения в системата.
* **Engines:**
  + **Описание:** Класове, отговарящи за логика, отделна от услугите.
  + **Цел:** Разделянето на различни аспекти на логиката на по-малки компоненти предоставя по-голяма гъвкавост и поддържа съгласуваност.

2. **Database:**

* **Entities:**
  + **Описание:** Модели на апликацията, които Hibernate използва за създаване на таблици в базата данни.
  + **Цел:** Предоставяне на структура и дефиниране на обекти, които отразяват бизнес обекти в базата данни.
* **Repositories:**
  + **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.
* **Utils:**
  + **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 "entities" 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 три enum класа: **BookFormStatus**, **BookStatus**, **Genre**, **Ratings**, и **Role**.

**1. BookFormStatus**

**Описание**

**BookFormStatus** представя възможните статуси на заявки за книга в библиотечна система. Всеки статус има съответен текст за представяне.

**Enum Константи**

* **IN\_USE**: Заявката за книга е в моментно ползване.
* **RETURNED**: Заявката за книга е върната.
* **LATE**: Заявката за книга е закъсняла.

**Полета**

* **displayValue**: Четимо представяне на статуса на заявката за книга.

**2. BookStatus**

**Описание**

**BookStatus** представя възможните статуси на книга в библиотечна система. Всеки статус има съответен текст за представяне.

**Enum Константи**

* **AVAILABLE**: Книгата е налична за заемане.
* **LENT**: Книгата е заета от читател.
* **ARCHIVED**: Книгата е архивирана и не е налична за обикновено заемане.
* **DAMAGED**: Книгата е повредена и може да не е налична за заемане.
* **IN\_READING\_ROOM**: Книгата е налична за четене в читалната на библиотеката.

**Полета**

* **displayValue**: Четимо представяне на статуса на книгата.

**3. Genre**

**Описание**

**Genre** представя различни жанрове, които могат да бъдат присвоени на книгите в библиотечната система. Всеки жанр има съответен текст за представяне и уникален идентификатор.

**Enum Константи**

* **FICTION**
* **NON\_FICTION**
* **MYSTERY**
* **SCIENCE\_FICTION**
* **ROMANCE**
* **FANTASY**
* **HORROR**
* **THRILLER**
* **HISTORY**
* **BIOGRAPHY**
* **SELF\_HELP**
* **BUSINESS**
* **CHILDREN**
* **POETRY**
* **DRAMA**
* **COMEDY**
* **CRIME**
* **ADVENTURE**
* **PHILOSOPHY**
* **TRAVEL**

**Полета**

* **value**: Уникален идентификатор за всеки жанр.
* **displayValue**: Четимо представяне на жанра.

**4. Ratings**

**Описание**

**Ratings** представя различни оценки, които могат да бъдат присвоени на читателите. Всяка оценка има съответен текст за представяне, цяло число и текуща стойност (която не се използва в момента).

**Enum Константи**

* **NONE**
* **ZERO\_STAR**
* **ONE\_STAR**
* **TWO\_STAR**
* **THREE\_STAR**
* **FOUR\_STAR**
* **FIVE\_STAR**

**Полета**

* **displayValue**: Четимо представяне на оценката.
* **value**: Цяло число, представляващо стойността на оценката.

**5. Role**

**Описание**

**Role** представя различни роли, които могат да бъдат присвоени на потребителите в системата. Всяка роля има съответен текст за представяне.

**Enum Константи**

* **ADMIN**: Администратор с пълен достъп и авторитет.
* **OPERATOR**: Оператор с ограничен достъп и авторитет.

**Полета**

* **role**: Четимо представяне на ролята.

Пакетът "repositories" съдържа класове, които предоставят репозиторий за взаимодействие с базата данни. Включват се следните класове: **Repository**, **AuthorRepository**, **BookFormRepository**, **BookInventoryRepository**, **BookRepository**, **EventNotificationRepository**, **ReaderRatingRepository**, **ReaderRepository**, и **UserRepository**.

**1. Repository**

**Описание**

**Repository** е абстрактен клас, който предоставя основни операции за взаимодействие с базата данни чрез Hibernate. Всяко репозитори предоставя базова функционалност за работа с обекти, които наследяват **DBEntity**.

**Полета**

* **session**: Hibernate **Session**, представляваща текущата сесия с базата данни.
* **transaction**: Hibernate **Transaction**, използвана за контрол на транзакциите.

**Методи**

1. **findById(Long id)**: Връща обект по дадено ID.
2. **findAll()**: Връща списък от всички обекти в базата данни.
3. **getById(Long id)**: Връща обект по дадено ID.
4. **delete(T object)**: Изтрива обект от базата данни.
5. **save(T object)**: Запазва обект в базата данни.
6. **update(T object)**: Актуализира обект в базата данни.
7. **deleteAll(Collection<T> entities)**: Изтрива всички обекти от предоставената колекция.

**2. AuthorRepository**

**Описание**

**AuthorRepository** е клас, който разширява **Repository** и предоставя специфични методи за работа с обекти от тип **Author**.

**Специфични Методи**

1. **getInstance()**: Връща единствена инстанция на **AuthorRepository** (сингълтон).

**3. BookFormRepository**

**Описание**

**BookFormRepository** е клас, който разширява **Repository** и предоставя специфични методи за работа с обекти от тип **BookForm**.

**Специфични Методи**

1. **getInstance()**: Връща единствена инстанция на **BookFormRepository** (сингълтон).

**4. BookInventoryRepository**

**Описание**

**BookInventoryRepository** е клас, който разширява **Repository** и предоставя специфични методи за работа с обекти от тип **BookInventory**.

**Специфични Методи**

1. **getInstance()**: Връща единствена инстанция на **BookInventoryRepository** (сингълтон).

**5. BookRepository**

**Описание**

**BookRepository** е клас, който разширява **Repository** и предоставя специфични методи за работа с обекти от тип **Book**.

**Специфични Методи**

1. **getInstance()**: Връща единствена инстанция на **BookRepository** (сингълтон).
2. **saveAll(Collection<Book> entities)**: Запазва всички обекти от предоставената колекция.

**6. EventNotificationRepository**

**Описание**

**EventNotificationRepository** е клас, който разширява **Repository** и предоставя специфични методи за работа с обекти от тип **EventNotification**.

**Специфични Методи**

1. **getInstance()**: Връща единствена инстанция на **EventNotificationRepository** (сингълтон).

**7. ReaderRatingRepository**

**Описание**

**ReaderRatingRepository** е клас, който разширява **Repository** и предоставя специфични методи за работа с обекти от тип **ReaderRating**.

**Специфични Методи**

1. **getInstance()**: Връща единствена инстанция на **ReaderRatingRepository** (сингълтон).

**8. ReaderRepository**

**Описание**

**ReaderRepository** е клас, който разширява **Repository** и предоставя специфични методи за работа с обекти от тип **Reader**.

**Специфични Методи**

1. **getInstance()**: Връща единствена инстанция на **ReaderRepository** (сингълтон).

**9. UserRepository**

**Описание**

**UserRepository** е клас, който разширява **Repository** и предоставя специфични методи за работа с обекти от тип **User**.

**Специфични Методи**

1. **getInstance()**: Връща единствена инстанция на **UserRepository** (сингълтон).
2. **findByUsername(String username)**: Връща обект от тип **User** по дадено потребителско име.

Бизнес логика:

Пакетът "engines" съдържа реализации на търсачки (**SearchEngine**) за различни обекти в системата, като **BookInventorySearchEngine**, **OperatorSearchEngine** и **ReaderSearchEngine**.

**1. BookInventorySearchEngine**

**Описание**

**BookInventorySearchEngine** е клас, който имплементира интерфейса **SearchEngine** за търсене на книжни инвентари.

**Методи**

1. **search(List<BookInventory> items, String stringToSearch)**: Търсене на книжни инвентари по зададен текст.

**Важни Параметри**

* **items**: Списък от книжни инвентари, върху които да се извърши търсенето.
* **stringToSearch**: Текст за търсене.

**Връщан Резултат**

* **List<BookInventory>**: Списък от книжни инвентари, отговарящи на критериите за търсене.

**Изключения**

* **SearchEngineException**: В случай на грешка по време на операцията на търсене.

**2. OperatorSearchEngine**

**Описание**

**OperatorSearchEngine** е клас, който имплементира интерфейса **SearchEngine** за търсене на оператори (потребители с роля "оператор").

**Методи**

1. **search(List<User> items, String stringToSearch)**: Търсене на оператори по зададен текст.

**Важни Параметри**

* **items**: Списък от оператори, върху които да се извърши търсенето.
* **stringToSearch**: Текст за търсене.

**Връщан Резултат**

* **Set<User>**: Множество от оператори, отговарящи на критериите за търсене.

**Изключения**

* **SearchEngineException**: В случай на грешка по време на операцията на търсене.

**3. ReaderSearchEngine**

**Описание**

**ReaderSearchEngine** е клас, който имплементира интерфейса **SearchEngine** за търсене на читатели.

**Методи**

1. **search(List<Reader> items, String stringToSearch)**: Търсене на читатели по зададен текст.

**Важни Параметри**

* **items**: Списък от читатели, върху които да се извърши търсенето.
* **stringToSearch**: Текст за търсене.

**Връщан Резултат**

* **Set<Reader>**: Множество от читатели, отговарящи на критериите за търсене.

**Изключения**

* **SearchEngineException**: В случай на грешка по време на операцията на търсене.

**4. SearchEngine Interface**

**Описание**

**SearchEngine** интерфейсът дефинира общите методи, които трябва да бъдат имплементирани от всеки клас-търсач в системата.

**Методи**

1. **search(List<T> items, String stringToSearch)**: Търсене на обекти от тип **T** по зададен текст.

**Важни Параметри**

* **items**: Списък от обекти, върху които да се извърши търсенето.
* **stringToSearch**: Текст за търсене.

**Връщан Резултат**

* **Collection<T>**: Колекция от обекти, отговарящи на критериите за търсене.

**Изключения**

* **SearchEngineException**: В случай на грешка по време на операцията на търсене.

Пакетът "exceptions" съдържа изключения, които се използват в бизнес логиката на библиотечната система.

**1. EmailException**

**Описание**

**EmailException** е изключение, което наследява **LibraryException** и се използва за грешки, свързани с електронната поща.

**2. TransportException**

**Описание**

**TransportException** е изключение, което наследява **EmailException** и се използва за грешки, свързани с транспорт на електронна поща.

**3. BookInventorySearchEngineException**

**Описание**

**BookInventorySearchEngineException** е изключение, което наследява **SearchEngineException** и се използва за грешки, свързани с търсенето на книжни инвентари.

**4. OperatorSearchEngineException**

**Описание**

**OperatorSearchEngineException** е изключение, което наследява **SearchEngineException** и се използва за грешки, свързани с търсенето на оператори.

**5. ReaderSearchEngineException**

**Описание**

**ReaderSearchEngineException** е изключение, което наследява **SearchEngineException** и се използва за грешки, свързани с търсенето на читатели.

**6. SearchEngineException**

**Описание**

**SearchEngineException** е изключение, което наследява **LibraryException** и се използва за грешки, свързани с търсачки.

**7. AdminNotFoundException**

**Описание**

**AdminNotFoundException** е изключение, което наследява **UserNotFoundException** и се използва, когато администратор не бъде намерен в системата.

**8. OperatorNotFoundException**

**Описание**

**OperatorNotFoundException** е изключение, което наследява **UserNotFoundException** и се използва, когато оператор не бъде намерен в системата.

**9. ReaderNotFoundException**

**Описание**

**ReaderNotFoundException** е изключение, което наследява **UserNotFoundException** и се използва, когато читател не бъде намерен в системата.

**10. UserNotFoundException**

**Описание**

**UserNotFoundException** е изключение, което наследява **LibraryException** и се използва, когато потребител не бъде намерен в системата.

**11. IncorrectInputException**

**Описание**

**IncorrectInputException** е изключение, което наследява **LibraryException** и се използва за грешки при некоректен вход.

**12. InvalidQuantityException**

**Описание**

**InvalidQuantityException** е изключение, което наследява **LibraryException** и се използва за грешки при невалидно количество.

**13. LibraryException**

**Описание**

**LibraryException** е основното изключение, което наследява **Exception** и се използва за общи грешки в библиотечната система.

**14. NonExistentServiceException**

**Описание**

**NonExistentServiceException** е изключение, което наследява **LibraryException** и се използва за грешки, свързани с несъществуващи услуги.

**15. ObjectCannotBeNullException**

**Описание**

**ObjectCannotBeNullException** е изключение, което наследява **LibraryException** и се използва, когато обект не може да бъде null.

**16. ReaderException**

**Описание**

**ReaderException** е изключение, което наследява **LibraryException** и се използва за грешки, свързани с читатели.

**17. ReturnBookException**

**Описание**

**ReturnBookException** е изключение, което наследява **LibraryException** и се използва за грешки при връщането на книга.

**18. UserExistException**

**Описание**

**UserExistException** е изключение, което наследява **LibraryException** и се използва, когато потребителят вече съществува.

**Заключение**

Пакетът "exceptions" предоставя разнообразни изключения, които покриват различни сценарии на грешки в библиотечната система. Те са подробно документирани и могат да бъдат използвани за подобрено управление на грешките и по-добра диагностика.

Пакетът "validators" съдържа класове и интерфейси, които се използват за валидация на различни типове данни в библиотечната система.

**1. StrongPasswordValidator**

**Описание**

Класът **StrongPasswordValidator** реализира интерфейса **Validator** и се използва за проверка дали предоставена парола отговаря на критериите за силна парола. Критериите включват:

* Поне една цифра [0-9]
* Поне една малка буква [a-z]
* Поне една голяма буква [A-Z]
* Без интервали
* Минимална дължина от 6 символа
* Максимална дължина от 20 символа

Класът използва регулярен израз за извършване на валидацията.

Полета

* **PASSWORD\_PATTERN**: Статично поле, дефиниращо регулярния израз за валидация на паролата.

Методи

* **isValid(String password)**: Проверява дали предоставената парола отговаря на критериите за силна парола.

**2. Validator**

**Описание**

Интерфейсът **Validator** дефинира контракт за класове, които извършват валидация на дадена стойност. Класовете, които реализират този интерфейс, трябва да предоставят реализация на метода **isValid(String value)**, за да определят дали стойността е валидна въз основа на конкретни критерии.

Методи

* **isValid(String value)**: Проверява дали предоставената стойност е валидна в съответствие с конкретните критерии на валидация.

**Заключение**

Пакетът "validators" предоставя гъвкави инструменти за валидация на различни типове данни в библиотечната система. Класът StrongPasswordValidator е конкретен пример за използване на регулярни изрази за извършване на сложни проверки. Интерфейсът Validator позволява създаването на персонализирани валидатори за различни критерии.

**1. EmailSenderService**

**Описание**

Класът **EmailSenderService** предоставя функционалност за изпращане на имейли, използвайки JavaMail API. Конфигурацията на имейла, като SMTP хост, порт и SSL настройки, може да бъде персонализирана. Класът поддържа и стандартни SMTP и SMTP с SSL връзки.

Полета

* **username**: Потребителско име за аутентикация.
* **password**: Парола за аутентикация.
* **smtpHost**: SMTP сървър хост.
* **smtpPort**: SMTP сървър порт.
* **useSSL**: Показва дали да се използва SSL.
* **transport**: Обект, отговарящ за транспортирането на имейли.

Методи

* **sendEmail(String to, String subject, String body)**: Изпраща имейл със зададени получател, тема и тяло.
* **getProperties()**: Връща и конфигурира имейл свойствата въз основа на предоставената SMTP конфигурация.
* **initializeTransport()**: Инициализира и връща обект за транспортиране на имейли.

**2. LogInService**

**Описание**

Класът **LogInService** предоставя функционалност, свързана с аутентикацията и входа на потребител. Взаимодейства с **UserRepository**, за да вземе информация за потребителите, базирайки се на предоставеното потребителско име.

Полета

* **logger**: Логгер за записване на информацията.
* **userRepository**: Репозиторий за достъп до данните за потребителите в базата данни.

Методи

* **getUser(String username, String password)**: Връща потребител по дадено потребителско име и парола.

**3. ServiceFactory**

**Описание**

Класът **ServiceFactory** предоставя фабрика за създаване на обекти от различни видове услуги в приложението. Използва се за централизирано създаване на инстанции на различни видове услуги.

Полета

* **logger**: Логгер за записване на информацията.

Методи

* **getService(Class<T> serviceClass)**: Създава инстанция на услугата от предоставения клас.

Админски сървиси:

1. **AddBookQuantityService**
   * Класът има метод **increaseBookQuantity**, който увеличава броя на наличните книги в инвентара.
   * Приема количество (**quantityString**) и **BookInventory**.
   * Проверява валидността на данните и след това създава и добавя нови книги в инвентара.
2. **AdministratorBooksDialogService**
   * Отговаря за операции свързани с диалогове за администрация на книги.
   * Методът **removeSelectedBooks** приема **BookInventory** и списък от книги за премахване.
   * Премахва избраните книги от инвентара и обновява свързаните форми.
3. **AdministratorBooksService**
   * Класът отговаря за администрация на книги - търсене на инвентари, премахване на инвентари, обновяване на форми.
   * Методът **searchBookInventory** търси инвентари, използвайки **SearchEngine** за филтриране.
   * **removeInventory** премахва инвентар и обновява свързаните форми.
   * **updateBookForms** обновява формите, свързани с книги.
4. **AdministratorOperatorsService**
   * Класът отговаря за администрация на оператори - търсене на оператори, премахване на оператори.
   * Методът **searchUser** търси оператори, използвайки **OperatorSearchEngine** за филтриране.
   * **removeOperator** премахва оператор, при условие, че не е администратор.
5. **BookRegistrationService**
   * Отговаря за регистрацията на нови книги в системата.
   * Методът **registerNewBook** приема параметри за нова книга и я добавя към инвентара.
   * Извършва валидация на входните данни и създава книги, добавяйки ги в инвентар.
6. **CreateOperatorService**
   * Отговаря за създаването на нови оператори.
   * Методът **createOperator** проверява валидността на входните данни, включително дали потребител с дадено име вече съществува.
   * Създава нов оператор с роля "OPERATOR" и го записва в репозиторията.

Тези услуги образуват модулен и ясен дизайн, който позволява различните части на системата да работят заедно. Те извършват нужните валидации и манипулации с данни в съответствие със задачите, които трябва да изпълнят.

Операторски сървиси:

1. **BookFormShowService:**
2. **CreateBookFormService:**
3. **CreateReaderProfileService:**
4. **InboxService:**
5. **OperatorBooksService и OperatorReadersService:**

**AddBookQuantityController**

1. **Описание:** Контролер за управление на добавянето на количество на книги.
2. **Публични Методи:**
   * **initialize(URL location, ResourceBundle resources)**: Инициализира контролера.
   * **addButtonOnMouseClicked()**: Обработва събитието при кликване на бутона за добавяне.
3. **Полета:**
   * **quantityTextField**: Текстово поле за въвеждане на количество.
   * **addButton**: Бутон за добавяне на количество.
   * **informationLabel**: Етикет за информация и грешки.
4. **Процеси:**
   * Инициализация на контролера при стартиране.
   * Валидация и обработка на въведените данни при натискане на бутона за добавяне.

**AdministratorBooksController**

1. **Описание:** Контролер за управление на администраторската част от книжарницата.
2. **Публични Методи:**
   * **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
   * **searchBookButtonOnMouseClicked()**: Обработка на търсенето на книги.
   * **operatorsButtonOnMouseClicked(MouseEvent mouseEvent)**: Преход към операторската част.
3. **Полета:**
   * **searchBookTextField**: Текстово поле за търсене на книги.
   * **inventoryTableView**: Таблица с информация за наличните книги.
   * **bookTextArea**: Текстово поле за показване на подробности за книгата.
4. **Процеси:**
   * Инициализация на контролера и създаване на таблицата с книги.
   * Търсене на книги и обновяване на таблицата.

**AdministratorBooksDialogController**

1. **Описание:** Контролер за управление на диалога с подробности за книга.
2. **Публични Методи:**
   * **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
   * **closeButtonOnMouseClicked()**: Затваряне на диалога с подробности.
3. **Полета:**
   * **bookTableView**: Таблица с информация за книгата.
   * **closeButton**: Бутон за затваряне на диалога.
4. **Процеси:**
   * Инициализация на контролера и създаване на таблицата с информация за книгата.

**AdministratorOperatorsController**

1. **Описание:** Контролер за управление на операторската част от книжарницата.
2. **Публични Методи:**
   * **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
   * **booksButtonOnMouseClicked(MouseEvent mouseEvent)**: Преход към администраторската част.
3. **Полета:**
   * **searchBookTextField**: Текстово поле за търсене на оператори.
   * **operatorTableView**: Таблица с информация за операторите.
4. **Процеси:**
   * Инициализация на контролера и създаване на таблицата с оператори.
   * Търсене на оператори и обновяване на таблицата.

**CreateOperatorController**

1. **Описание:** Контролер за създаване на нов оператор.
2. **Публични Методи:**
   * **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
   * **createOperatorButtonOnMouseClicked(MouseEvent mouseEvent)**: Създаване на нов оператор.
   * **cancelButtonOnMouseClicked(MouseEvent mouseEvent)**: Отказ от създаване на оператор.
3. **Полета:**
   * **usernameTextField**: Текстово поле за въвеждане на потребителско име.
   * **passwordPasswordField**: Поле за въвеждане на парола (скрито).
   * **repeatPasswordPasswordField**: Поле за повторно въвеждане на паролата (скрито).
   * **showPasswordCheckBox**: Опция за показване на паролата.
4. **Процеси:**
   * Инициализация на контролера и управление на въвеждането на данни за нов оператор.

**1. BookFormShowController**

* **Описание:** Контролерът се грижи за показването на подробности за формулярите за книги и действията свързани с тях.
* **Методи:**
  + **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
  + **returnButtonOnMouseClicked(MouseEvent mouseEvent)**: Обработка на връщането на книги.
  + **closeButtonOnMouseClicked(MouseEvent mouseEvent)**: Обработка на затварянето на прозореца.
  + **notifyButtonOnMouseClicked()**: Обработка на уведомлението за връщане на книги.
  + **getTransferObjects()**: Извличане на обекти за трансфер.

**2. CreateBookFormController**

* **Описание:** Контролерът се грижи за създаването на формуляри за книги и свързаната функционалност.
* **Методи:**
  + **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
  + **searchReaderButtonOnMouseClicked()**: Обработка на търсенето на читатели.
  + **lendButtonOnMouseClicked()**: Обработка на заемане на книги.
  + **lendReadingRoomButtonOnMouseClicked()**: Обработка на заемане на книги за четене в читалище.
  + **cancelButtonOnMouseClicked(MouseEvent mouseEvent)**: Обработка на отказ от операция.

**3. InboxController**

* **Описание:** Контролерът се грижи за обработката на съобщенията в инбокса на оператора.
* **Методи:**
  + **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
  + **closeButtonOnMouseClicked(MouseEvent mouseEvent)**: Обработка на затварянето на инбокса.

**4. CreateReaderProfileController**

* **Описание:** Контролерът управлява създаването на профили на читатели.
* **Методи:**
  + **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
  + **createReaderProfileButtonOnMouseClicked(MouseEvent mouseEvent)**: Обработка на създаването на профил.
  + **cancelButtonOnMouseClicked(MouseEvent mouseEvent)**: Обработка на отказ от създаването на профил.

**5. OperatorBooksController**

* **Описание:** Контролерът управлява операциите, свързани с книгите, за операторите.
* **Методи:**
  + **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
  + **readersButtonOnMouseClicked(MouseEvent mouseEvent)**: Превключване към преглед на читатели.
  + **searchBookButtonOnMouseClicked(MouseEvent mouseEvent)**: Търсене на книги.
  + **logOutButtonOnMouseClicked(MouseEvent mouseEvent)**: Излизане от профила на оператора.
  + **anchorPaneOnMouseClicked()**: Обработка на кликване върху областта на екрана.
  + **bookTreeTableViewOnMouseClicked(MouseEvent mouseEvent)**: Обработка на кликване върху таблицата с книги.
  + **selectedBooksListViewOnMouseClicked()**: Обработка на кликване върху списъка с избрани книги.
  + **lendButtonOnMouseClicked()**: Обработка на заемане на книги.
  + **inboxButtonOnMouseClicked()**: Преглед на входящи съобщения.

**6. OperatorBooksDialogController**

* **Описание:** Контролерът управлява диалоговия прозорец с подробности за книгите за операторите.
* **Методи:**
  + **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
  + **closeButtonOnMouseClicked(MouseEvent mouseEvent)**: Затваряне на диалоговия прозорец.

**7. ResumeShowController**

* **Описание:** Контролерът управлява прозорец за показване на резюмета на книги.
* **Методи:**
  + **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
  + **closeButtonOnMouseClicked(MouseEvent mouseEvent)**: Затваряне на прозореца.

**8. OperatorReadersController**

* **Описание:** Контролерът управлява операциите, свързани с читателите, за операторите.
* **Методи:**
  + **initialize(URL location, ResourceBundle resources)**: Инициализация на контролера.
  + **booksButtonOnMouseClicked(MouseEvent mouseEvent)**: Превключване към преглед на книги.
  + **searchReaderButtonOnMouseClicked()**: Търсене на читатели.
  + **readerTableViewOnClicked()**: Обработка на кликване върху таблицата с читатели.
  + **bookFormListViewOnMouseClicked()**: Обработка на кликване върху списъка с формуляри за книги.
  + **readerRatingOnMouseClicked()**: Обновяване на рейтинга на читател.
  + **createReader(ActionEvent actionEvent)**: Създаване на нов профил на читател.
  + **removeReader(ActionEvent actionEvent)**: Премахване на профил на читател.

Основни компоненти в пакета frontend/controllers:

**1. LogInController**

**1.1 Въведение**

**LogInController** е JavaFX контролер, отговорен за управлението на логин функционалността в библиотечното приложение. Този контролер взаимодейства с **LogInService** за валидация на потребителски идентификационни данни и навигира към съответния изглед в зависимост от ролята на потребителя.

**1.2 Основни Характеристики**

* **Анотации:** Класът е анотиран с **@NoArgsConstructor** за генериране на конструктор без параметри от Lombok.
* **FXML Полета:**
  + **logInButton**: Бутона за вход.
  + **logInMessageLabel**: Етикет за съобщения свързани с входа.
  + **usernameTextField**: Поле за въвеждане на потребителско име.
  + **passwordPasswordField**: Поле за въвеждане на парола.
* **Приватни Полета:**
  + **service**: Обект от тип **LogInService**, отговарящ за логиката на входа.
* **Методи:**
  + **initialize**: Метод, извикан при инициализация на контролера, насочва фокуса върху полето за въвеждане на потребителско име и дефинира обработчици на събития.
  + **logInButtonOnAction**: Метод, извикан при натискане на бутона за вход, извършва валидация на въведените данни и пренасочва потребителя към съответния изглед.
  + **checkInput**: Метод, извършващ валидация на въведените потребителски данни.

**1.3 Взаимодействие с Сервизи и SceneLoader**

* **LogInController** използва **LogInService** от **com.library.backend.services** за валидация на потребителски идентификационни данни.
* При успешен вход, **SceneLoader** се използва за пренасочване към съответния изглед в зависимост от ролята на потребителя.

**2. Controller Интерфейс**

**2.1 Въведение**

**Controller** интерфейсът служи като базов интерфейс за всички JavaFX контролери в библиотечния фронтенд. Той разширява **javafx.fxml.Initializable** интерфейса, гарантирайки, че класовете, които го реализират, предоставят метод за инициализация на контролера.

**2.2 Основни Характеристики**

* **Методи:**
  + **initialize**: Метод, който трябва да бъде презареден от конкретните контролери. Извиква се при инициализация на контролера и се използва за извършване на необходимите настройки.

**2.3 Роля на Контролерите**

* Контролерите в JavaFX са отговорни за инициализацията на компонентите на потребителския интерфейс, обработката на потребителски вход, и управлението на комуникацията между изгледа и подлежащата данна.

Пакетът utils:

Пакет tableviews:

**1. BookTableViewBuilder**

**1.1 Въведение**

**BookTableViewBuilder** е клас, който реализира интерфейса **TableViewBuilder** за създаване и конфигуриране на колони в **TableView** за обекти от тип **Book**.

**1.2 Основни Характеристики**

* **Полета:**
  + **logger**: Статичен логгер, използван за записване на събития в журнала.
* **Методи:**
  + **createTableViewColumns(TableView<Book> tableView)**: Метод, който създава и конфигурира колоните на дадения **TableView** за обекти от тип **Book**.

**2. BookTreeTableViewBuilder**

**2.1 Въведение**

**BookTreeTableViewBuilder** е клас, който реализира интерфейса **TreeTableViewBuilder** за създаване и конфигуриране на колони в **TreeTableView** за обекти от тип **Book** в дървовидна структура.

**2.2 Основни Характеристики**

* **Полета:**
  + **logger**: Статичен логгер, използван за записване на събития в журнала.
* **Методи:**
  + **createTreeTableViewColumns(TreeTableView<Book> bookTreeTableView)**: Метод, който създава и конфигурира колоните на дадения **TreeTableView** за обекти от тип **Book** в дървовидна структура.

**3. ContextMenuBuilder**

**3.1 Въведение**

**ContextMenuBuilder** е клас, предоставящ статични методи за създаване на контекстно меню със зададени елементи.

**3.2 Основни Характеристики**

* **Методи:**
  + **prepareContextMenu(Map<String, EventHandler<ActionEvent>> menuItems)**: Статичен метод, създаващ **ContextMenu** с елементи, подадени чрез **menuItems**.

**4. HiddenCheckBoxListCell**

**4.1 Въведение**

**HiddenCheckBoxListCell** е клас, наследяващ **ListCell**, който скрива полето за отметка и показва само текста.

**4.2 Основни Характеристики**

* **Методи:**
  + **updateItem(T item, boolean empty)**: Пренаписан метод, който скрива полето за отметка и показва само текста на елемента, когато **empty** е **false**.

**5. InventoryTableViewBuilder**

**5.1 Въведение**

**InventoryTableViewBuilder** е клас, който реализира интерфейса **TableViewBuilder** за създаване и конфигуриране на колони в **TableView** за обекти от тип **BookInventory**.

**5.2 Основни Характеристики**

* **Полета:**
  + **logger**: Статичен логгер, използван за записване на събития в журнала.
* **Методи:**
  + **createTableViewColumns(TableView<BookInventory> tableView)**: Метод, който създава и конфигурира колоните на дадения **TableView** за обекти от тип **BookInventory**.

**6. OperatorTableViewBuilder**

**6.1 Въведение**

**OperatorTableViewBuilder** е клас, който реализира интерфейса **TableViewBuilder** за създаване и конфигуриране на колони в **TableView** за обекти от тип **User** с роля "OPERATOR".

**6.2 Основни Характеристики**

* **Полета:**
  + **logger**: Статичен логгер, използван за записване на събития в журнала.
* **Методи:**
  + **createTableViewColumns(TableView<User> tableView)**: Метод, който създава и конфигурира колоните на дадения **TableView** за обекти от тип **User** с роля "OPERATOR".

**7. ReaderTableViewBuilder**

**7.1 Въведение**

**ReaderTableViewBuilder** е клас, който реализира интерфейса **TableViewBuilder** за създаване и конфигуриране на колони в **TableView** за обекти от тип **Reader**.

**7.2 Основни Характеристики**

* **Полета:**
  + **logger**: Статичен логгер, използван за записване на събития в журнала.
* **Методи:**
  + **createTableViewColumns(TableView<Reader> tableView)**: Метод, който създава и конфигурира колоните на дадения **TableView** за обекти от тип **Reader**.

**8. Интерфейс TableViewBuilder**

**8.1 Въведение**

**TableViewBuilder** е интерфейс, дефиниращ методи за създаване и конфигуриране на колони в **TableView** за обекти от даден тип.

**8.2 Основни Характеристики**

* **Полета:**
  + **logger**: Статичен логгер, използван за записване на събития в журнала.
* **Методи:**
  + **createTableViewColumns(TableView<T> tableView)**: Метод, който трябва да бъде реализиран от класовете, използващи интерфейса, за създаване и конфигуриране на колони.
  + **updateTableView(TableView<T> tableView, Collection<T> collection)**: Метод, който обновява съдържанието на **TableView** с нова колекция от обекти.
  + **getSelectedItem(TableView<T> tableView)**: Метод, който връща избрания елемент от **TableView**.
  + **getSelectedItems(TableView<T> tableView)**: Метод, който връща списък от избраните елементи от **TableView**.

**9. Интерфейс TreeTableViewBuilder**

**9.1 Въведение**

**TreeTableViewBuilder** е интерфейс, дефиниращ метод за създаване и конфигуриране на колони в **TreeTableView** за обекти от даден тип в дървовидна структура.

**9.2 Основни Характеристики**

* **Методи:**
  + **createTreeTableViewColumns(TreeTableView<T> treeTableView)**: Метод, който трябва да бъде реализиран от класовете, използващи интерфейса, за създаване и конфигуриране на колони в **TreeTableView** в дървовидна структура.
  + **getSelectedItem(TreeTableView<T> tableView)**: Метод, който връща избрания елемент от **TreeTableView**.
  + **getSelectedItems(TreeTableView<T> tableView)**: Метод, който връща списък от избраните елементи от **TreeTableView**.

Помощен клас DialogUtils:

**DialogUtils** е клас, предоставящ методи за изобразяване на различни видове JavaFX диалози, включително предупреждения и диалози за потвърждение.

**1. Публични Методи**

**1.1 showAlert(AlertType alertType, String title, String contentText)**

Методът показва диалог със зададен тип, заглавие и съдържание.

* **Параметри:**
  + **alertType**: Типът на диалога (например, INFORMATION, ERROR).
  + **title**: Заглавието на диалога.
  + **contentText**: Съдържанието на диалога.

**1.2 showInfo(String title, String contentText)**

Методът показва диалог с информационен тип със зададено заглавие и съдържание.

* **Параметри:**
  + **title**: Заглавието на информационния диалог.
  + **contentText**: Съдържанието на информационния диалог.

**1.3 showError(String title, String contentText)**

Методът показва диалог с грешен тип със зададено заглавие и съдържание.

* **Параметри:**
  + **title**: Заглавието на грешния диалог.
  + **contentText**: Съдържанието на грешния диалог.

**1.4 showWarning(String title, String contentText)**

Методът показва диалог с предупредителен тип със зададено заглавие и съдържание.

* **Параметри:**
  + **title**: Заглавието на предупредителния диалог.
  + **contentText**: Съдържанието на предупредителния диалог.

**1.5 showConfirmation(String title, String contentText): boolean**

Методът показва диалог за потвърждение със зададено заглавие и съдържание, позволявайки на потребителя да потвърди или отмени действие.

* **Параметри:**
  + **title**: Заглавието на диалога за потвърждение.
  + **contentText**: Съдържанието на диалога за потвърждение.
* **Резултат:**
  + Връща **true**, ако потребителят кликне 'Да'.
  + Връща **false**, ако потребителят кликне 'Не' или затвори диалога.

Пакетът utils/colors:

**1. Въведение**

Пакетът **utils.colors** предоставя различни Logback конвертори за оцветяване на лог съобщения в конзолата в зависимост от различни аспекти като име на логера, ниво на логиране, маркери и други.

**2. CustomHighlightingLoggerName**

**CustomHighlightingLoggerName** е конвертор, който осигурява персонализирано оцветяване на името на логера в лог съобщенията. Името на логера се оцветява в жълто, когато съдържа подниз "important", в противен случай се използва цвета на фона по подразбиране.

**3. CustomHighlightingLogLevel**

**CustomHighlightingLogLevel** е конвертор, който осигурява персонализирано оцветяване на нивата на логиране в лог съобщенията. Цветът на предната част на лог нивото се персонализира в зависимост от сериозността на лог съобщението.

**4. CustomHighlightingMarker**

**CustomHighlightingMarker** е конвертор, който осигурява персонализирано оцветяване на лог съобщенията в зависимост от наличието на конкретен маркер. Ако лог съобщението съдържа маркера с име "highlight", цвета на предната част се персонализира; в противен случай се използва цвета на фона по подразбиране.

**5. CustomHighlightingMessage**

**CustomHighlightingMessage** е конвертор, който осигурява персонализирано оцветяване на лог съобщенията в зависимост от тяхното ниво на логиране. Лог съобщенията с нива "ERROR" и "WARN" се оцветяват съответно в червено и жълто, а съобщенията с други нива използват цвета на фона по подразбиране.

**6. CustomHighlightingPackageName**

**CustomHighlightingPackageName** е конвертор, който осигурява персонализирано оцветяване на лог съобщенията в зависимост от тяхното ниво на логиране. Лог съобщенията с нива "ERROR" и "WARN" се оцветяват съответно в червено и жълто, а съобщенията с други нива използват цвета на фона по подразбиране.

**7. CustomHighlightingThread**

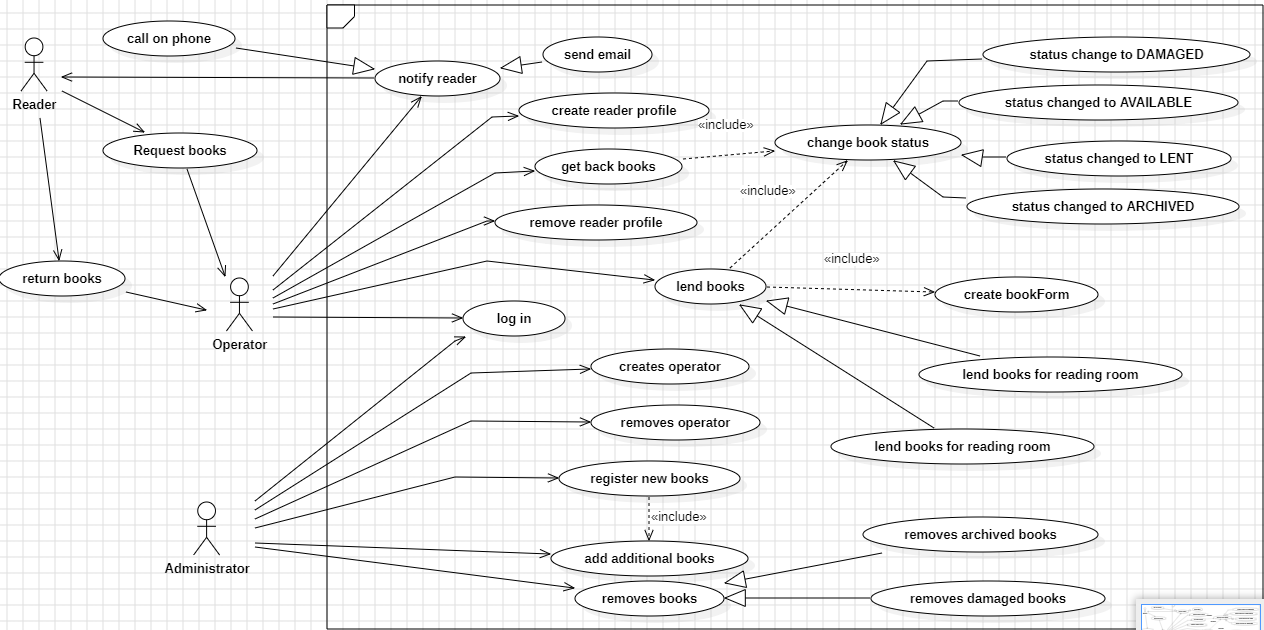
**CustomHighlightingThread** е конвертор, който осигурява персонализирано оцветяване на лог съобщенията в зависимост от името на техните нишки. Лог съобщенията от нишки, съдържащи термина "background" в името си, се оцветяват в синьо, докато останалите използват цвета на фона по подразбиране.

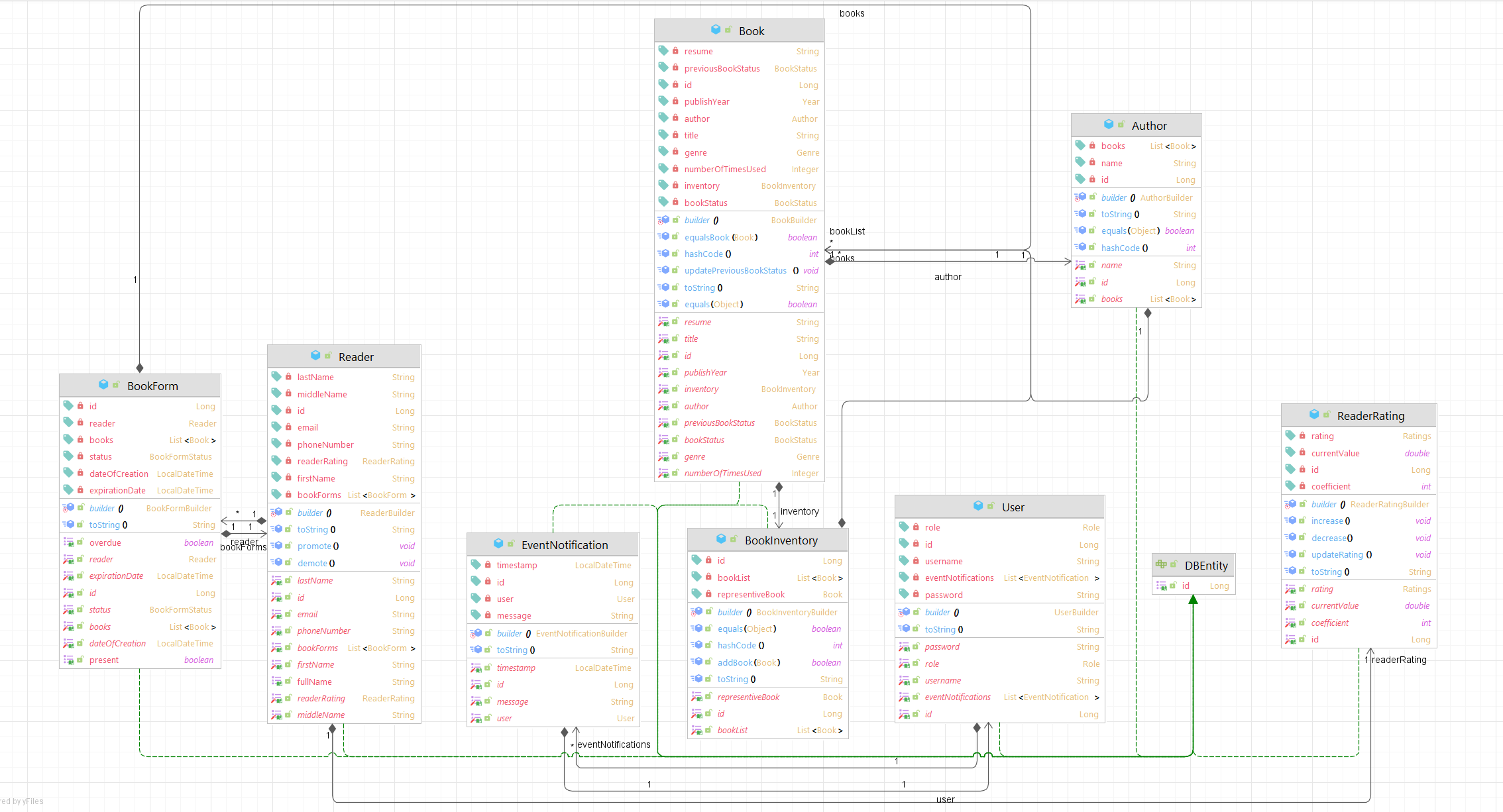
**8. CustomHighlightingTimestamp**

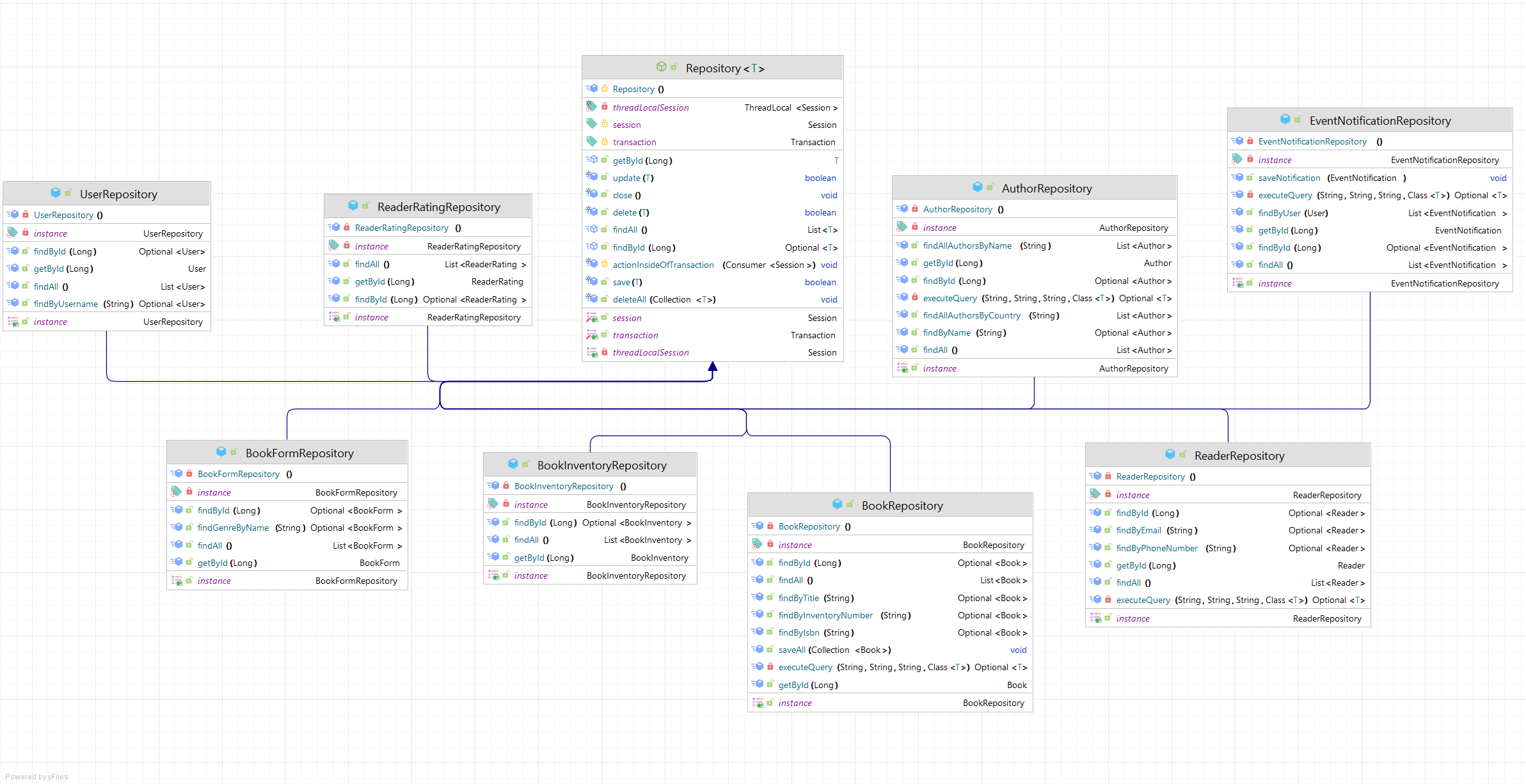
**CustomHighlightingTimestamp** е конвертор, който осигурява персонализирано оцветяване на лог съобщенията в зависимост от техните времеви марки и нива на логиране. Лог съобщенията с нива "ERROR" се оцветяват в червено, "WARN" в жълто, а съобщенията с други нива използват цвета на фона по подразбиране.

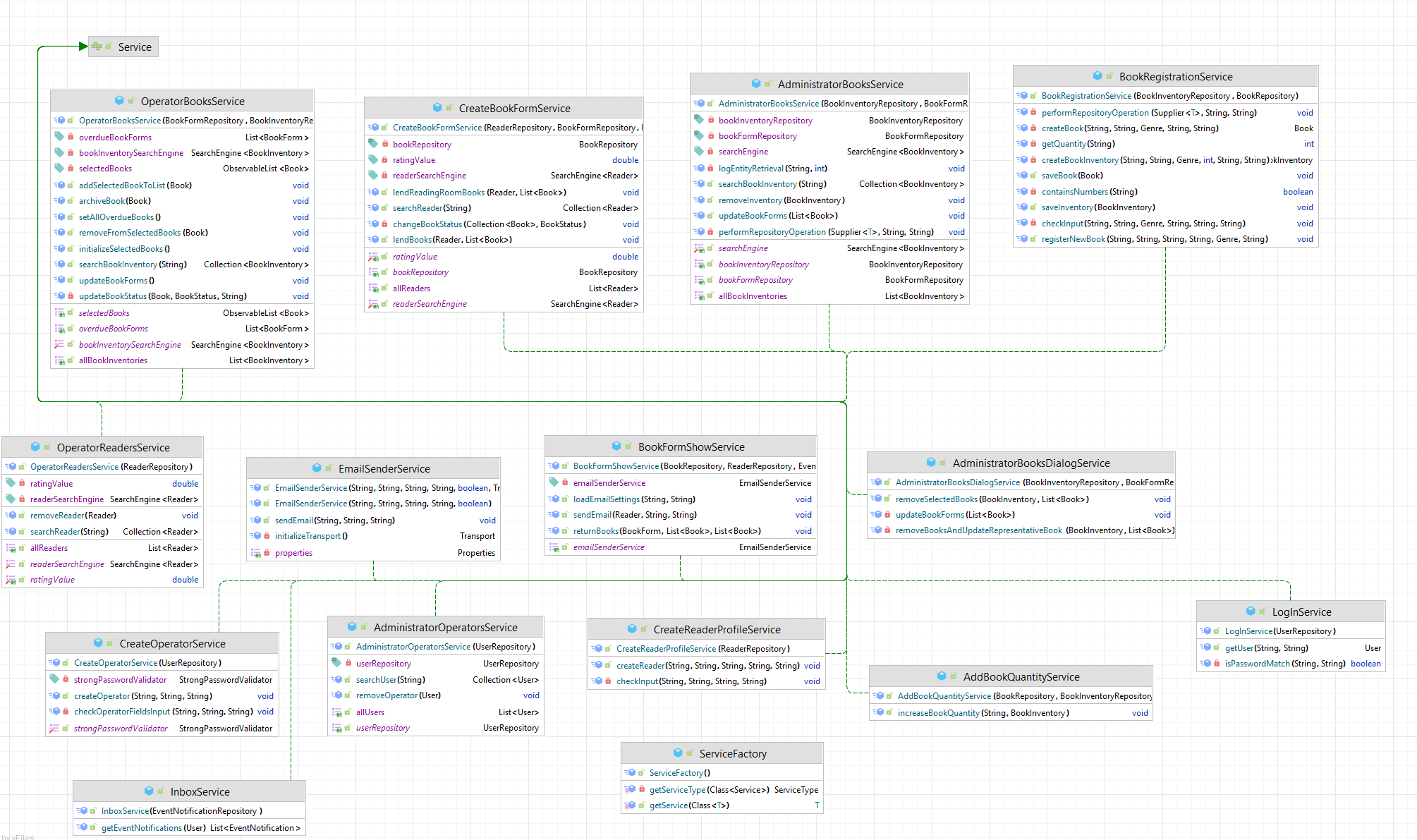
Тези конвертори предоставят възможност за детайлно персонализиране на визуалния стил на лог съобщенията в зависимост от различни характеристики и критерии.

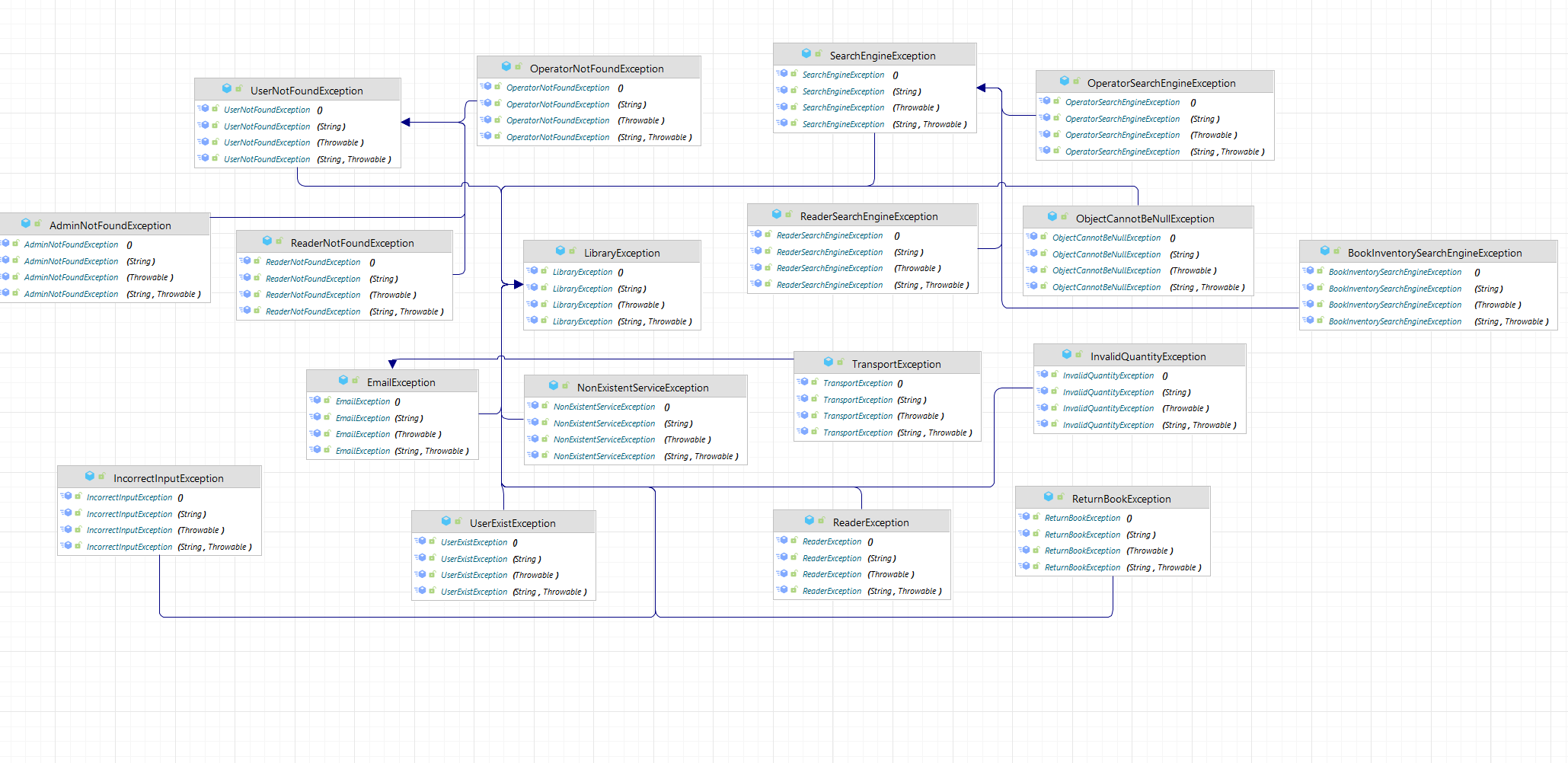
**3.2.UML диаграми**

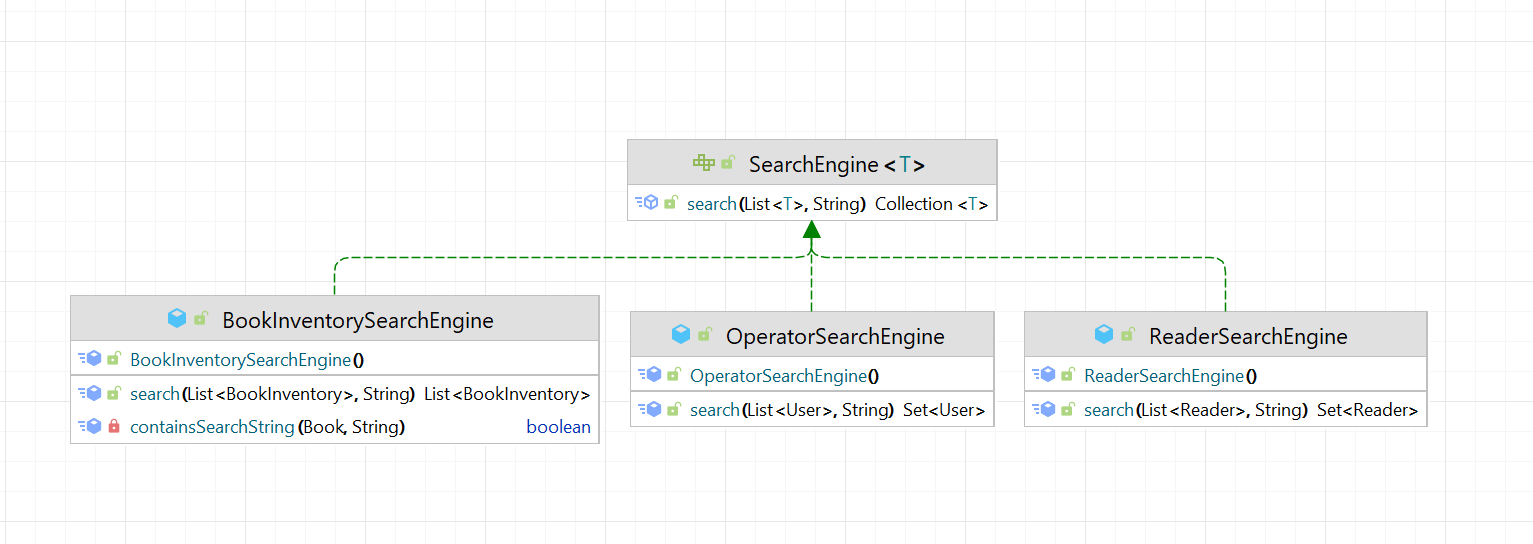
****

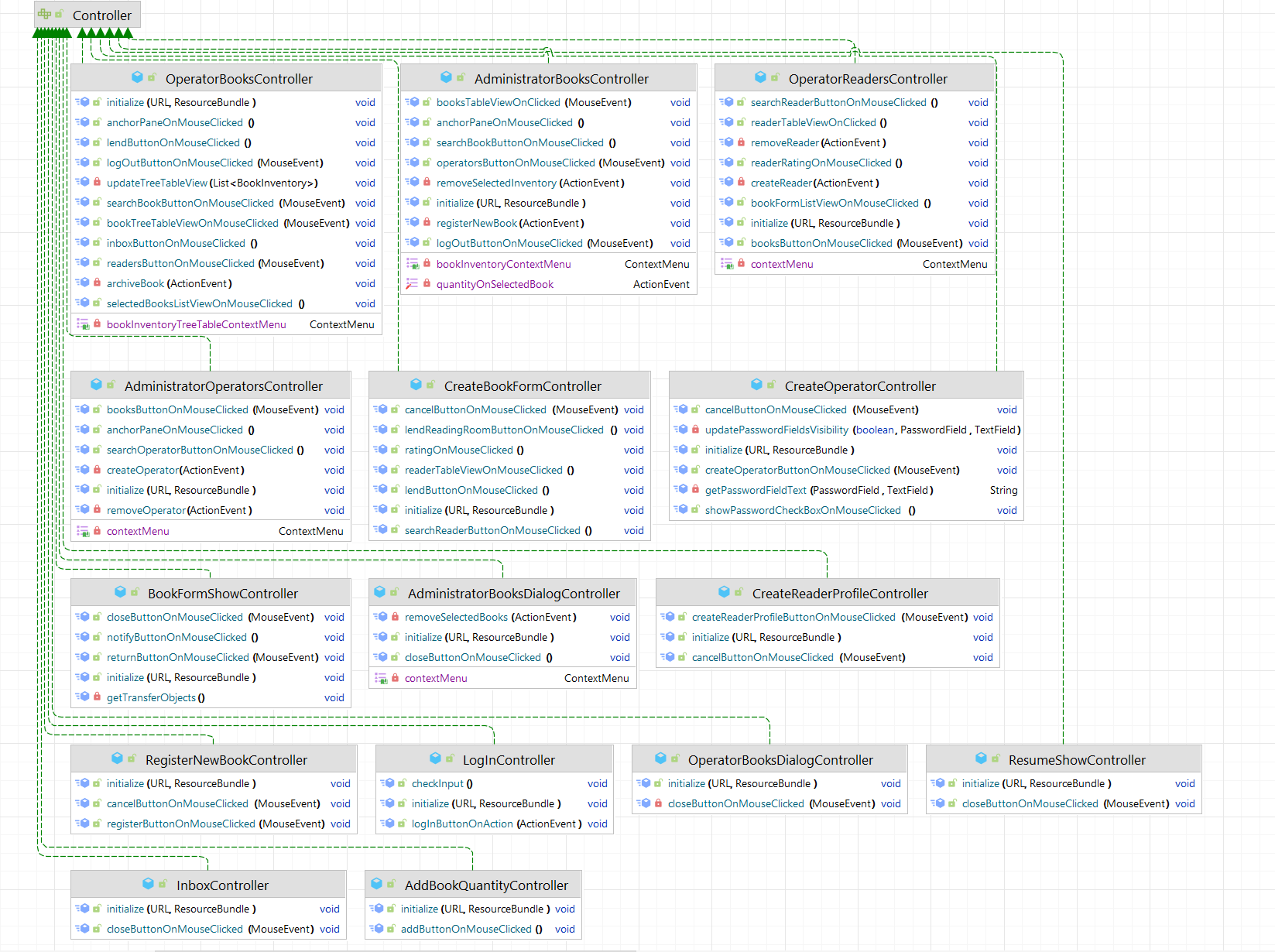
****

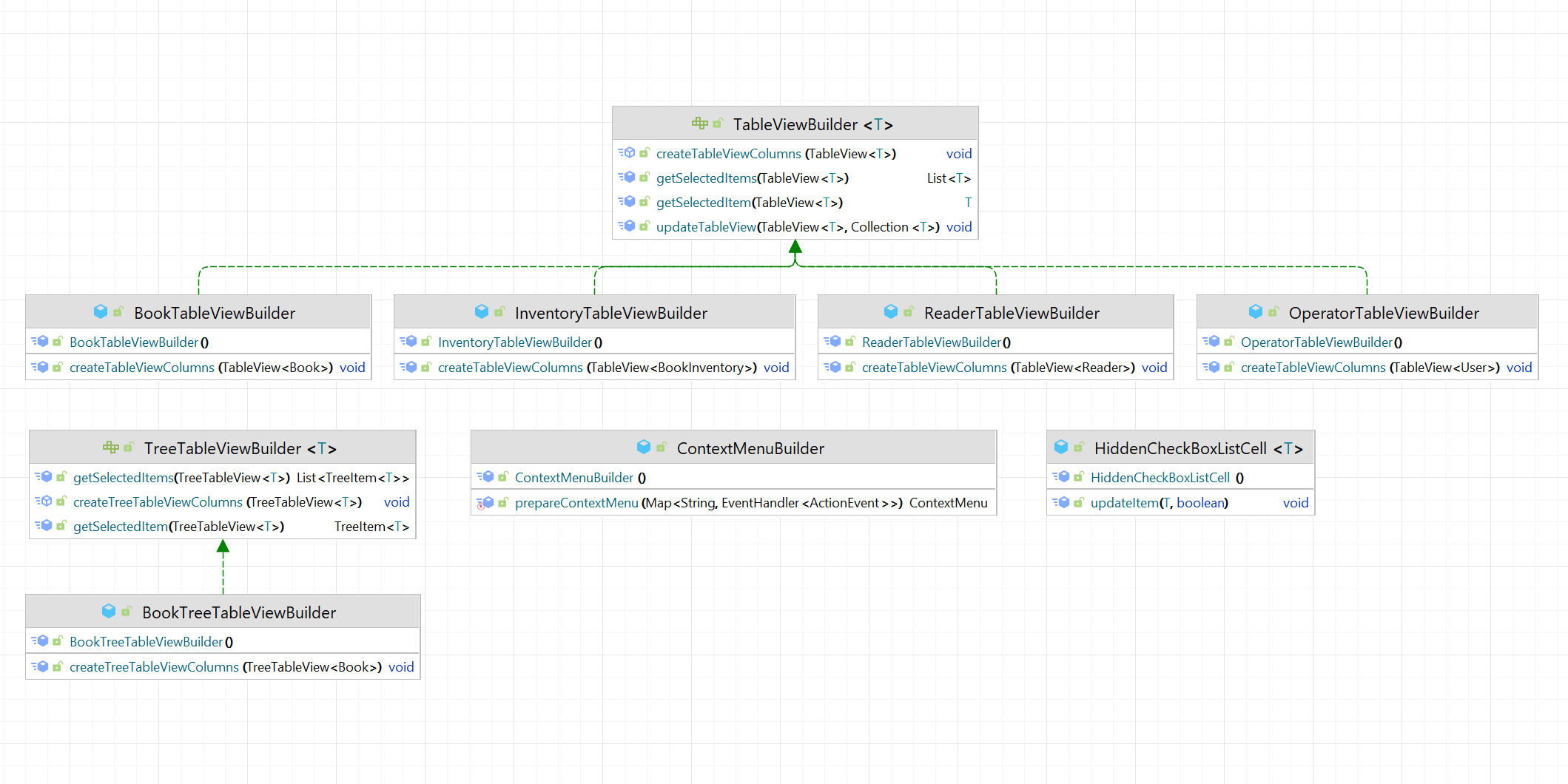
****

****

****

****

****

****

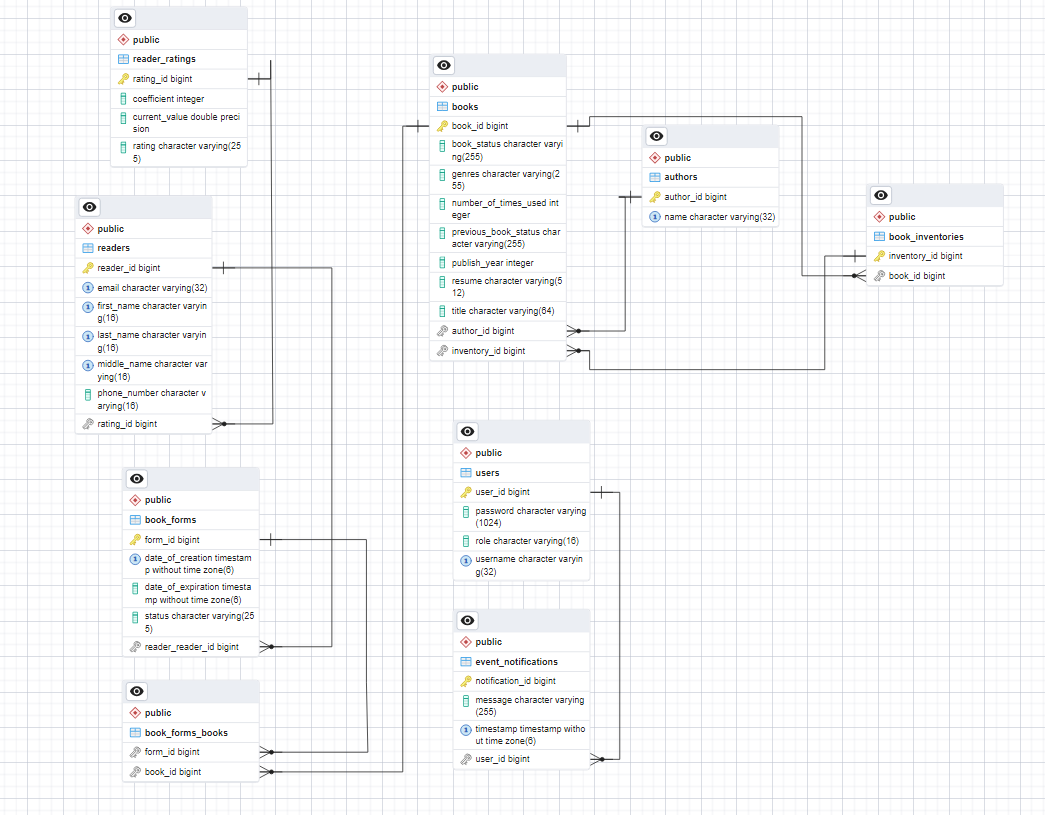
**3.3.Концептуален модел на базата от данни**

1. **Author**
   * Представлява автор на книга.
   * Има уникален идентификатор, име и списък от книги, които той е написал.
2. **Book**
   * Описва книга в библиотечната система.
   * Включва идентификатор, заглавие, резюме, година на издаване, жанр, текущ статус, предишен статус, инвентар и автор.
3. **BookForm**
   * Представя форма за заявка на книга от страна на читател.
   * Включва идентификатор, списък от книги, четец, статус, дата на създаване и дата на изтичане.
4. **BookInventory**
   * Описва инвентар за определена книга в библиотеката.
   * Съдържа уникален идентификатор, списък от книги, които споделят общи характеристики, и представителна книга от този инвентар.
5. **EventNotification**
   * Представя съобщение за събитие, свързано с потребител в системата.
   * Съдържа уникален идентификатор, потребител, съобщение и времева маркировка на събитието.
6. **Reader**
   * Описва читател в системата на библиотеката.
   * Съдържа идентификатор, имена, телефонен номер, имейл адрес, списък от заявки за книги и рейтинг на читателя.
7. **ReaderRating**
   * Представя рейтинг на читателя.
   * Включва уникален идентификатор, оценка, коефициент, текуща стойност на рейтинга.
8. **User**
   * Описва потребител в системата.
   * Съдържа идентификатор, потребителско име, парола, роля и списък от съобщения за събития.

Тези класове описват съответните обекти в системата на библиотеката и техните връзки помежду си. Например, една книга принадлежи на определен автор и може да бъде свързана с форма за заявка от страна на читател. Това представя концептуалния модел на данните, като показва важните ентитети и техните връзки в системата.

**Глава 4. Реализация на системата**

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

****

**Author (Таблица за Автори)**

* **Описание:** Съхранява информация за авторите на книги в библиотеката.
* **Колони:**
  + author\_id: Уникален идентификатор за всеки автор.
  + name: Име на автора.
* **Връзки**:
  + Едно към много с таблицата **Book**.

**Book (Таблица за Книги)**

* **Описание:** Съдържа информация за книгите в библиотеката.
* **Колони:**
  + book\_id: Уникален идентификатор за всяка книга.
  + title: Заглавие на книгата.
  + publish\_year: Година на издаване на книгата.
  + resume: Кратко описание на съдържанието на книгата.
  + genres: Жанр на книгата.
  + book\_status: Състояние на книгата (например, налична, заета и др.).
* **Връзки:**
  + Много към едно с таблицата **Author.**
  + Много към едно с таблицата **BookInventory.**

**BookForm (Таблица за Формуляри за Книги)**

**Описание**: Съхранява информация за формулярите за заемане на книги от читатели.

**Колони:**

form\_id: Уникален идентификатор за всяка форма.

status: Състояние на формуляра (например, използва се, невръщане и др.).

date\_of\_creation: Дата на създаване на формуляра.

expiration\_date: Дата на изтичане на формуляра.

**Връзки:**

Много към много с таблицата **Book.**

**BookInventory (Таблица за Инвентар на Книги)**

* **Описание:** Съдържа информация за наличността на книги в библиотеката.
* **Колони:**
  + inventory\_id: Уникален идентификатор за инвентара.
  + bookList: Колекция от множество книги с идентични заглавия, автори и жанрове.
  + representiveBook: Представя една книга от bookList.
* **Връзки:**
  + Много към едно с таблицата **Book.**

**EventNotification (Таблица за Съобщения за Събития)**

* **Описание:** Съдържа информация за уведомленията за събитията за конкретен потребител.
* **Колони:**
  + notification\_id: Уникален идентификатор за всяко уведомление.
  + message: Съдържание на уведомлението.
  + timestamp: Маркира времето на настъпване на събитието.

**Reader (Таблица за Читатели)**

* **Описание:** Съхранява информация за читателите в библиотеката.
* **Колони:**
  + reader\_id: Уникален идентификатор за всякакъв читател.
  + first\_name, middle\_name, last\_name: Име на читателя.
  + phone\_number: Телефонен номер на читателя.
  + email: Електронна поща на читателя.
* **Връзки:**
  + Много към едно с таблицата BookForm.
  + Едно към едно с таблицата ReaderRating.

**ReaderRating (Таблица за Рейтинг на Читатели)**

* **Описание:** Съдържа информация за рейтинга на читателите.
* **Колони:**
  + rating\_id: Уникален идентификатор за рейтинга.
  + rating: Оценка на читателя (например, от 1 до 5).
  + coefficient: Коефициент за определяне на рейтинга на читателя.
  + current\_value: Текуща стойност на рейтинга.

**User (Таблица за Потребители)**

* **Описание:** Съхранява информация за потребителите на системата.
* **Колони:**
  + **user\_id**: Уникален идентификатор за всякакъв потребител.
  + **username**: Потребителско име на потребителя.
  + **password**: Парола на потребителя.
  + **role**: Роля на потребителя (например, администратор, читател и др.).
* **Връзки:**
  + Много към едно с таблицата **EventNotification**.

**4.2.Реализация на слоя за работа с базата от данни**

Използването на Hibernate за работа с базата данни е от ключово значение. Класовете са анотирани с JPA анотации като: **@Entity,@Table,@Column, @Id, @GeneratedValue, @ManyToOne, @OneToMany,** и други, които указват как да бъдат създадени таблиците и връзките между тях в базата данни.

**Реализация на DAO (Data Access Object) слой:**

* Всяка от класовете представлява съответния обект, който се съхранява в базата данни (например, **Author**, **Book**, **BookForm**, и т.н.).
* Всяка колона от таблиците е представена чрез полетата на класовете с анотации като **@Column**.
* Използват се анотации като **@Id** за уникалния идентификатор на обектите.
* Определени са връзки помежду класовете чрез анотации като **@ManyToOne**, **@OneToMany**, които отразяват връзките между таблиците в базата данни.

**Операции с базата данни:**

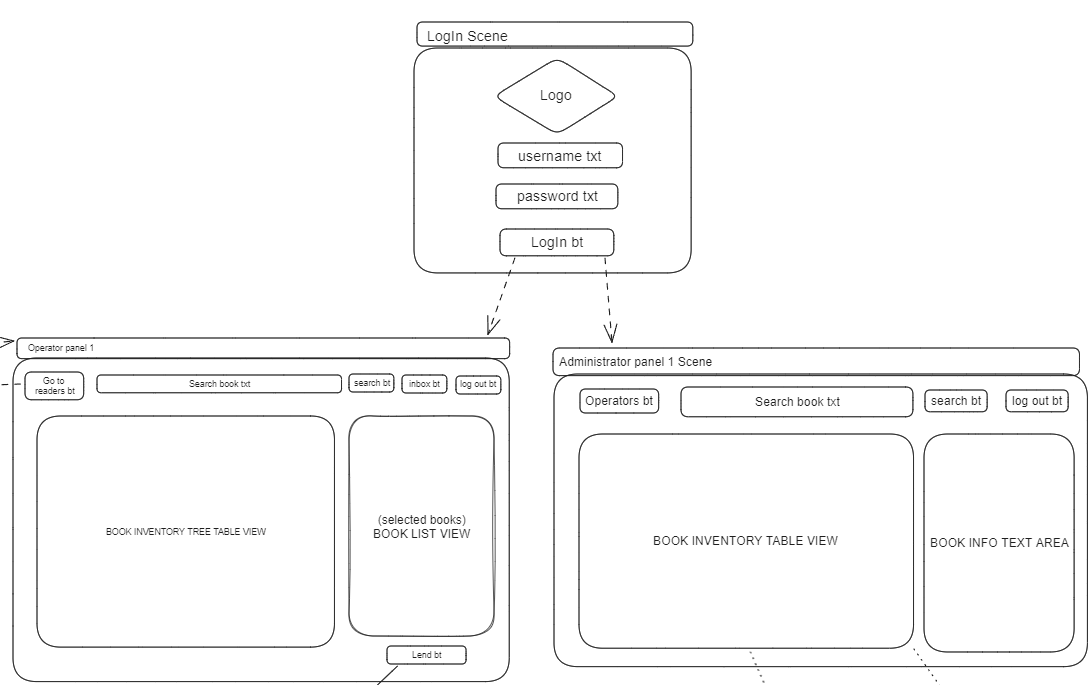
* Добавяне, извличане, обновяване и изтриване на обекти в базата данни се осъществява чрез методи в съответните DAO класове (например, добавяне на нов автор се осъществява чрез метод в **AuthorRepository**).

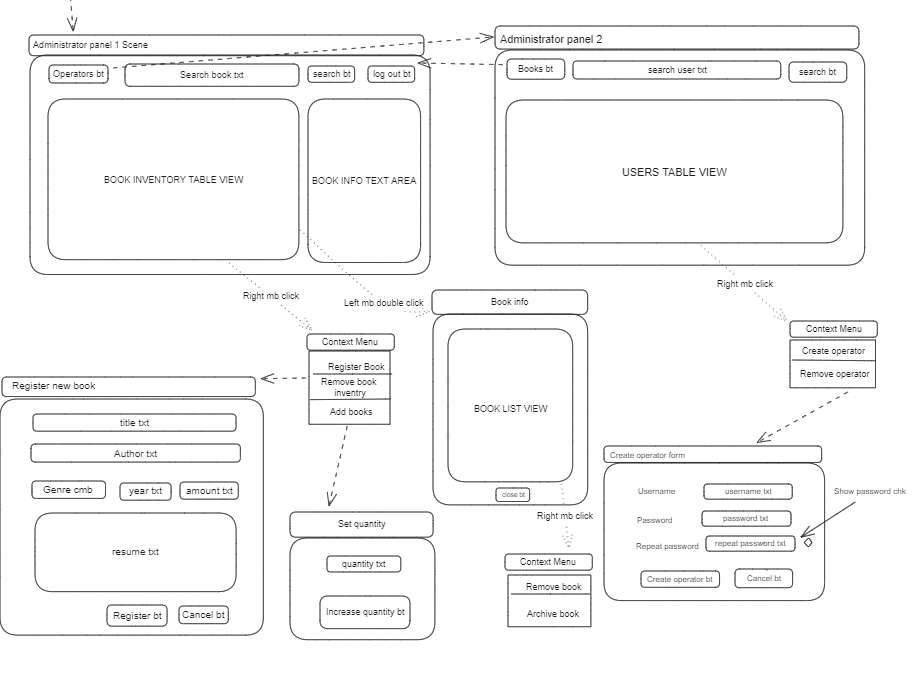
**4.3.Реализация на бизнес логика**

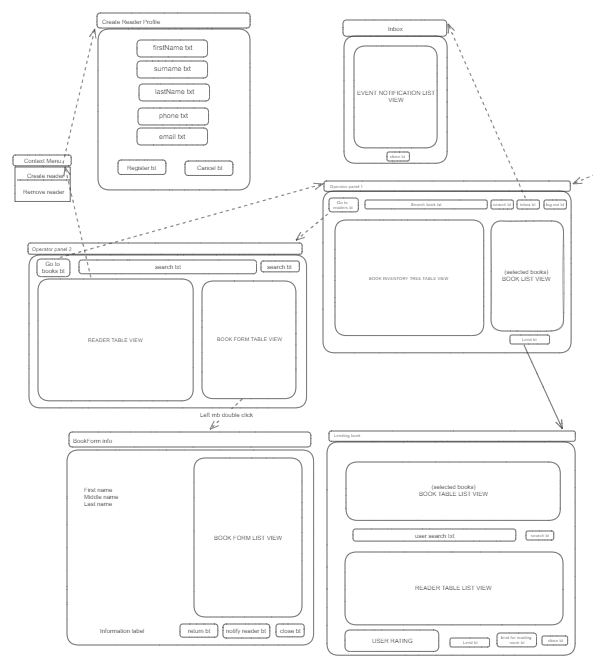
1. **AddBookQuantityService**: Този клас има метод, който увеличава количеството на дадена книга в инвентара. Той създава нови книги и ги свързва с инвентара.
2. **AdministratorBooksDialogService:** Този клас предоставя услуги за администриране на книги, включително премахване на определени книги от инвентара, както и обновяване на свързаната форма за книги.
3. **AdministratorBooksService:** Предоставя функционалности за търсене, премахване и обновяване на инвентари от книги. Също така обновява формите за книги, когато книги биват премахнати от инвентара.
4. **AdministratorOperatorsService:** Този клас управлява операторите в системата, предоставящ търсене на потребители и възможност за премахване на оператори.
5. **BookRegistrationService:** Служи за регистрация на нови книги в системата, като прави проверки за валидност на въведените данни и създава книги и инвентар при необходимост.
6. **CreateOperatorService:** Този клас позволява създаване на нови оператори в системата, като проверява валидността на въведените данни и проверява уникалността на потребителското име.
7. **BookFormShowService:** Този сервиз управлява връщането на книги от читателите. Извършва проверки за състоянието на всяка книга - ако е повредена, задава статуса й на "повредена". Ако не е, книгата става "достъпна" или "архивирана", в зависимост от предишното й състояние. Увеличава брояча на използванията на всяка книга и генерира известие, ако книгата е използвана 50 пъти.
8. **CreateBookFormService:** Този сервиз управлява заемането на книги от читателите. Проверява статуса на читателя и книгите, които иска да вземе. Променя статуса на книгите на "заета" или "в читалнята" и създава форма за заемане на книги.
9. **CreateReaderProfileService:** Създава нов профил на читател в системата. Проверява входните данни за валидност и създава нов читател с празна история на заемане.
10. **InboxService:** Предоставя списък със съобщения към потребителите. Извлича и връща известията, свързани с потребителски профил, като гарантира, че се предоставят само за съответния потребител.
11. **OperatorBooksService:** Управлява книгите в системата, позволявайки операторите да търсят, избират и архивират книги. Осигурява списък с изтекли заемания и обновява статусите на книгите според изискванията.
12. **OperatorReadersService:** Управлява профилите на читателите в системата, позволявайки операторите да търсят, преглеждат и изтриват профили на читатели.
13. **LogInService:** Този клас предоставя функционалност за аутентикация и вход на потребителите в системата. Използва UserRepository, за да провери дали съществува потребител с предоставеното потребителско име. Проверява паролата, като я сравнява със съхранената за съответния потребител. Ако информацията е вярна, връща потребителя, в противен случай хвърля изключение UserNotFoundException.
14. **EmailSenderService:** Този клас предоставя възможност за изпращане на имейли чрез JavaMail API. Съдържа методи за конфигуриране на SMTP настройките, изпращане на имейли и обработка на възможни грешки, като използва предварително конфигуриран Transport обект за изпращане на имейли.

Всеки от тези класове предоставя различни функционалности за управление на книги, читатели, заемания и съобщения в системата. За по лесно инстанциране на класовете се използва маркиращ интерфейс **Service и ServiceFactory.**

**4.4.Реализация на графичен интерфейс**

****

****

****

**4.5.Реализация на модул за тестване на събития в системата**

**JUnit 5**

JUnit 5 е framework за тестване на Java приложения. Той поддържа писането на unit тестове, които проверяват отделни части (units) от кода, като методи, класове или пакети. JUnit 5 предоставя анотации като **@Test**, които маркират методи като тестове, **@BeforeEach** и **@AfterEach**, които се изпълняват преди и след всяко тестване, и други, които позволяват различни настройки на тестовете. Той осигурява богата функционалност за проверка на очаквани резултати и поведение на методите.

**Mockito**

Mockito е Java framework, използван за създаване на мокове (mocks) в юнит тестове. Моковете са фалшиви обекти, които заместват реални зависимости на тестовете, позволявайки изолиране на тестовите сценарии от външни системи или компоненти. Mockito предоставя лесен за използване API за създаване на мокове, настройка на техните поведения с помощта на stubs или mocks и проверка на това как са използвани в тестовете с помощта на verify statements.

**TestFX**

TestFX е инструмент за функционално тестване на графичния потребителски интерфейс на JavaFX приложения. Той позволява създаването на автоматизирани тестове, които симулират действията на потребителя върху графичния интерфейс на приложението. TestFX предоставя богат API, който позволява навигация, манипулация и проверка на компонентите на потребителския интерфейс, като бутони, текстови полета, таблиците и други. Този инструмент помага на разработчиците да напишат тестове, които автоматично проверяват правилното функциониране на графичния интерфейс при промени в приложението без необходимостта от ръчно тестване.

**Глава 5. Тестове**

* 1. Автоматизирани тестове…………………………

import com.library.backend.exception.users.UserNotFoundException;  
import com.library.backend.services.LogInService;  
import com.library.database.entities.User;  
import com.library.database.repositories.UserRepository;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.InjectMocks;  
import org.mockito.Mock;  
import org.mockito.MockitoAnnotations;  
  
import java.util.Optional;  
  
import static org.junit.jupiter.api.Assertions.*assertEquals*;  
import static org.junit.jupiter.api.Assertions.*assertThrows*;  
import static org.mockito.ArgumentMatchers.*anyString*;  
import static org.mockito.ArgumentMatchers.*isNull*;  
import static org.mockito.Mockito.*verify*;  
import static org.mockito.Mockito.*when*;  
  
public class LogInServiceTest {  
  
 @Mock  
 private UserRepository userRepository;  
  
 @InjectMocks  
 private LogInService logInService;  
  
 @BeforeEach  
 public void setUp() {  
 MockitoAnnotations.*openMocks*(this);  
 }  
  
 @Test  
 public void testGetUserUserNotFound() throws UserNotFoundException {  
 *// Arrange* User testUser = new User();  
 testUser.setUsername("nonExistentUser");  
  
 *// Mocking the behavior of UserRepository  
 when*(userRepository.findByUsername("nonExistentUser")).thenReturn(Optional.*empty*());  
  
 *// Act & Assert  
 assertThrows*(UserNotFoundException.class, () -> logInService.getUser("nonExistentUser", null));  
 *verify*(userRepository).findByUsername("nonExistentUser");  
 }  
  
 @Test  
 public void testGetUserWithNullUsername() throws UserNotFoundException {  
 *// Arrange* User testUser = new User();  
 testUser.setUsername(null);  
  
 *// Act & Assert  
 assertThrows*(UserNotFoundException.class, () -> logInService.getUser(null, null));  
 *verify*(userRepository).findByUsername(*isNull*());  
 }  
  
  
 @Test  
 public void testGetUserWithEmptyUsername() throws UserNotFoundException {  
 *// Arrange* User testUser = new User();  
 testUser.setUsername("");  
  
 *// Mocking the behavior of UserRepository  
 when*(userRepository.findByUsername("")).thenReturn(Optional.*empty*());  
  
 *// Act & Assert  
 assertThrows*(UserNotFoundException.class, () -> logInService.getUser("", null));  
 *verify*(userRepository).findByUsername("");  
 }  
  
 @Test  
 public void testGetUserWithWhitespaceUsername() throws UserNotFoundException {  
 *// Arrange* User testUser = new User();  
 testUser.setUsername(" ");  
  
 *// Mocking the behavior of UserRepository  
 when*(userRepository.findByUsername(" ")).thenReturn(Optional.*empty*());  
  
 *// Act & Assert  
 assertThrows*(UserNotFoundException.class, () -> logInService.getUser(" ", null));  
 *verify*(userRepository).findByUsername(" ");  
 }  
  
 @Test  
 public void testGetUserWithNonexistentUsername() throws UserNotFoundException {  
 *// Arrange* User testUser = new User();  
 testUser.setUsername("nonexistentUser");  
  
 *// Mocking behavior  
 when*(userRepository.findByUsername(*anyString*())).thenReturn(Optional.*empty*());  
  
 *// Act & Assert  
 assertThrows*(UserNotFoundException.class, () -> logInService.getUser("nonexistentUser", null));  
 *verify*(userRepository).findByUsername("nonexistentUser");  
 }  
  
 @Test  
 public void testGetUserWithValidCredentials() throws UserNotFoundException {  
 *// Arrange* User testUser = new User();  
 testUser.setUsername("validUser");  
 testUser.setPassword("validPassword");  
  
 *// Mocking behavior  
 when*(userRepository.findByUsername(*anyString*())).thenReturn(Optional.*of*(testUser));  
  
 *// Act* User resultUser = logInService.getUser("validUser", "validPassword");  
  
 *// Assert  
 assertEquals*(testUser, resultUser);  
 *verify*(userRepository).findByUsername("validUser");  
 }  
  
 @Test  
 public void testGetUserWithInvalidCredentials() throws UserNotFoundException {  
 *// Arrange* User testUser = new User();  
 testUser.setUsername("validUser");  
 testUser.setPassword("invalidPassword");  
  
 *// Mocking behavior  
 when*(userRepository.findByUsername(*anyString*())).thenReturn(Optional.*empty*());  
  
 *// Act & Assert  
 assertThrows*(UserNotFoundException.class, () -> logInService.getUser("validUser", "invalidPassword"));  
 *verify*(userRepository).findByUsername("validUser");  
 }  
}

import com.library.backend.validators.StrongPasswordValidator;  
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.*assertFalse*;  
import static org.junit.jupiter.api.Assertions.*assertTrue*;  
  
class StrongPasswordValidatorTest {  
  
 @Test  
 void testValidPassword() {  
 *// Arrange* StrongPasswordValidator validator = new StrongPasswordValidator();  
  
 *// Act* boolean isValid = validator.isValid("StrongPwd1");  
  
 *// Assert  
 assertTrue*(isValid, "The password should be considered valid");  
 }  
  
 @Test  
 void testInvalidPasswordNoDigit() {  
 *// Arrange* StrongPasswordValidator validator = new StrongPasswordValidator();  
  
 *// Act* boolean isValid = validator.isValid("NoDigitPwd");  
  
 *// Assert  
 assertFalse*(isValid, "The password should be considered invalid (missing digit)");  
 }  
  
 @Test  
 void testInvalidPasswordNoLowercase() {  
 *// Arrange* StrongPasswordValidator validator = new StrongPasswordValidator();  
  
 *// Act* boolean isValid = validator.isValid("NOCASEPWD1");  
  
 *// Assert  
 assertFalse*(isValid, "The password should be considered invalid (missing lowercase)");  
 }  
  
 @Test  
 void testInvalidPasswordNoUppercase() {  
 *// Arrange* StrongPasswordValidator validator = new StrongPasswordValidator();  
  
 *// Act* boolean isValid = validator.isValid("nocasepwd1");  
  
 *// Assert  
 assertFalse*(isValid, "The password should be considered invalid (missing uppercase)");  
 }  
  
 @Test  
 void testInvalidPasswordWhitespace() {  
 *// Arrange* StrongPasswordValidator validator = new StrongPasswordValidator();  
  
 *// Act* boolean isValid = validator.isValid("Pwd With Spaces1");  
  
 *// Assert  
 assertFalse*(isValid, "The password should be considered invalid (contains whitespace)");  
 }  
  
 @Test  
 void testInvalidPasswordShortLength() {  
 *// Arrange* StrongPasswordValidator validator = new StrongPasswordValidator();  
  
 *// Act* boolean isValid = validator.isValid("Sh1");  
  
 *// Assert  
 assertFalse*(isValid, "The password should be considered invalid (short length)");  
 }  
  
 @Test  
 void testInvalidPasswordNull() {  
 *// Arrange* StrongPasswordValidator validator = new StrongPasswordValidator();  
  
 *// Act* boolean isValid = validator.isValid(null);  
  
 *// Assert  
 assertFalse*(isValid, "Null password should be considered invalid");  
 }  
  
 @Test  
 void testValidPasswordEdgeCaseMinLength() {  
 *// Test a password with the minimum length (6 characters)* StrongPasswordValidator validator = new StrongPasswordValidator();  
 boolean isValid = validator.isValid("Min6CA");  
  
 *assertTrue*(isValid, "The password should be considered valid (minimum length)");  
 }  
  
 @Test  
 void testValidPasswordEdgeCaseMaxLength() {  
 *// Test a password with the maximum length (assuming the maximum length is allowed)* StrongPasswordValidator validator = new StrongPasswordValidator();  
 *assertTrue*(validator.isValid("MaxPassword123456789"), "The password should be considered valid (maximum length)");  
 }  
  
 @Test  
 void testInvalidPasswordEdgeCaseMaxLengthExceeded() {  
 *// Test a password that exceeds the assumed maximum length* StrongPasswordValidator validator = new StrongPasswordValidator();  
 *assertFalse*(validator.isValid("TooLongPassword12345678901"), "The password should be considered invalid (exceeds maximum length)");  
 }  
}

* 1. Тестови резултати на апликацията

Тестовите резултати с подробности можете да видите в папката htmlReport на проекта.