Технически университет – Варна

Факултет: ФИТА

Специалност: СИТ

Дисциплина:  
Обектно-ориентирано програмиране – част 2 – проект

Изготвен от:

Анатоли Яворов Димов

Ф.н. 20621662

Съдържание:

Увод:

* Условие:

A white text with black text

Description automatically generated

* Интерпретация на условието

Приложението е предвидено да се използва от служители на библиотека – администратор и оператори. Целта е да се улеснят и автоматизират процеси, свързани с управлението на библиотеката – читатели, книги и подобни.

Читателите взаимодействат с операторите на библиотеката, които имат основната роля във взаимодействието клиент-бизнес.

Те имат следните възможности:

* Заявяване на книга – операторът открива читателски профил(или използва съществуващия за човека такъв), и отдава книгата
* Връщане на книги - книгата се приема от оператор, като спрямо състоянието в което е върната се афектира рейтингът на читателя
* Нямат отговорности в системата

Възможностите на оператор са следните:

* Справки за книги и екземпляри
* Създаване и премахване на записи за читатели
* Справки за читатели
* Приемане на книги от читател
* Отдаване на книги на читател
* Известяване на читател за връщане на книги

Възможности на администратор:

* Създаване на оператори
* Премахване на оператори
* Регистриране на книги
* Добавяне на екземпляри на книги
* Премахване на архивирани екземпляри на книги
* Премахване на повредени екземпляри на книги
* Проучване и избор на технологии

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

**Spring Boot**

Spring Boot е рамка за разработка на Java приложения, dependency-injection framework, която има за цел да улесни създаването на готови за производство, самостоятелни приложения. Тя предоставя много вградени функции и инструменти, които помагат на разработчиците да се съсредоточат върху логиката на приложението, без да се занимават с конфигурационни подробности.

**Spring Boot Starter Data JPA**

Интеграция с JPA (Java Persistence API): Осигурява лесна работа с релационни бази данни чрез Hibernate, като облекчава конфигурацията и настройката на JPA в проекта.

Улеснена работа с релационни бази данни: Spring Data JPA автоматизира създаването на репозитории, които позволяват изпълнение на CRUD операции (създаване, четене, обновяване, изтриване) без нужда от писане на обикновен SQL код.

Автоматична конфигурация: Включва всички необходими зависимости за стартиране на JPA проекти, като конфигуриране на EntityManager и DataSource автоматично.

Примери на използване в кода:

@Getter  
@Setter  
@Builder  
@NoArgsConstructor  
@AllArgsConstructor  
@Entity  
@Table(name = "bookitems")  
public class BookItem {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*UUID*)  
 @Column(name = "bookitem\_id", nullable = false)  
 private UUID id;  
  
 @ManyToOne  
 @JoinColumn(name = "book\_id", nullable = false)  
 private Book book;  
  
 @ManyToOne  
 private Reader reader;  
  
 @Column  
 @Enumerated(EnumType.*STRING*)  
 private BookStatus bookStatus;  
}

@Getter  
@Setter  
@AllArgsConstructor  
@NoArgsConstructor  
@Builder  
@Entity  
@Table(name = "reader")  
public class Reader {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*UUID*)  
 @Column(name = "reader\_id")  
 private UUID id;  
 @Column  
 private String firstName;  
 @Column  
 private String lastName;  
 @Column  
 private String phoneNumber;  
 @Column  
 private String email;  
 @OneToMany(mappedBy = "reader")  
 private List<BookItem> bookItemList;  
 @Column  
 @Min(1)  
 @Max(5)  
 private Integer readerRating;  
}

**PostgreSQL**

PostgreSQL е мощна, open source SQL база данни, която предлага множество функции и възможности за разработчиците. Използването на PostgreSQL в Java приложенията се осъществява чрез JDBC драйвери, които позволяват лесно интегриране и работа с базата данни. Използван е и JDBC драйверът за PostgreSQL, който осигурява комуникацията между Java приложението и PostgreSQL базата данни. Той поддържа всички функции на PostgreSQL и предоставя необходимите инструменти за работа с базата данни от Java код.

Ключови предимства на PostgreSQL:

Сигурност: Поддържа SSL, разширени механизми за автентикация и контрол на достъпа, което гарантира висока степен на сигурност на данните.

Съвместимост: Работи на множество операционни системи и поддържа стандарти като SQL:2008.

Мащабируемост: Лесно се мащабира както вертикално (с повече ресурси на един сървър), така и хоризонтално (с повече сървъри).

**OpenJFX**

OpenJFX е официалната отворена реализация на JavaFX, която предоставя мощен набор от инструменти за създаване на модерни, богати на функции графични потребителски интерфейси.

Ключови функционалности на OpenJFX:

Сцени и сцени граф: Позволява създаването на сцени (scenes) и сцени графове (scene graphs), които организират визуалните компоненти в йерархична структура.

Контроли: Включва богат набор от вградени контроли като бутони, текстови полета, таблици и други, които могат лесно да бъдат използвани и персонализирани.

CSS стилове: Поддържа стилизиране на компоненти чрез CSS, което позволява създаване на персонализирани и атрактивни потребителски интерфейси.

Мултимедия: Поддържа възпроизвеждане на аудио и видео съдържание, което добавя мултимедийни възможности към приложенията.

Графики: Включва поддръжка за рисуване на графики и диаграми, което е полезно за визуализиране на данни.

FXML: XML-базиран маркъп език, който позволява декларативно дефиниране на потребителския интерфейс, отделяйки логиката от визията.

Примери:

* Javafx file:
* <?xml version="1.0" encoding="UTF-8"?>  
    
  <?import javafx.scene.control.Button?>  
  <?import javafx.scene.control.TableColumn?>  
  <?import javafx.scene.control.TableView?>  
  <?import javafx.scene.layout.AnchorPane?>  
  <?import javafx.scene.layout.BorderPane?>  
    
  <BorderPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0" xmlns="http://javafx.com/javafx/21" xmlns:fx="http://javafx.com/fxml/1" fx:controller="bg.tuvarna.frontend.controller.OperatorController">  
   <bottom>  
   <AnchorPane prefHeight="400.0" prefWidth="600.0" BorderPane.alignment="CENTER">  
   <children>  
   <TableView fx:id="booksTableView" layoutX="322.0" layoutY="75.0" prefHeight="200.0" prefWidth="250.0">  
   <columns>  
   <TableColumn fx:id="bookTitleColumn" prefWidth="100.0" text="Title" />  
   <TableColumn fx:id="bookAuthorColumn" prefWidth="100.0" text="Author" />  
   <TableColumn fx:id="bookQuantityColumn" prefWidth="50.0" text="Quantity" />  
   <TableColumn fx:id="bookIsbnColumn" prefWidth="100.0" text="ISBN" />  
   </columns>  
   </TableView>  
   <TableView fx:id="readersTableView" layoutX="28.0" layoutY="75.0" prefHeight="200.0" prefWidth="250.0">  
   <columns>  
   <TableColumn fx:id="readerFirstNameColumn" prefWidth="100.0" text="First Name" />  
   <TableColumn fx:id="readerLastNameColumn" prefWidth="100.0" text="Last Name" />  
   <TableColumn fx:id="readerPhoneNumberColumn" prefWidth="100.0" text="Phone Number" />  
   <TableColumn fx:id="readerEmailColumn" prefWidth="150.0" text="Email" />  
   <TableColumn fx:id="readerRatingColumn" prefWidth="50.0" text="Rating" />  
   </columns>  
   </TableView>  
   <Button fx:id="createReaderButton" layoutX="48.0" layoutY="305.0" mnemonicParsing="false" onAction="#createReader" text="Create Reader" />  
   <Button fx:id="deleteReaderButton" layoutX="145.0" layoutY="305.0" mnemonicParsing="false" onAction="#removeReader" text="Delete Reader" />  
   <Button fx:id="logoutButton" layoutX="507.0" layoutY="25.0" mnemonicParsing="false" onAction="#logout" text="Logout" />  
   <Button fx:id="lendBookButton" layoutX="411.0" layoutY="305.0" mnemonicParsing="false" onAction="#lendBook" text="Lend Book" />  
   <Button fx:id="notifyButton" layoutX="108.0" layoutY="349.0" mnemonicParsing="false" onAction="#notifyReader" text="Notify Reader" />  
   </children>  
   </AnchorPane>  
   </bottom>  
  </BorderPane>
* Javafx controller:

package bg.tuvarna.frontend.controller;  
  
import bg.tuvarna.api.operations.administrator.addbookitems.AddBookItems;  
import bg.tuvarna.api.operations.administrator.addbookitems.AddBookItemsInput;  
import bg.tuvarna.api.operations.administrator.getoperators.GetOperators;  
import bg.tuvarna.api.operations.administrator.getoperators.GetOperatorsInput;  
import bg.tuvarna.api.operations.administrator.removearchivedbooks.RemoveArchivedBookItems;  
import bg.tuvarna.api.operations.administrator.removearchivedbooks.RemoveArchivedBookItemsInput;  
import bg.tuvarna.api.operations.administrator.removedamagedbooks.RemoveDamagedBookItems;  
import bg.tuvarna.api.operations.administrator.removedamagedbooks.RemoveDamagedBookItemsInput;  
import bg.tuvarna.api.operations.administrator.removeoperator.RemoveOperator;  
import bg.tuvarna.api.operations.administrator.removeoperator.RemoveOperatorInput;  
import bg.tuvarna.api.operations.user.getbooks.GetBooks;  
import bg.tuvarna.api.operations.user.getbooks.GetBooksInput;  
import bg.tuvarna.api.operations.user.logout.UserLogout;  
import bg.tuvarna.api.operations.user.logout.UserLogoutInput;  
import bg.tuvarna.api.operations.util.BookDTO;  
import bg.tuvarna.api.operations.util.OperatorDTO;  
import bg.tuvarna.frontend.utils.SceneChanger;  
import javafx.collections.FXCollections;  
import javafx.collections.ObservableList;  
import javafx.fxml.FXML;  
import javafx.scene.control.\*;  
import javafx.scene.control.cell.PropertyValueFactory;  
import javafx.stage.Stage;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
import java.util.Optional;  
  
@Component  
@Slf4j  
public class AdminController {  
  
 @FXML  
 private Button addBookItemsButton;  
  
 @FXML  
 private TableColumn<?, ?> bookAuthorColumn;  
 @FXML  
  
 private TableColumn<?, ?> bookQuantityColumn;  
 @FXML  
 private TableColumn<?, ?> bookIsbnColumn;  
  
 @FXML  
 private TableView bookTable;  
  
 @FXML  
 private TableColumn<?, ?> bookTitleColumn;  
  
 @FXML  
 private Button createOperatorButton;  
  
 @FXML  
 private Button logoutButton;  
  
 @FXML  
 private TableColumn<?, ?> operatorEmailColumn;  
 @FXML  
 private TableColumn<?, ?> operatorIdColumn;  
  
 @FXML  
 private TableView<OperatorDTO> operatorTable;  
  
 @FXML  
 private Button registerBookButton;  
  
 @FXML  
 private Button removeArchivedBookItemsButton;  
  
 @FXML  
 private Button removeDamagedBookItemsButton;  
  
 @FXML  
 private Button removeOperatorButton;  
  
 @Autowired  
 private UserLogout userLogout;  
 @Autowired  
 private GetOperators getOperators;  
 @Autowired  
 private SceneChanger sceneChanger;  
 @Autowired  
 private RemoveOperator removeOperator;  
 @Autowired  
 private GetBooks getBooks;  
 @Autowired  
 private AddBookItems addBookItems;  
 @Autowired  
 private RemoveArchivedBookItems removeArchivedBookItems;  
 @Autowired  
 private RemoveDamagedBookItems removeDamagedBookItems;  
  
 @Value("${fxml.paths.loginForm}")  
 private String loginFormPath;  
 @Value("${fxml.paths.createOperatorForm}")  
 private String createOperatorFormPath;  
 @Value("${fxml.paths.registerBookForm}")  
 private String registerBookFormPath;  
  
 @FXML  
 void createOperator() {  
 sceneChanger.changeScene((Stage) logoutButton.getScene().getWindow(), createOperatorFormPath);  
 }  
  
 @FXML  
 void registerBook() {  
 sceneChanger.changeScene((Stage) registerBookButton.getScene().getWindow(), registerBookFormPath);  
 }  
  
 @FXML  
 void logout() {  
 try {  
 userLogout.process(new UserLogoutInput());  
  
 sceneChanger.changeScene((Stage) logoutButton.getScene().getWindow(), loginFormPath);  
 *log*.info("User logged out successfully");  
 } catch (Exception e) {  
 *log*.error(e.getMessage());  
// e.printStackTrace();  
 }  
 }  
  
 @FXML  
 public void initialize() {  
 initOperators();  
 initBooks();  
 }  
  
 @FXML  
 void removeOperator() {  
 // Get the selected item from the TableView  
 Optional.*ofNullable*(operatorTable.getSelectionModel().getSelectedItem()).ifPresentOrElse(  
 selectedOperator -> {  
 // Show a confirmation dialog  
 Alert confirmationAlert = new Alert(Alert.AlertType.*CONFIRMATION*);  
 confirmationAlert.setTitle("Confirmation");  
 confirmationAlert.setHeaderText(null);  
 confirmationAlert.setContentText("Are you sure you want to delete the operator " + selectedOperator.getEmail() + "?");  
  
 Optional<ButtonType> result = confirmationAlert.showAndWait();  
 if (result.isPresent() && result.get() == ButtonType.*OK*) {  
 // Create input for RemoveOperator process  
 RemoveOperatorInput input = RemoveOperatorInput.*builder*()  
 .operatorEmail(selectedOperator.getEmail())  
 .build();  
  
 // Process the removal  
 removeOperator.process(input);  
  
 // Optionally, you might want to remove the item from the table view  
 operatorTable.getItems().remove(selectedOperator);  
 }  
 },  
 () -> {  
 // Handle the case when no item is selected  
 Alert alert = new Alert(Alert.AlertType.*WARNING*);  
 alert.setTitle("Warning");  
 alert.setHeaderText(null);  
 alert.setContentText("Please select an operator to remove.");  
 alert.showAndWait();  
 }  
 );  
 }  
 @FXML  
 void removeArchivedBookItems(){  
 removeArchivedBookItems.process(RemoveArchivedBookItemsInput.*builder*().build());  
 }  
 @FXML  
 void removeDamagedBookItems(){  
 removeDamagedBookItems.process(RemoveDamagedBookItemsInput.*builder*().build());  
 }  
  
 @FXML  
 void addBookItems() {  
 // Get the selected book from the TableView  
 Optional<BookDTO> selectedBook = Optional.*ofNullable*((BookDTO) bookTable.getSelectionModel().getSelectedItem());  
  
 if (selectedBook.isPresent()) {  
 // Show an input dialog to get the quantity  
 TextInputDialog dialog = new TextInputDialog();  
 dialog.setTitle("Add Book Items");  
 dialog.setHeaderText("Add new items for the book: " + selectedBook.get().getTitle());  
 dialog.setContentText("Please enter the quantity:");  
  
 Optional<String> result = dialog.showAndWait();  
  
 result.ifPresent(quantityStr -> {  
 try {  
 int quantity = Integer.*parseInt*(quantityStr);  
  
 // Validate the quantity  
 if (quantity <= 0) {  
 Alert alert = new Alert(Alert.AlertType.*ERROR*);  
 alert.setTitle("Invalid Quantity");  
 alert.setHeaderText(null);  
 alert.setContentText("The quantity must be a positive integer.");  
 alert.showAndWait();  
 return;  
 }  
  
 // Create input for AddBookItems process  
 AddBookItemsInput input = AddBookItemsInput.*builder*()  
 .isbn(selectedBook.get().getIsbn())  
 .quantity(quantity)  
 .build();  
  
 // Process adding book items  
 addBookItems.process(input);  
  
 // Optionally, show a confirmation dialog  
 Alert confirmationAlert = new Alert(Alert.AlertType.*INFORMATION*);  
 confirmationAlert.setTitle("Success");  
 confirmationAlert.setHeaderText(null);  
 confirmationAlert.setContentText(quantity + " items have been added for the book: " + selectedBook.get().getTitle());  
 confirmationAlert.showAndWait();  
  
 // Refresh the book list (if necessary)  
 initBooks();  
  
 } catch (NumberFormatException e) {  
 Alert alert = new Alert(Alert.AlertType.*ERROR*);  
 alert.setTitle("Invalid Input");  
 alert.setHeaderText(null);  
 alert.setContentText("Please enter a valid number.");  
 alert.showAndWait();  
 }  
 });  
 } else {  
 // Handle the case when no book is selected  
 Alert alert = new Alert(Alert.AlertType.*WARNING*);  
 alert.setTitle("Warning");  
 alert.setHeaderText(null);  
 alert.setContentText("Please select a book to add items.");  
 alert.showAndWait();  
 }  
 }  
  
  
 private void initOperators() {  
  
 operatorEmailColumn.setCellValueFactory(new PropertyValueFactory<>("email"));  
 operatorIdColumn.setCellValueFactory(new PropertyValueFactory<>("id"));  
  
  
 ObservableList<OperatorDTO> operators = FXCollections.*observableArrayList*(  
 getOperators.process(new GetOperatorsInput()).getOperators()  
 );  
 operatorTable.setItems(operators);  
 }  
  
 private void initBooks() {  
 bookTitleColumn.setCellValueFactory(new PropertyValueFactory<>("title"));  
 bookAuthorColumn.setCellValueFactory(new PropertyValueFactory<>("author"));  
 bookQuantityColumn.setCellValueFactory(new PropertyValueFactory<>("quantity"));  
 bookIsbnColumn.setCellValueFactory(new PropertyValueFactory<>("isbn"));  
  
 ObservableList<BookDTO> bookDTOS = FXCollections.*observableArrayList*(  
 getBooks.process(new GetBooksInput()).getBookDTOList()  
 );  
 bookTable.setItems(bookDTOS);  
 }  
  
}

**SLF4J (Simple Logging Facade for Java)**

SLF4J (Simple Logging Facade for Java) е библиотека за логване, която предоставя универсален интерфейс за различни логващи фреймуъркове. Това позволява на разработчиците да използват различни логващи библиотеки в своите приложения без да се налага да променят кода си при смяна на библиотеката.

Пример за използване:

public RemoveArchivedBookItemsResult process(RemoveArchivedBookItemsInput input) {  
 List<BookItem> archivedBookItems = bookItemRepository.findByBookStatus(BookStatus.*ARCHIVED*);  
 int removedCount = archivedBookItems.size();  
  
 bookItemRepository.deleteAll(archivedBookItems);  
 archivedBookItems.forEach(bookItem -> *log*.info("Removed Book with ID: {}", bookItem.getId()));  
  
 *log*.info("Removed {} archived book items", removedCount);  
  
 return RemoveArchivedBookItemsResult.*builder*()  
 .removedCount(removedCount)  
 .build();  
}

Ключови функционалности на SLF4J:

Универсален интерфейс: Предоставя общ API, който може да бъде имплементиран с различни логващи фреймуъркове като Logback, Log4j и други.

Абстракция: SLF4J абстрахира реалната имплементация на логването, което улеснява подмяната на логващия фреймуърк без промени в кода на приложението.

Гъвкавост: Поддържа различни логващи системи чрез различни "binding" модули, което позволява лесно конфигуриране на логването.

Основни предимства на SLF4J

Лесна интеграция: Позволява лесна интеграция с различни логващи фреймуъркове, което улеснява поддръжката и миграцията.

Производителност: Предлага оптимизации, които намаляват въздействието на логването върху производителността на приложението.

Гъвкавост: Позволява конфигуриране на логването според нуждите на проекта и различни среди (разработка, тест, продукция).

**Lombok**

Lombok е библиотека за Java, която предоставя анотации, които автоматично генерират код за общи операции като гетъри, сетъри, конструктори, методи за toString(), equals() и hashCode(), и други. Целта на Lombok е да опрости и улесни писането на по-кратък и по-четлив код, като намали шаблонния код (boilerplate code).

Ключови функционалности на Lombok:

Гетъри и Сетъри: Автоматично генерира гетъри и сетъри за полета в класовете.

Конструктори: Генерира различни типове конструктори (без аргументи, с всички аргументи и други).

toString(), equals() и hashCode() методи: Генерира тези методи на базата на полетата в класа.

Builder шаблон: Позволява лесно използване на Builder шаблона за създаване на обекти.

Логиране: Генерира логиращи полета за различни логиращи фреймуъркове. **Hibernate Validator**

Предоставя инструменти за валидация на обекти и данни в Java приложения, като поддържат декларативна валидация чрез анотации.

Основни предимства на Lombok

Намалява шаблонния код: Автоматично генерира код за често използвани операции, което прави кода по-четлив и лесен за поддръжка.

Подобрява производителността на разработчиците: Позволява на разработчиците да се фокусират върху бизнес логиката, вместо върху писането на шаблонен код.

Интеграция с различни инструменти: Работи добре с различни инструменти за разработка и рамки.

**Mockito и JUnit**

Mockito и JUnit са две от най-популярните библиотеки за писане на тестове в Java. Те често се използват заедно, за да осигурят изчерпателно тестване на приложението, включително единични тестове (unit tests) и интеграционни тестове.

Mockito

Mockito е рамка за мокиране в Java, която позволява създаването на мока (mock) обекти за тестване. Тези обекти могат да бъдат използвани за симулиране на поведението на реални обекти по време на тестовете.

Ключови функционалности на Mockito:

Създаване на мок обекти: Позволява лесно създаване на мок обекти.

Настройка на поведение: Позволява задаване на поведение за мок обекти.

Верификация на взаимодействия: Позволява проверка дали определени методи са били извикани върху мок обектите.

Поддръжка на анотации: Предоставя анотации за улесняване на конфигурацията на мок обекти в тестовите класове.

Основни предимства на Mockito и JUnit

Лесно писане на тестове: Позволяват лесно писане на единични тестове и мок тестове, което подобрява качеството на кода.

Интеграция с други инструменти: Лесно се интегрират с различни инструменти за изграждане и CI/CD системи.

Широка поддръжка и общност: Имат голяма общност и много ресурси за обучение и поддръжка.

Използването на Mockito и JUnit заедно предоставя мощен инструментариум за тестване на Java приложения, осигурявайки високо качество и надеждност на кода.

Пример за използване:

package bg.tuvarna.core.service;  
  
import bg.tuvarna.api.BookStatus;  
import bg.tuvarna.api.operations.operator.changebookstatus.ChangeBookStatusInput;  
import bg.tuvarna.api.operations.operator.changebookstatus.ChangeBookStatusResult;  
import bg.tuvarna.persistence.entity.BookItem;  
import bg.tuvarna.persistence.repository.BookItemRepository;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.ArgumentCaptor;  
import org.mockito.InjectMocks;  
import org.mockito.Mock;  
import org.mockito.MockitoAnnotations;  
  
import java.util.Optional;  
import java.util.UUID;  
  
import static org.junit.jupiter.api.Assertions.*assertEquals*;  
import static org.junit.jupiter.api.Assertions.*assertThrows*;  
import static org.mockito.Mockito.\*;  
  
class ChangeBookStatusServiceTest {  
  
 @Mock  
 private BookItemRepository bookItemRepository;  
  
 @InjectMocks  
 private ChangeBookStatusService changeBookStatusService;  
  
 @BeforeEach  
 void setUp() {  
 MockitoAnnotations.*openMocks*(this);  
 }  
  
 @Test  
 void testProcessInvalidBookItemId() {  
 String invalidBookItemId = "invalid-uuid";  
 ChangeBookStatusInput input = ChangeBookStatusInput.*builder*()  
 .bookItemId(invalidBookItemId)  
 .bookStatus(BookStatus.*DAMAGED*)  
 .build();  
  
 Exception exception = *assertThrows*(IllegalArgumentException.class, () -> {  
 changeBookStatusService.process(input);  
 });  
  
 *assertEquals*("Invalid book item ID", exception.getMessage());  
 }  
  
 @Test  
 void testProcessBookItemNotFound() {  
 String bookItemId = UUID.*randomUUID*().toString();  
 ChangeBookStatusInput input = ChangeBookStatusInput.*builder*()  
 .bookItemId(bookItemId)  
 .bookStatus(BookStatus.*DAMAGED*)  
 .build();  
  
 *when*(bookItemRepository.findById(UUID.*fromString*(bookItemId))).thenReturn(Optional.*empty*());  
  
 Exception exception = *assertThrows*(IllegalArgumentException.class, () -> {  
 changeBookStatusService.process(input);  
 });  
  
 *assertEquals*("Book item not found", exception.getMessage());  
 }  
  
 @Test  
 void testProcessBookStatusChangedSuccessfully() {  
 String bookItemId = UUID.*randomUUID*().toString();  
 ChangeBookStatusInput input = ChangeBookStatusInput.*builder*()  
 .bookItemId(bookItemId)  
 .bookStatus(BookStatus.*DAMAGED*)  
 .build();  
  
 BookItem bookItem = new BookItem();  
 bookItem.setId(UUID.*fromString*(bookItemId));  
 bookItem.setBookStatus(BookStatus.*AVAILABLE*);  
  
 *when*(bookItemRepository.findById(UUID.*fromString*(bookItemId))).thenReturn(Optional.*of*(bookItem));  
  
 ChangeBookStatusResult result = changeBookStatusService.process(input);  
  
 *assertEquals*(BookStatus.*DAMAGED*, result.getBookStatus());  
  
 ArgumentCaptor<BookItem> bookItemCaptor = ArgumentCaptor.*forClass*(BookItem.class);  
 *verify*(bookItemRepository, *times*(1)).save(bookItemCaptor.capture());  
  
 BookItem savedBookItem = bookItemCaptor.getValue();  
 *assertEquals*(BookStatus.*DAMAGED*, savedBookItem.getBookStatus());  
 }  
}

**Структура на проекта**

Проектната структура, която е използвана, е разделяне на слоевете на пакети, вариация на Domain-Driven Design. Този стил на архитектура организира приложението в отделни пакети, всеки със специфична отговорност, за да се постигне разделение на отговорностите и подобряване на поддържаемостта.

Разбивката изглежда по следния начин:

* API Модул:
  + Цел: Определя интерфейсите за услугите, обектите за пренос на данни (DTO) за входни и изходни данни, exceptions и други определения на контракти.
  + Компоненти: Интерфейси на услугите, DTO, персонализирани изключения.
* Core модул:
  + Цел: Съдържа основната бизнес логика и реализациите на интерфейсите, дефинирани в модула API.
  + Компоненти: Реализации на услугите, бизнес логика.
* Модул Persistence (съхранение):
  + Цел: Управлява съхранението на данни и взаимодействията с базата данни. Съдържа ентитита (които се мапват към таблиците в базата данни) и репозиторите (за CRUD операции и заявки към базата данни).
  + Компоненти: JPA ентитети, интерфейси на репозиториите, конфигурация на базата данни.
* Модул Front-end:
  + Цел: Управлява потребителския интерфейс и взаимодействията с потребителите. Този модул включва частта с контролерите, които обработват потребителския вход и делегират обработката към основните услуги.
  + Компоненти: UI компоненти, контролери, FXML файлове.

Ползи от тази структура:

* Разделение на отговорностите: Всеки пакет има специфична отговорност, което прави кода по-лесен за управление и разбиране.
* Повторна употреба: Пакетите могат да се използват впоследствие като модули повторно в различни проекти, ако са проектирани правилно.
* Тестируемост: Изолираните пакети правят по-лесно тестването на единици и интеграционното тестване.
* Поддържаемост: Промените в един пакет (например FE) е по-малко вероятно да повлияят на други модули (например, съхранение).

Структура на проекта:

A screenshot of a computer

Description automatically generated

**Проектиране на системата**

* **Api/base**

Тези класове и интерфейси представляват основни компоненти от системата, които са част от архитектурния дизайн на приложението. Те дефинират основни концепции и договори, които ще бъдат имплементирани и използвани в различни модули на системата.

public interface Processor<R extends ProcessorResult, I extends ProcessorInput> {  
 R process(I input);  
}

**Processor** е параметризиран интерфейс, който дефинира основния метод **process**. Този интерфейс приема два типа параметри: **R** и **I**, които трябва да разширяват **ProcessorResult** и **ProcessorInput** съответно.

**Цел:**

* **Обработка на данни**: Интерфейсът **Processor** дефинира основния контракт за обработка на входни данни и връщане на резултат.
* **Гъвкавост**: Параметризацията позволява създаването на различни имплементации за различни типове входни данни и резултати.

ProcessorInput е маркерен интерфейс, който служи като базов тип за всички входни данни, които ще бъдат обработвани от процесорите.

Цел:

Унифициране на входни данни: Дефинира общ тип за всички класове, които ще се използват като входни данни за процесорите.

Разширяемост: Позволява лесно добавяне на нови типове входни данни чрез имплементиране на този интерфейс.

ProcessorResult е маркерен интерфейс, който служи като базов тип за всички резултати от обработката, които ще бъдат връщани от процесорите.

Цел:

Унифициране на резултати: Дефинира общ тип за всички класове, които ще се използват като резултати от обработката.

Разширяемост: Позволява лесно добавяне на нови типове резултати чрез имплементиране на този интерфейс.

**Обща концепция:**

**Архитектурна роля:**

Тези интерфейси са част от архитектурния слой на системата, който дефинира основни договори и концепции. Те осигуряват:

* **Чистота на кода**: Разделят бизнес логиката от конкретните имплементации на процесорите.
* **Гъвкавост**: Позволяват лесно разширяване на системата чрез добавяне на нови типове входни данни и резултати, без да се променят съществуващите имплементации.
* **Повторна употреба**: Общите интерфейси могат да се използват в различни модули и контексти на приложението.

**Пример за интеграция в проектната структура:**

1. **API модул**:
   * Дефиниране на интерфейсите **Processor**, **ProcessorInput** и **ProcessorResult**.
2. **Core модул**:
   * Имплементиране на конкретни процесори, които реализират бизнес логиката чрез имплементиране на интерфейса **Processor**.
3. **Persistence модул**:
   * Създаване на входни данни и резултати, които могат да бъдат свързани с базата данни, ако е необходимо.
4. **UI модул**:
   * Използване на процесорите за обработка на потребителски заявки и показване на резултатите в потребителския интерфейс.

Тази структура осигурява ясно разграничение на отговорностите и улеснява поддържането и разширяването на системата.

Описание на operations пакета

Пакетът operations съдържа различни контракти и класове, които имплементират интерфейса Processor, както и конкретни реализации на ProcessorInput и ProcessorResult. Този пакет е сърцевината на бизнес логиката в системата и дефинира конкретни операции, които могат да бъдат извършвани.

**Архитектурна роля**

Пакетът **operations** дефинира конкретните операции, които системата може да извършва. Той служи като мост между основните интерфейси (дефинирани в **api** пакета) и реалните бизнес логики, които трябва да бъдат изпълнени.

**Цели и ползи:**

* **Капсулация на бизнес логика**: Чрез дефиниране на конкретни процесори, входове и резултати, пакетът капсулира бизнес логиката, правейки я лесно разширяема и поддържана.
* **Повторна употреба**: Интерфейсите и класовете в този пакет могат да бъдат лесно използвани в различни контексти и модули на приложението.
* **Разделяне на отговорности**: Всеки процесор е отговорен за конкретна операция, което улеснява поддръжката и развитието на системата.

**Пример за използване:**

Когато трябва да регистрирате нова книга, може да създадете **BookRegistrationInput**, да го подадете на **BookRegistrationProcessor** и да получите резултат в **BookRegistrationResult**:

Controller:

RegisterBookInput input = RegisterBookInput.*builder*()  
 .title(titleForm.getText())  
 .isbn(isbnForm.getText())  
 .author(authorForm.getText())  
 .bookGenre(genreForm.getValue())  
 .build();

…

…

…

RegisterBookResult result = registerBook.process(input);

Service:

return RegisterBookResult.*builder*()  
 .bookId(String.*valueOf*(book.getId()))  
 .build();

Енумерации:

public enum UserAuthority {  
 *ADMINISTRATOR*,  
 *OPERATOR*}

public enum BookStatus {  
 *AVAILABLE*,  
 *RENTED\_OUT*,  
 *ARCHIVED*,  
 *DAMAGED*,  
 *IN\_READING\_ROOM*}

public enum BookGenre {  
 *FICTION*,  
 *CYBERPUNK*,  
 *CHILDREN*,  
 *SCI\_FI*,  
 *HISTORICAL*}

Тези класове представляват изброими типове (enums), които дефинират различни стойности, използвани в системата. Те осигуряват структуриран начин за представяне на фиксирани набори от константи, което улеснява работата с различни типове данни.

**UserAuthority** дефинира различни нива на достъп за потребителите в системата.

**Цел:**

* **Контрол на достъпа**: Определя ролите на потребителите в системата, като осигурява подходящи права за всяка роля.
* **Улеснена поддръжка**: Чрез използването на enum, добавянето на нови роли или промяната на съществуващи е по-лесно и по-структурирано.

**BookStatus** дефинира различни състояния, в които може да се намира една книга.

**Цел:**

* **Управление на книги**: Позволява лесно проследяване и управление на статуса на книгите в библиотеката.
* **Информативност**: Улеснява потребителите и системата при разпознаване на текущото състояние на книгите.

BookGenre дефинира различни жанрове на книгите.

Цел:

Категоризация на книги: Помага за организиране и класифициране на книгите според техния жанр.

Търсене и филтриране: Улеснява потребителите при търсене и филтриране на книги по жанр.

Тези enum класове играят важна роля в дефинирането на бизнес логиката и предоставянето на структурирани и предефинирани стойности, които улесняват управлението на потребителите и книгите в системата.

**Цели и ползи:**

* **Яснота и четимост**: Използването на enum класове прави кода по-четим и лесен за разбиране.
* **Поддържане на инварианти**: Enum класовете гарантират, че стойностите са ограничени до предварително дефинираните опции, което помага за поддържане на инварианти в системата.
* **Лесна разширяемост**: Добавянето на нови стойности към enum класовете е лесно и безопасно, без риск от нарушаване на съществуващата логика.

**Core пакет:**

Пакетът **core** съдържа имплементациите на процесорите, които са отговорни за различните операции в системата. Всеки процесор приема специфичен входен клас, имплементиращ **ProcessorInput**, и връща специфичен резултатен клас, имплементиращ **ProcessorResult**. Тези класове са основната част от бизнес логиката на приложението, осигурявайки изпълнението на различни функционалности в системата.

Примери за класове от core пакета:

@RequiredArgsConstructor  
@Service  
public class AddBookItemsService implements AddBookItems {  
  
 private final BookItemRepository bookItemRepository;  
 private final BookRepository bookRepository;  
  
 @Override  
 @Transactional  
 public AddBookItemsResult process(AddBookItemsInput input) {  
 Optional<Book> book = bookRepository.findBookByIsbn(input.getIsbn());  
  
 if (book.isEmpty()) {  
 return AddBookItemsResult.*builder*()  
 .isSuccessful(false)  
 .message("Book with ISBN " + input.getIsbn() + " not found.")  
 .build();  
 }  
  
 for (int i = 0; i < input.getQuantity(); i++) {  
 bookItemRepository.save(BookItem.*builder*()  
 .book(book.get())  
 .bookStatus(BookStatus.*AVAILABLE*)  
 .build());  
 }  
  
 return AddBookItemsResult.*builder*()  
 .isSuccessful(true)  
 .message(input.getQuantity() + " items added for book with ISBN " + input.getIsbn())  
 .build();  
 }  
}

**AddBookItemsService** е сървис клас, който имплементира интерфейса **AddBookItems**. Този клас отговаря за добавянето на нови екземпляри от книги в системата. С използването на анотациите **@Service** и **@RequiredArgsConstructor**, той се регистрира като Spring Bean и автоматично инжектира своите зависимости.

**Компоненти и функционалности**

* **Зависимости**:
  + **BookItemRepository**: Репозитори за управление на екземплярите на книги в базата данни.
  + **BookRepository**: Репозиторий за управление на книгите в базата данни.
* **Методи**:
  + **process(AddBookItemsInput input)**: Основният метод, който обработва входните данни и добавя нови екземпляри на книги.

**Логика на метода process**

1. **Търсене на книга по ISBN**:
   * Използва **BookRepository** за намиране на книга с дадения ISBN.
   * Ако книгата не е намерена, връща резултат с неуспешно съобщение.
2. **Добавяне на екземпляри**:
   * Ако книгата е намерена, създава и запазва нови екземпляри на книгата в базата данни с помощта на **BookItemRepository**.
   * Всички нови екземпляри се маркират със статус **AVAILABLE**.
3. **Връщане на резултат**:
   * След успешното добавяне на екземплярите, връща резултат с успешно съобщение и броя на добавените екземпляри.

package bg.tuvarna.core.service;  
  
import bg.tuvarna.api.operations.operator.createreader.CreateReader;  
import bg.tuvarna.api.operations.operator.createreader.CreateReaderInput;  
import bg.tuvarna.api.operations.operator.createreader.CreateReaderResult;  
import bg.tuvarna.persistence.entity.Reader;  
import bg.tuvarna.persistence.repository.ReaderRepository;  
import lombok.RequiredArgsConstructor;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.stereotype.Service;  
  
import java.util.ArrayList;  
  
@Service  
@RequiredArgsConstructor  
@Slf4j  
public class CreateReaderService implements CreateReader {  
 private final ReaderRepository readerRepository;  
  
 @Override  
 public CreateReaderResult process(CreateReaderInput input) {  
 *log*.info("Processing CreateReaderInput: {}", input);  
  
 Reader newReader = Reader.*builder*()  
 .email(input.getEmail())  
 .firstName(input.getFirstName())  
 .lastName(input.getLastName())  
 .email(input.getEmail())  
 .phoneNumber(input.getPhoneNumber())  
 .readerRating(3)  
 .bookItemList(new ArrayList<>())  
 .build();  
  
 *log*.info("Saving new Reader: {}", newReader);  
 readerRepository.save(newReader);  
 *log*.info("New Reader saved successfully.");  
  
 return CreateReaderResult.*builder*()  
 .email(newReader.getEmail())  
 .firstName(newReader.getFirstName())  
 .lastName(newReader.getLastName())  
 .email(newReader.getEmail())  
 .phoneNumber(newReader.getPhoneNumber())  
 .readerRating(newReader.getReaderRating())  
 .build();  
 }  
}

**CreateReaderService** е сървис клас, който имплементира интерфейса **CreateReader**. Този клас отговаря за създаването на нови читатели в системата. С помощта на анотациите **@Service**, **@RequiredArgsConstructor** и **@Slf4j**, класът се регистрира като Spring Bean, автоматично инжектира своите зависимости и добавя логиране.

**Компоненти и функционалности**

* **Зависимости**:
  + **ReaderRepository**: Репозиторий за управление на читателите в базата данни.
* **Методи**:
  + **process(CreateReaderInput input)**: Основният метод, който обработва входните данни и създава нов читател.

**Логика на метода process**

1. **Логиране на входните данни**:
   * Използва **log.info** за да логира входните данни, които се обработват.
2. **Създаване на нов читател**:
   * Създава нов обект **Reader** с данните, предоставени в **CreateReaderInput**.
   * Задава първоначална оценка на читателя **readerRating** със стойност 3 и празен списък с книги **bookItemList**.
3. **Запазване на новия читател**:
   * Използва **readerRepository** за да запази новия читател в базата данни.
   * Логира успешното запазване на читателя.
4. **Връщане на резултат**:
   * Връща обект **CreateReaderResult** с данните на новосъздадения читател.

package bg.tuvarna.core.service;  
  
import bg.tuvarna.api.BookStatus;  
import bg.tuvarna.api.exception.BookNotFoundException;  
import bg.tuvarna.api.exception.UserNotFoundException;  
import bg.tuvarna.api.operations.operator.lendbookitem.LendBookItem;  
import bg.tuvarna.api.operations.operator.lendbookitem.LendBookItemInput;  
import bg.tuvarna.api.operations.operator.lendbookitem.LendBookItemResult;  
import bg.tuvarna.persistence.entity.Book;  
import bg.tuvarna.persistence.entity.BookItem;  
import bg.tuvarna.persistence.entity.Reader;  
import bg.tuvarna.persistence.repository.BookItemRepository;  
import bg.tuvarna.persistence.repository.BookRepository;  
import bg.tuvarna.persistence.repository.ReaderRepository;  
import lombok.RequiredArgsConstructor;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;  
  
import java.util.Optional;  
  
@Service  
@RequiredArgsConstructor  
@Slf4j  
public class LendBookItemService implements LendBookItem {  
  
 private final ReaderRepository readerRepository;  
 private final BookRepository bookRepository;  
 private final BookItemRepository bookItemRepository;  
  
 @Override  
 @Transactional  
 public LendBookItemResult process(LendBookItemInput input) {  
 *log*.info("Processing lend book item for reader email: {} and book isbn: {}", input.getReaderEmail(), input.getIsbn());  
  
 Reader reader = readerRepository.findByEmail(input.getReaderEmail())  
 .orElseThrow(() -> {  
 *log*.warn("Reader with email {} not found.", input.getReaderEmail());  
 return new UserNotFoundException("User with email not found");  
 });  
  
 Book book = bookRepository.findBookByIsbn(input.getIsbn()).get();  
 Optional<BookItem> bookItemToLend = bookItemRepository.findAllByBook(book)  
 .stream()  
 .filter(bookItem -> bookItem.getBookStatus() == BookStatus.*AVAILABLE*)  
 .findFirst();  
  
 if (bookItemToLend.isEmpty()) {  
 throw new BookNotFoundException("No available book items found for this book.");  
 }  
  
 BookItem availableBookItem = bookItemToLend.get();  
  
 reader.getBookItemList().add(availableBookItem);  
 readerRepository.save(reader);  
  
 availableBookItem.setBookStatus(input.getBookStatus());  
 availableBookItem.setReader(reader);  
 bookItemRepository.save(availableBookItem);  
  
 *log*.info("Book item with ID {} successfully lent to reader with email {}", availableBookItem.getId(), input.getReaderEmail());  
  
 return LendBookItemResult.*builder*()  
 .bookStatus(availableBookItem.getBookStatus())  
 .build();  
 }  
}

**LendBookItemService** е сървис клас, който имплементира интерфейса **LendBookItem**. Този клас отговаря за управлението на процеса по заемане на книга от читател. С помощта на анотациите **@Service**, **@RequiredArgsConstructor** и **@Slf4j**, класът се регистрира като Spring Bean, автоматично инжектира своите зависимости и добавя логиране.

**Компоненти и функционалности**

* **Зависимости**:
  + **ReaderRepository**: Репозиторий за управление на читателите в базата данни.
  + **BookRepository**: Репозиторий за управление на книгите в базата данни.
  + **BookItemRepository**: Репозиторий за управление на екземплярите на книги в базата данни.
* **Методи**:
  + **process(LendBookItemInput input)**: Основният метод, който обработва входните данни и извършва операцията по заемане на книга.

**Логика на метода process**

1. **Логиране на входните данни**:
   * Използва **log.info** за да логира входните данни, които се обработват.
2. **Намиране на читателя**:
   * Търси читател по предоставения имейл чрез **ReaderRepository**.
   * Ако читателят не е намерен, логира предупреждение и хвърля **UserNotFoundException**.
3. **Намиране на книгата**:
   * Търси книга по предоставения ISBN чрез **BookRepository**.
   * Ако книгата не е намерена, хвърля **BookNotFoundException**.
4. **Намиране на наличен екземпляр на книгата**:
   * Намира всички екземпляри на книгата чрез **BookItemRepository**.
   * Филтрира екземплярите, които са със статус **AVAILABLE**.
   * Ако няма налични екземпляри, хвърля **BookNotFoundException**.
5. **Обновяване на статуса на екземпляра**:
   * Добавя екземпляра към списъка с книги на читателя.
   * Обновява статуса на екземпляра на книгата и го асоциира с читателя.
   * Запазва промените в базата данни чрез **ReaderRepository** и **BookItemRepository**.
6. **Връщане на резултат**:
   * Връща обект **LendBookItemResult** със статуса на книгата.

package bg.tuvarna.core.service;  
  
import bg.tuvarna.api.exception.IncorrectInputException;  
import bg.tuvarna.api.exception.UserNotFoundException;  
import bg.tuvarna.api.operations.user.login.UserLogin;  
import bg.tuvarna.api.operations.user.login.UserLoginInput;  
import bg.tuvarna.api.operations.user.login.UserLoginResult;  
import bg.tuvarna.core.util.ActiveUser;  
import bg.tuvarna.persistence.repository.UserRepository;  
import lombok.RequiredArgsConstructor;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.stereotype.Service;  
  
import java.util.Optional;  
  
@Service  
@RequiredArgsConstructor  
@Slf4j  
public class UserLoginService implements UserLogin {  
 private final UserRepository userRepository;  
 private final ActiveUser activeUser;  
  
 @Override  
 public UserLoginResult process(UserLoginInput input) {  
 *log*.info("Processing UserLoginInput: {}", input);  
  
 if (input.getEmail().isEmpty() || input.getPassword().isEmpty()) {  
 *log*.warn("Email or password is empty.");  
 throw new IncorrectInputException("Email or password cannot be empty");  
 }  
  
 var user = userRepository.findByEmailAndPassword(input.getEmail(), input.getPassword())  
 .orElseThrow(() -> {  
 *log*.warn("Incorrect username or password");  
 return new UserNotFoundException("Incorrect username or password");  
 });  
  
 *log*.info("User authenticated successfully: {}", user);  
  
 activeUser.setUserId(Optional.*ofNullable*(user.getId()));  
 activeUser.setUserAuthority(Optional.*ofNullable*(user.getUserAuthority()));  
  
 *log*.info("Setting active user ID: {}", activeUser.getUserId().orElse(null));  
  
 return UserLoginResult.*builder*()  
 .userId(activeUser.getUserId().get())  
 .authority(activeUser.getUserAuthority().get())  
 .build();  
 }  
}

**UserLoginService** е сървис клас, който имплементира интерфейса **UserLogin**. Този клас отговаря за процеса на аутентикация на потребители в системата. С помощта на анотациите **@Service**, **@RequiredArgsConstructor** и **@Slf4j**, класът се регистрира като Spring Bean, автоматично инжектира своите зависимости и добавя логиране.

**Компоненти и функционалности**

* **Зависимости**:
  + **UserRepository**: Репозиторий за управление на потребителите в базата данни.
  + **ActiveUser**: Обект за управление на текущо активния потребител в сесията.
* **Методи**:
  + **process(UserLoginInput input)**: Основният метод, който обработва входните данни и извършва аутентикация на потребителя.

**Логика на метода process**

1. **Логиране на входните данни**:
   * Използва **log.info** за да логира входните данни, които се обработват.
2. **Проверка на входните данни**:
   * Проверява дали полетата за имейл и парола не са празни.
   * Ако някое от полетата е празно, логира предупреждение и хвърля **IncorrectInputException**.
3. **Аутентикация на потребителя**:
   * Търси потребител по имейл и парола чрез **UserRepository**.
   * Ако потребителят не е намерен, логира предупреждение и хвърля **UserNotFoundException**.
4. **Настройка на активния потребител**:
   * След успешна аутентикация, настройва текущо активния потребител чрез **ActiveUser**.
   * Логира успешното настройване на активния потребител.
5. **Връщане на резултат**:
   * Връща обект **UserLoginResult** с данните на аутентикация, включително идентификатор и роля на потребителя.

Има единствен клас, част от core, който не е имплементация на сервиз и това е ActiveUser класът.

@Component  
@Getter  
@Setter  
public class ActiveUser {  
 private Optional<UUID> userId;  
 private Optional<UserAuthority> userAuthority;  
}

**ActiveUser** е компонент клас, който се използва за управление на текущо активния потребител в системата. Този клас съдържа информация за идентификатора и ролята на активния потребител. С помощта на анотациите **@Component**, **@Getter** и **@Setter**, класът се регистрира като Spring Bean и автоматично генерира гетъри и сетъри за своите полета.

**Компоненти и функционалности**

* **Аннотации**:
  + **@Component**: Декларира класа като Spring Bean, което позволява той да бъде инжектиран в други компоненти на приложението.
  + **@Getter** и **@Setter**: Генерират автоматично гетъри и сетъри за полетата на класа.
* **Полета**:
  + **userId**: Съдържа идентификатора на активния потребител, представен като **Optional<UUID>**.
  + **userAuthority**: Съдържа ролята на активния потребител, представена като **Optional<UserAuthority>**.

**Описание на полетата**

1. **userId**:
   * **Тип**: **Optional<UUID>**
   * **Описание**: Съдържа уникалния идентификатор на текущо активния потребител. Използването на **Optional** позволява да се представи възможността, че може да няма активен потребител в даден момент.
2. **userAuthority**:
   * **Тип**: **Optional<UserAuthority>**
   * **Описание**: Съдържа ролята (авторитета) на текущо активния потребител. Използването на **Optional** позволява да се представи възможността, че може да няма зададена роля за активния потребител.

Този клас е лесен за интеграция и употреба в други компоненти на приложението чрез Spring Dependency Injection.

**Описание на класовете в пакета bg.tuvarna.frontend**

Тези класове представляват основни компоненти на фронтенд частта на приложението, което използва JavaFX за потребителския интерфейс и Spring Boot за бекенд логиката. Те осигуряват интеграция между JavaFX и Spring, позволявайки на приложението да използва предимствата на двете технологии.

public class AppContext {  
 private static AppContext *appContext* = null;  
  
 @Getter  
 @Setter  
 private ConfigurableApplicationContext context;  
  
 public AppContext() {  
 }  
  
 public static AppContext getInstance() {  
 if (*appContext* == null) {  
 *appContext* = new AppContext();  
 }  
 return *appContext*;  
 }  
}

**Описание:**

**AppContext** е сингълтон клас, който съдържа статична инстанция на Spring **ConfigurableApplicationContext**. Той осигурява централизирано управление на контекста на приложението.

**Компоненти:**

* **Полетата**:
  + **context**: Инстанция на **ConfigurableApplicationContext**, която представлява текущия Spring контекст.
* **Методи**:
  + **getInstance()**: Метод за получаване на единствената инстанция на **AppContext**.
  + Гетъри и сетъри за **context**, генерирани чрез Lombok анотации.

**Цел:**

* Управление на Spring контекста в JavaFX приложението, позволявайки интеграция между двата фреймуърка.

@Slf4j  
public class JavaFxApplication extends javafx.application.*Application* {  
 private AppContext context;  
 private *Parent* rootNode;  
  
 @Override  
 public void init() throws Exception {  
 AppContext.*getInstance*().setContext(SpringApplication.*run*(bg.tuvarna.frontend.SpringApplication.class));  
 FXMLLoader fxmlLoader = new FXMLLoader(JavaFxApplication.class.getResource("/javafxml/main-page.fxml"));  
 fxmlLoader.setControllerFactory(AppContext.*getInstance*().getContext()::getBean);  
 rootNode = fxmlLoader.load();  
 }  
  
 @Override  
 public void stop() {  
 *log*.info("Ended application");  
 Platform.*exit*();  
 }  
  
 @Override  
 public void start(Stage stage) throws IOException {  
 stage.setScene(new Scene(rootNode));  
 stage.show();  
 *log*.info("Started application");  
 }  
  
 public static void main(String[] args) {  
 *launch*();  
 }  
}

**Описание:**

**JavaFxApplication** е основният клас за JavaFX приложението, което се стартира като JavaFX приложение и интегрира Spring контекста.

**Компоненти:**

* **Методи**:
  + **init()**: Инициализира Spring контекста и зарежда основния FXML файл за потребителския интерфейс.
  + **stop()**: Логира и прекратява JavaFX платформата.
  + **start(Stage stage)**: Настройва сцената и показва основния прозорец на приложението.
  + **main(String[] args)**: Главен метод за стартиране на приложението.

**Цел:**

* Стартиране на JavaFX приложението и интегриране на Spring контекста за управление на зависимостите и бизнес логиката.

@SpringBootApplication  
@ComponentScan(basePackages = "bg.tuvarna")  
@EntityScan(basePackages = "bg.tuvarna.persistence.entity")  
@EnableJpaRepositories(basePackages = "bg.tuvarna.persistence.repository")  
public class SpringApplication {  
 public static void main(String[] args) {  
 *Application*.*launch*(JavaFxApplication.class, args);  
 }  
}

**SpringApplication** е конфигурационен клас за Spring Boot приложението, който също така стартира JavaFX приложението.

**Аннотации:**

* **@SpringBootApplication**: Декларира този клас като конфигурационен клас за Spring Boot.
* **@ComponentScan**: Определя базовия пакет за сканиране на Spring компоненти.
* **@EntityScan**: Определя базовия пакет за сканиране на JPA ентитети.
* **@EnableJpaRepositories**: Активира JPA репозиториите и определя базовия пакет за тяхното сканиране.

**Цел:**

* Конфигуриране и стартиране на Spring Boot приложението, както и интегрирането му с JavaFX чрез стартиране на **JavaFxApplication**.

Тези класове осигуряват необходимата инфраструктура за интеграция на JavaFX и Spring Boot в едно приложение. **AppContext** управлява единния Spring контекст, **JavaFxApplication** се грижи за стартирането на JavaFX приложението и интегрирането му със Spring, а **SpringApplication** конфигурира и стартира Spring Boot приложението. Тази архитектура позволява на разработчиците да използват предимствата на двата фреймуърка, осигурявайки мощна и гъвкава платформа за разработка на съвременни десктоп приложения.

@Component  
public class SpringFXMLLoader {  
 private final ApplicationContext context;  
  
 public SpringFXMLLoader(ApplicationContext context) {  
 this.context = context;  
 }  
  
 public Object load(String fxmlPath) throws IOException {  
 try (*InputStream* fxmlStream = getClass().getResourceAsStream(fxmlPath)) {  
 FXMLLoader loader = new FXMLLoader();  
 loader.setControllerFactory(context::getBean);  
 return loader.load(fxmlStream);  
 }  
 }  
}

}

**Описание:**

**SpringFXMLLoader** е компонент клас, който се грижи за зареждането на FXML файлове, като използва Spring контекста за управление на зависимостите. Той осигурява интеграция между JavaFX и Spring, позволявайки на контролерите в FXML файловете да бъдат управлявани от Spring.

**Компоненти:**

* **Полетата**:
  + **context**: Обект от тип **ApplicationContext**, който представлява текущия Spring контекст.
* **Методи**:
  + **load(String fxmlPath)**: Метод за зареждане на FXML файл, като използва Spring контекста за създаване на контролерите.

**Цел:**

* Зареждане на FXML файлове и свързване на контролерите със Spring контекста, осигурявайки управление на зависимостите в JavaFX приложението.

@Component  
@RequiredArgsConstructor  
public class SceneChanger {  
  
 private final SpringFXMLLoader springFXMLLoader;  
  
 public void changeScene(Stage stage, String fxmlFilePath) {  
 try {  
 *Parent* root = (*Parent*) springFXMLLoader.load(fxmlFilePath);  
 Scene scene = new Scene(root);  
 stage.setScene(scene);  
 stage.show();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

**Описание:**

**SceneChanger** е компонент клас, който предоставя метод за смяна на сцените в JavaFX приложението. Той използва **SpringFXMLLoader** за зареждане на FXML файловете и настройка на нова сцена.

**Компоненти:**

* **Полетата**:
  + **springFXMLLoader**: Обект от тип **SpringFXMLLoader**, който се използва за зареждане на FXML файлове.
* **Методи**:
  + **changeScene(Stage stage, String fxmlFilePath)**: Метод за смяна на сцената, като приема текущия **Stage** и пътя до FXML файла.

**Цел:**

* Осигуряване на лесен начин за смяна на сцените в JavaFX приложението, интегрирано със Spring контекста.

**SceneChanger** и **SpringFXMLLoader** са помощни класове, които осигуряват интеграция между JavaFX и Spring в едно приложение. **SceneChanger** предоставя метод за лесна смяна на сцените, използвайки **SpringFXMLLoader** за зареждане на FXML файловете и настройка на нова сцена. **SpringFXMLLoader** се грижи за зареждането на FXML файловете и свързването на контролерите със Spring контекста, осигурявайки управление на зависимостите и интеграция на двата фреймуърка. Тази архитектура позволява създаването на модерни и лесно поддържани JavaFX приложения, които използват предимствата на Spring.

**Слой с JavaFX контролерите**

Слоят с JavaFX контролерите в приложението е отговорен за обработката на потребителските взаимодействия и свързването на потребителския интерфейс (UI) с бизнес логиката. Този слой съдържа класове, които имплементират контролери за различни FXML изгледи. Контролерите взаимодействат с UI компонентите, обработват събития и извикват съответните услуги от бизнес логиката.

**Основни компоненти и функционалности на слоя с JavaFX контролерите**

**1. Контролери**

Контролерите са класове, които се асоциират с конкретни FXML файлове. Те съдържат логиката за управление на UI компонентите и обработка на събитията, генерирани от потребителите.

**2. Свързване с FXML**

Контролерите са свързани с FXML файловете чрез **fx:controller** атрибут в самите FXML файлове. Това позволява на FXMLLoader да знае кой контролер да използва за дадения UI.

**3. Инжектиране на зависимости**

С помощта на Spring и аннотацията **@FXML**, контролерите могат да използват Dependency Injection (DI), за да инжектират услуги и други компоненти от Spring контекста.

@Component  
@Slf4j  
public class AdminController {  
  
 @FXML  
 private Button addBookItemsButton;  
  
 @FXML  
 private TableColumn<?, ?> bookAuthorColumn;  
 @FXML  
  
 private TableColumn<?, ?> bookQuantityColumn;  
 @FXML  
 private TableColumn<?, ?> bookIsbnColumn;  
  
 @FXML  
 private TableView bookTable;  
  
 @FXML  
 private TableColumn<?, ?> bookTitleColumn;  
  
 @FXML  
 private Button createOperatorButton;  
  
 @FXML  
 private Button logoutButton;  
  
 @FXML  
 private TableColumn<?, ?> operatorEmailColumn;  
 @FXML  
 private TableColumn<?, ?> operatorIdColumn;  
  
 @FXML  
 private TableView<OperatorDTO> operatorTable;  
  
 @FXML  
 private Button registerBookButton;  
  
 @FXML  
 private Button removeArchivedBookItemsButton;  
  
 @FXML  
 private Button removeDamagedBookItemsButton;  
  
 @FXML  
 private Button removeOperatorButton;  
  
 @Autowired  
 private UserLogout userLogout;  
 @Autowired  
 private GetOperators getOperators;  
 @Autowired  
 private SceneChanger sceneChanger;  
 @Autowired  
 private RemoveOperator removeOperator;  
 @Autowired  
 private GetBooks getBooks;  
 @Autowired  
 private AddBookItems addBookItems;  
 @Autowired  
 private RemoveArchivedBookItems removeArchivedBookItems;  
 @Autowired  
 private RemoveDamagedBookItems removeDamagedBookItems;  
  
 @Value("${fxml.paths.loginForm}")  
 private String loginFormPath;  
 @Value("${fxml.paths.createOperatorForm}")  
 private String createOperatorFormPath;  
 @Value("${fxml.paths.registerBookForm}")  
 private String registerBookFormPath;  
  
 @FXML  
 void createOperator() {  
 sceneChanger.changeScene((Stage) logoutButton.getScene().getWindow(), createOperatorFormPath);  
 }  
  
 @FXML  
 void registerBook() {  
 sceneChanger.changeScene((Stage) registerBookButton.getScene().getWindow(), registerBookFormPath);  
 }  
  
 @FXML  
 void logout() {  
 try {  
 userLogout.process(new UserLogoutInput());  
  
 sceneChanger.changeScene((Stage) logoutButton.getScene().getWindow(), loginFormPath);  
 *log*.info("User logged out successfully");  
 } catch (Exception e) {  
 *log*.error(e.getMessage());  
// e.printStackTrace();  
 }  
 }  
  
 @FXML  
 public void initialize() {  
 initOperators();  
 initBooks();  
 }  
  
 @FXML  
 void removeOperator() {  
 // Get the selected item from the TableView  
 Optional.*ofNullable*(operatorTable.getSelectionModel().getSelectedItem()).ifPresentOrElse(  
 selectedOperator -> {  
 // Show a confirmation dialog  
 Alert confirmationAlert = new Alert(Alert.AlertType.*CONFIRMATION*);  
 confirmationAlert.setTitle("Confirmation");  
 confirmationAlert.setHeaderText(null);  
 confirmationAlert.setContentText("Are you sure you want to delete the operator " + selectedOperator.getEmail() + "?");  
  
 Optional<ButtonType> result = confirmationAlert.showAndWait();  
 if (result.isPresent() && result.get() == ButtonType.*OK*) {  
 // Create input for RemoveOperator process  
 RemoveOperatorInput input = RemoveOperatorInput.*builder*()  
 .operatorEmail(selectedOperator.getEmail())  
 .build();  
  
 // Process the removal  
 removeOperator.process(input);  
  
 // Optionally, you might want to remove the item from the table view  
 operatorTable.getItems().remove(selectedOperator);  
 }  
 },  
 () -> {  
 // Handle the case when no item is selected  
 Alert alert = new Alert(Alert.AlertType.*WARNING*);  
 alert.setTitle("Warning");  
 alert.setHeaderText(null);  
 alert.setContentText("Please select an operator to remove.");  
 alert.showAndWait();  
 }  
 );  
 }  
 @FXML  
 void removeArchivedBookItems(){  
 removeArchivedBookItems.process(RemoveArchivedBookItemsInput.*builder*().build());  
 }  
 @FXML  
 void removeDamagedBookItems(){  
 removeDamagedBookItems.process(RemoveDamagedBookItemsInput.*builder*().build());  
 }  
  
 @FXML  
 void addBookItems() {  
 // Get the selected book from the TableView  
 Optional<BookDTO> selectedBook = Optional.*ofNullable*((BookDTO) bookTable.getSelectionModel().getSelectedItem());  
  
 if (selectedBook.isPresent()) {  
 // Show an input dialog to get the quantity  
 TextInputDialog dialog = new TextInputDialog();  
 dialog.setTitle("Add Book Items");  
 dialog.setHeaderText("Add new items for the book: " + selectedBook.get().getTitle());  
 dialog.setContentText("Please enter the quantity:");  
  
 Optional<String> result = dialog.showAndWait();  
  
 result.ifPresent(quantityStr -> {  
 try {  
 int quantity = Integer.*parseInt*(quantityStr);  
  
 // Validate the quantity  
 if (quantity <= 0) {  
 Alert alert = new Alert(Alert.AlertType.*ERROR*);  
 alert.setTitle("Invalid Quantity");  
 alert.setHeaderText(null);  
 alert.setContentText("The quantity must be a positive integer.");  
 alert.showAndWait();  
 return;  
 }  
  
 // Create input for AddBookItems process  
 AddBookItemsInput input = AddBookItemsInput.*builder*()  
 .isbn(selectedBook.get().getIsbn())  
 .quantity(quantity)  
 .build();  
  
 // Process adding book items  
 addBookItems.process(input);  
  
 // Optionally, show a confirmation dialog  
 Alert confirmationAlert = new Alert(Alert.AlertType.*INFORMATION*);  
 confirmationAlert.setTitle("Success");  
 confirmationAlert.setHeaderText(null);  
 confirmationAlert.setContentText(quantity + " items have been added for the book: " + selectedBook.get().getTitle());  
 confirmationAlert.showAndWait();  
  
 // Refresh the book list (if necessary)  
 initBooks();  
  
 } catch (NumberFormatException e) {  
 Alert alert = new Alert(Alert.AlertType.*ERROR*);  
 alert.setTitle("Invalid Input");  
 alert.setHeaderText(null);  
 alert.setContentText("Please enter a valid number.");  
 alert.showAndWait();  
 }  
 });  
 } else {  
 // Handle the case when no book is selected  
 Alert alert = new Alert(Alert.AlertType.*WARNING*);  
 alert.setTitle("Warning");  
 alert.setHeaderText(null);  
 alert.setContentText("Please select a book to add items.");  
 alert.showAndWait();  
 }  
 }  
  
  
 private void initOperators() {  
  
 operatorEmailColumn.setCellValueFactory(new PropertyValueFactory<>("email"));  
 operatorIdColumn.setCellValueFactory(new PropertyValueFactory<>("id"));  
  
  
 ObservableList<OperatorDTO> operators = FXCollections.*observableArrayList*(  
 getOperators.process(new GetOperatorsInput()).getOperators()  
 );  
 operatorTable.setItems(operators);  
 }  
  
 private void initBooks() {  
 bookTitleColumn.setCellValueFactory(new PropertyValueFactory<>("title"));  
 bookAuthorColumn.setCellValueFactory(new PropertyValueFactory<>("author"));  
 bookQuantityColumn.setCellValueFactory(new PropertyValueFactory<>("quantity"));  
 bookIsbnColumn.setCellValueFactory(new PropertyValueFactory<>("isbn"));  
  
 ObservableList<BookDTO> bookDTOS = FXCollections.*observableArrayList*(  
 getBooks.process(new GetBooksInput()).getBookDTOList()  
 );  
 bookTable.setItems(bookDTOS);  
 }  
  
}

**Описание на AdminController**

**AdminController** е JavaFX контролер клас, който управлява административния интерфейс на приложението. Той предоставя функционалности за управление на книги и оператори, включително добавяне на нови книги, създаване на оператори, премахване на оператори и книги, и изход от системата. Класът е интегриран със Spring контекста за управление на зависимости и използва различни услуги за извършване на бизнес логика.

**Компоненти и функционалности**

**Аннотации**

* **@Component**: Декларира класа като Spring Bean, което позволява той да бъде инжектиран и управляван от Spring контекста.
* **@Slf4j**: Добавя логиране към класа чрез Lombok, осигурявайки лесен достъп до логиращия механизъм.

**UI Компоненти (FXML полета)**

* **Button**: UI компоненти за различни действия като добавяне на книги, създаване на оператори, изход и т.н.
* **TableView** и **TableColumn**: Компоненти за визуализация и управление на таблици с книги и оператори.

**Инжектирани услуги**

* **UserLogout**: Услуга за изход от системата.
* **GetOperators**: Услуга за получаване на списък с оператори.
* **SceneChanger**: Услуга за смяна на сцените в JavaFX приложението.
* **RemoveOperator**: Услуга за премахване на оператори.
* **GetBooks**: Услуга за получаване на списък с книги.
* **AddBookItems**: Услуга за добавяне на нови екземпляри от книги.
* **RemoveArchivedBookItems**: Услуга за премахване на архивирани книги.
* **RemoveDamagedBookItems**: Услуга за премахване на повредени книги.

**Конфигурационни стойности**

* **loginFormPath**: Път към формата за вход.
* **createOperatorFormPath**: Път към формата за създаване на оператор.
* **registerBookFormPath**: Път към формата за регистрация на книга.

@Component  
@Slf4j  
public class CreateOperatorController {  
  
 @FXML  
 private Button backButton;  
  
 @FXML  
 private PasswordField confirmPasswordForm;  
  
 @FXML  
 private Label confirmPasswordLabel;  
  
 @FXML  
 private Button createOperatorButton;  
  
 @FXML  
 private Text createOperatorLabel;  
  
 @FXML  
 private TextField emailForm;  
  
 @FXML  
 private Label emailLabel;  
  
 @FXML  
 private PasswordField passwordForm;  
  
 @FXML  
 private Label passwordLabel;  
  
 @FXML  
 private Label failedCreationLabel;  
  
 @Autowired  
 private SceneChanger sceneChanger;  
 @Value("${fxml.paths.adminForm}")  
 private String adminViewPath;  
 @Autowired  
 private CreateOperator createOperator;  
  
 @FXML  
 void goBack() {  
 sceneChanger.changeScene((Stage) backButton.getScene().getWindow(), adminViewPath);  
  
 }  
  
 @FXML  
 void register() {  
 CreateOperatorInput input = CreateOperatorInput.*builder*()  
 .email(emailForm.getText())  
 .password(passwordForm.getText())  
 .confirmPassword(confirmPasswordForm.getText())  
 .build();  
  
 try {  
 CreateOperatorResult result = createOperator.process(input);  
 *log*.info("Operator created successfully: {}", result);  
 } catch (Exception e) {  
 *log*.error("Error creating operator: {}", e.getMessage());  
  
 failedCreationLabel.setText("Failed to create operator: " + e.getMessage());  
 }  
 sceneChanger.changeScene((Stage) backButton.getScene().getWindow(), adminViewPath);  
 }  
  
}

**CreateOperatorController** е JavaFX контролер клас, който управлява формата за създаване на нов оператор в библиотечната система. Този клас предоставя функционалност за въвеждане на данни за нов оператор, проверка на валидността на данните и създаване на нов оператор чрез бизнес логиката, интегрирана със Spring.

**Компоненти и функционалности**

**Аннотации**

* **@Component**: Декларира класа като Spring Bean, което позволява той да бъде инжектиран и управляван от Spring контекста.
* **@Slf4j**: Добавя логиране към класа чрез Lombok, осигурявайки лесен достъп до логиращия механизъм.

**UI Компоненти (FXML полета)**

* **Button**: UI компоненти за действия като връщане назад и създаване на оператор.
* **PasswordField**: Полета за въвеждане на парола и потвърждение на парола.
* **Label**: Етикети за полета и съобщения за грешки.
* **TextField**: Поле за въвеждане на имейл.
* **Text**: Текст за заглавието на формата.

**Инжектирани услуги**

* **SceneChanger**: Услуга за смяна на сцените в JavaFX приложението.
* **CreateOperator**: Услуга за създаване на нов оператор.

**Конфигурационни стойности**

* **adminViewPath**: Път към формата за администрация, зададен чрез **@Value**.

package bg.tuvarna.frontend.controller;  
  
import bg.tuvarna.api.operations.operator.createreader.CreateReader;  
import bg.tuvarna.api.operations.operator.createreader.CreateReaderInput;  
import bg.tuvarna.api.operations.operator.createreader.CreateReaderResult;  
import bg.tuvarna.frontend.utils.SceneChanger;  
import javafx.event.ActionEvent;  
import javafx.fxml.FXML;  
import javafx.scene.control.Button;  
import javafx.scene.control.Label;  
import javafx.scene.control.TextField;  
import javafx.scene.text.Text;  
import javafx.stage.Stage;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
@Slf4j  
@Component  
public class CreateReaderController {  
  
 @FXML  
 private Button backButton;  
  
 @FXML  
 private Button createReaderButton;  
  
 @FXML  
 private Text createReaderLabel;  
  
 @FXML  
 private TextField emailField;  
  
 @FXML  
 private Label emailLabel;  
  
 @FXML  
 private TextField firstNameField;  
  
 @FXML  
 private Label firstNameLabel;  
  
 @FXML  
 private TextField lastNameField;  
  
 @FXML  
 private Label lastNameLabel;  
  
 @FXML  
 private TextField phoneNumberField;  
  
 @FXML  
 private Label phoneNumberLabel;  
  
 @FXML  
 private Label failedCreationLabel;  
  
 @Value("${fxml.paths.operatorForm}")  
 private String operatorViewPath;  
  
 @Autowired  
 private SceneChanger sceneChanger;  
  
 @Autowired  
 private CreateReader createReader;  
  
 @FXML  
 void createReader(ActionEvent event) {  
 CreateReaderInput input = CreateReaderInput.*builder*()  
 .firstName(firstNameField.getText())  
 .lastName(lastNameField.getText())  
 .phoneNumber(phoneNumberField.getText())  
 .email(emailField.getText())  
 .build();  
  
 try {  
 CreateReaderResult result = createReader.process(input);  
 *log*.info("Reader created successfully: {}", result);  
 } catch (Exception e) {  
 *log*.error("Error creating reader: {}", e.getMessage());  
  
 failedCreationLabel.setText("Failed to create reader: " + e.getMessage());  
 return;  
 }  
 sceneChanger.changeScene((Stage) backButton.getScene().getWindow(), operatorViewPath);  
 }  
  
 @FXML  
 void goBack(ActionEvent event) {  
 sceneChanger.changeScene((Stage) backButton.getScene().getWindow(), operatorViewPath);  
 }  
}

**CreateReaderController** е JavaFX контролер клас, който управлява формата за създаване на нов читател в библиотечната система. Този клас предоставя функционалност за въвеждане на данни за нов читател, проверка на валидността на данните и създаване на нов читател чрез бизнес логиката, интегрирана със Spring.

**Компоненти и функционалности**

**Аннотации**

* **@Component**: Декларира класа като Spring Bean, което позволява той да бъде инжектиран и управляван от Spring контекста.
* **@Slf4j**: Добавя логиране към класа чрез Lombok, осигурявайки лесен достъп до логиращия механизъм.

**UI Компоненти (FXML полета)**

* **Button**: UI компоненти за действия като връщане назад и създаване на читател.
* **TextField**: Полета за въвеждане на информация за новия читател (имейл, име, фамилия, телефонен номер).
* **Label**: Етикети за полетата и съобщения за грешки.
* **Text**: Текст за заглавието на формата.

**Инжектирани услуги**

* **SceneChanger**: Услуга за смяна на сцените в JavaFX приложението.
* **CreateReader**: Услуга за създаване на нов читател.

**Конфигурационни стойности**

* **operatorViewPath**: Път към формата за оператор, зададен чрез **@Value**.

**Методи**

**createReader**

Обработва данните от формата и създава нов читател. В случай на грешка, показва съобщение за неуспех.

package bg.tuvarna.frontend.controller;  
  
import bg.tuvarna.api.UserAuthority;  
import bg.tuvarna.api.operations.user.login.UserLoginInput;  
import bg.tuvarna.api.operations.user.login.UserLoginResult;  
import bg.tuvarna.core.service.UserLoginService;  
import bg.tuvarna.frontend.utils.SceneChanger;  
import javafx.fxml.FXML;  
import javafx.scene.control.Button;  
import javafx.scene.control.Label;  
import javafx.scene.control.PasswordField;  
import javafx.scene.control.TextField;  
import javafx.stage.Stage;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
@Component  
@Slf4j  
public class MainController {  
  
 @FXML  
 private TextField emailForm;  
  
 @FXML  
 private Label emailLabel;  
  
 @FXML  
 private Button loginButton;  
  
 @FXML  
 private Label mainTitle;  
  
 @FXML  
 private PasswordField passwordForm;  
  
 @FXML  
 private Label passwordLabel;  
  
 @FXML  
 private Label userNotFoundLabel;  
  
 @Autowired  
 private SceneChanger sceneChanger;  
 @Autowired  
 private UserLoginService userLoginService;  
  
 @Value("${fxml.paths.adminForm}")  
 private String adminFormPath;  
 @Value("${fxml.paths.operatorForm}")  
 private String operatorFormPath;  
  
 @FXML  
 void login() {  
  
 UserLoginInput userLoginInput = UserLoginInput.*builder*()  
 .email(emailForm.getText())  
 .password(passwordForm.getText())  
 .build();  
 try {  
 UserLoginResult result = userLoginService.process(userLoginInput);  
 *log*.info(adminFormPath);  
 if (result.getAuthority().equals(UserAuthority.*ADMINISTRATOR*)) {  
 sceneChanger.changeScene((Stage) mainTitle.getScene().getWindow(), adminFormPath);  
 }  
 if(result.getAuthority().equals(UserAuthority.*OPERATOR*)){  
 sceneChanger.changeScene((Stage) mainTitle.getScene().getWindow(), operatorFormPath);  
 }  
 *log*.info("User logged in successfully: {}", userLoginInput.getEmail());  
 } catch (Exception e) {  
 userNotFoundLabel.setText(e.getMessage());  
 *log*.error(e.getMessage());  
 }  
 }  
  
}

**MainController** е JavaFX контролер клас, който управлява главната форма за вход в системата. Този клас предоставя функционалност за въвеждане на имейл и парола, проверка на валидността на данните и логване на потребителя чрез бизнес логиката, интегрирана със Spring. В зависимост от ролята на потребителя, контролерът пренасочва към различни сцени.

**Компоненти и функционалности**

**Аннотации**

* **@Component**: Декларира класа като Spring Bean, което позволява той да бъде инжектиран и управляван от Spring контекста.
* **@Slf4j**: Добавя логиране към класа чрез Lombok, осигурявайки лесен достъп до логиращия механизъм.

**UI Компоненти (FXML полета)**

* **TextField**: Поле за въвеждане на имейл.
* **PasswordField**: Поле за въвеждане на парола.
* **Button**: Бутон за извършване на вход.
* **Label**: Етикети за полетата и съобщения за грешки.
* **Text**: Текст за заглавието на формата.

**Инжектирани услуги**

* **SceneChanger**: Услуга за смяна на сцените в JavaFX приложението.
* **UserLoginService**: Услуга за логване на потребители.

**Конфигурационни стойности**

* **adminFormPath**: Път към формата за администратори, зададен чрез **@Value**.
* **operatorFormPath**: Път към формата за оператори, зададен чрез **@Value**.
* package bg.tuvarna.frontend.controller;  
    
  import bg.tuvarna.api.BookStatus;  
  import bg.tuvarna.api.operations.operator.createreader.CreateReader;  
  import bg.tuvarna.api.operations.operator.getreaders.GetReaders;  
  import bg.tuvarna.api.operations.operator.getreaders.GetReadersInput;  
  import bg.tuvarna.api.operations.operator.lendbookitem.LendBookItem;  
  import bg.tuvarna.api.operations.operator.lendbookitem.LendBookItemInput;  
  import bg.tuvarna.api.operations.operator.notifyreader.NotifyReader;  
  import bg.tuvarna.api.operations.operator.notifyreader.NotifyReaderInput;  
  import bg.tuvarna.api.operations.operator.removereader.RemoveReader;  
  import bg.tuvarna.api.operations.operator.removereader.RemoveReaderInput;  
  import bg.tuvarna.api.operations.user.getbooks.GetBooks;  
  import bg.tuvarna.api.operations.user.getbooks.GetBooksInput;  
  import bg.tuvarna.api.operations.user.logout.UserLogout;  
  import bg.tuvarna.api.operations.user.logout.UserLogoutInput;  
  import bg.tuvarna.api.operations.util.BookDTO;  
  import bg.tuvarna.api.operations.util.ReaderDTO;  
  import bg.tuvarna.frontend.utils.SceneChanger;  
  import javafx.collections.FXCollections;  
  import javafx.collections.ObservableList;  
  import javafx.fxml.FXML;  
  import javafx.fxml.FXMLLoader;  
  import javafx.scene.*Parent*;  
  import javafx.scene.Scene;  
  import javafx.scene.control.\*;  
  import javafx.scene.control.cell.PropertyValueFactory;  
  import javafx.stage.Stage;  
  import lombok.extern.slf4j.Slf4j;  
  import org.springframework.beans.factory.annotation.Autowired;  
  import org.springframework.beans.factory.annotation.Value;  
  import org.springframework.stereotype.Component;  
    
  import java.io.IOException;  
  import java.util.Arrays;  
  import java.util.List;  
  import java.util.Optional;  
    
  @Component  
  @Slf4j  
  public class OperatorController {  
    
   @FXML  
   private TableColumn<?, ?> bookAuthorColumn;  
    
   @FXML  
   private TableColumn<?, ?> bookIsbnColumn;  
    
   @FXML  
   private TableColumn<?, ?> bookQuantityColumn;  
    
   @FXML  
   private TableColumn<?, ?> bookTitleColumn;  
    
   @FXML  
   private TableView<BookDTO> booksTableView;  
    
   @FXML  
   private Button createReaderButton;  
    
   @FXML  
   private Button deleteReaderButton;  
    
   @FXML  
   private Button logoutButton;  
    
   @FXML  
   private TableColumn<?, ?> readerEmailColumn;  
    
   @FXML  
   private TableColumn<?, ?> readerFirstNameColumn;  
    
   @FXML  
   private TableColumn<?, ?> readerLastNameColumn;  
    
   @FXML  
   private TableColumn<?, ?> readerPhoneNumberColumn;  
    
   @FXML  
   private TableColumn<?, ?> readerRatingColumn;  
    
   @FXML  
   private TableView<ReaderDTO> readersTableView;  
   @Value("${fxml.paths.createReaderForm}")  
   private String createReaderPath;  
   @Value("${fxml.paths.loginForm}")  
   private String loginFormPath;  
   @Value("${fxml.paths.returnBookForm}")  
   private String returnBookScenePath;  
   @Autowired  
   private SceneChanger sceneChanger;  
    
   @Autowired  
   private CreateReader createReader;  
   @Autowired  
   private RemoveReader removeReader;  
   @Autowired  
   private GetBooks getBooks;  
   @Autowired  
   private UserLogout userLogout;  
   @Autowired  
   private GetReaders getReaders;  
   @Autowired  
   private LendBookItem lendBookItem;  
   @Autowired  
   private NotifyReader notifyReader;  
   @Autowired  
   private ReturnBookController returnBookController;  
    
   @FXML  
   void initialize(){  
    
   initBooks();  
   initReaders();  
   }  
    
   @FXML  
   void createReader() {  
   sceneChanger.changeScene((Stage) createReaderButton.getScene().getWindow(), createReaderPath);  
   }  
    
   @FXML  
   void removeReader() {  
   ReaderDTO selectedReader = readersTableView.getSelectionModel().getSelectedItem();  
   if (selectedReader != null) {  
   try {  
   RemoveReaderInput input = RemoveReaderInput.*builder*()  
   .email(selectedReader.getEmail())  
   .build();  
   removeReader.process(input);  
   *log*.info("Reader with email {} deleted successfully", selectedReader.getEmail());  
    
   initReaders();  
   } catch (Exception e) {  
   *log*.error("Error deleting reader: {}", e.getMessage());  
   }  
   } else {  
   *log*.warn("No reader selected for deletion.");  
   }  
   }  
    
   private void initBooks() {  
   bookTitleColumn.setCellValueFactory(new PropertyValueFactory<>("title"));  
   bookAuthorColumn.setCellValueFactory(new PropertyValueFactory<>("author"));  
   bookQuantityColumn.setCellValueFactory(new PropertyValueFactory<>("quantity"));  
   bookIsbnColumn.setCellValueFactory(new PropertyValueFactory<>("isbn"));  
    
   ObservableList<BookDTO> bookDTOS = FXCollections.*observableArrayList*(  
   getBooks.process(new GetBooksInput()).getBookDTOList()  
   );  
   booksTableView.setItems(bookDTOS);  
   }  
    
   public void logout() {  
   try {  
   userLogout.process(new UserLogoutInput());  
    
   sceneChanger.changeScene((Stage) logoutButton.getScene().getWindow(), loginFormPath);  
   *log*.info("User logged out successfully");  
   } catch (Exception e) {  
   *log*.error(e.getMessage());  
   }  
   }  
    
   private void initReaders() {  
   readerFirstNameColumn.setCellValueFactory(new PropertyValueFactory<>("firstName"));  
   readerLastNameColumn.setCellValueFactory(new PropertyValueFactory<>("lastName"));  
   readerPhoneNumberColumn.setCellValueFactory(new PropertyValueFactory<>("phoneNumber"));  
   readerEmailColumn.setCellValueFactory(new PropertyValueFactory<>("email"));  
   readerRatingColumn.setCellValueFactory(new PropertyValueFactory<>("rating"));  
    
   ObservableList<ReaderDTO> readerDTOS = FXCollections.*observableArrayList*(  
   getReaders.process(new GetReadersInput()).getReaderDTOList()  
   );  
   readersTableView.setItems(readerDTOS);  
    
   readersTableView.setOnMouseClicked(click -> {  
   if (click.getClickCount() == 2) {  
   ReaderDTO selectedReader = readersTableView.getSelectionModel().getSelectedItem();  
   if (selectedReader != null) {  
   openReaderBookItemsView(selectedReader);  
   }  
   }  
   });  
   }  
   @FXML  
   void notifyReader(){  
   String readerEmail = readersTableView.getSelectionModel().getSelectedItem().getEmail();  
   notifyReader.process(NotifyReaderInput.*builder*()  
   .email(readerEmail)  
   .build());  
   }  
    
   @FXML  
   void lendBook(){  
   ReaderDTO selectedReader = readersTableView.getSelectionModel().getSelectedItem();  
   String bookIsbn = booksTableView.getSelectionModel().getSelectedItem().getIsbn();  
    
   if (selectedReader == null || bookIsbn == null) {  
   showAlert("Selection Error", "Please select a reader and a book.");  
   return;  
   }  
    
   List<String> choices = Arrays.*asList*("Reading Room", "Rent Out");  
    
   ChoiceDialog<String> dialog = new ChoiceDialog<>("Reading Room", choices);  
   dialog.setTitle("Lend Book");  
   dialog.setHeaderText("Choose lending option");  
   dialog.setContentText("Lend the book in the reading room or rent it out:");  
    
   Optional<String> result = dialog.showAndWait();  
   if (result.isPresent()) {  
   BookStatus bookStatus = result.get().equals("Reading Room") ? BookStatus.*IN\_READING\_ROOM* : BookStatus.*RENTED\_OUT*;  
    
   try {  
   lendBookItem.process(  
   LendBookItemInput.*builder*()  
   .isbn(bookIsbn)  
   .readerEmail(selectedReader.getEmail())  
   .bookStatus(bookStatus)  
   .build()  
   );  
   showAlert("Success", "Book has been successfully lent.");  
   } catch (Exception e) {  
   showAlert("Error", "Failed to lend the book: " + e.getMessage());  
   }  
   }  
   }  
    
   private void showAlert(String title, String message) {  
   Alert alert = new Alert(Alert.AlertType.*INFORMATION*);  
   alert.setTitle(title);  
   alert.setHeaderText(null);  
   alert.setContentText(message);  
   alert.showAndWait();  
   }  
    
   private void openReaderBookItemsView(ReaderDTO selectedReader) {  
   returnBookController.setReaderEmail(selectedReader.getEmail());  
   sceneChanger.changeScene((Stage) createReaderButton.getScene().getWindow(), returnBookScenePath);  
   }  
    
  }

**OperatorController** е JavaFX контролер клас, който управлява потребителския интерфейс за операторите в библиотечната система. Този клас предоставя функционалности като създаване и премахване на читатели, заемане на книги, уведомяване на читатели и изход от системата. Контролерът използва Spring за инжектиране на зависимости и интегрира бизнес логиката чрез различни услуги.

**Компоненти и функционалности**

**Аннотации**

* **@Component**: Декларира класа като Spring Bean, което позволява той да бъде инжектиран и управляван от Spring контекста.
* **@Slf4j**: Добавя логиране към класа чрез Lombok, осигурявайки лесен достъп до логиращия механизъм.

**UI Компоненти (FXML полета)**

* **TableView**: Таблици за визуализация на книги и читатели.
* **TableColumn**: Колони за атрибутите на книгите и читателите.
* **Button**: Бутони за действия като създаване и премахване на читатели, заемане на книги и изход.
* **Label**: Етикети за полетата и съобщения за грешки.

**Инжектирани услуги**

* **SceneChanger**: Услуга за смяна на сцените в JavaFX приложението.
* **CreateReader**: Услуга за създаване на нов читател.
* **RemoveReader**: Услуга за премахване на читател.
* **GetBooks**: Услуга за получаване на списък с книги.
* **UserLogout**: Услуга за изход от системата.
* **GetReaders**: Услуга за получаване на списък с читатели.
* **LendBookItem**: Услуга за заемане на книга.
* **NotifyReader**: Услуга за уведомяване на читател.
* **ReturnBookController**: Контролер за връщане на книги.

**Конфигурационни стойности**

* **createReaderPath**: Път към формата за създаване на читател, зададен чрез **@Value**.
* **loginFormPath**: Път към формата за вход, зададен чрез **@Value**.
* **returnBookScenePath**: Път към формата за връщане на книги, зададен чрез **@Value**.

package bg.tuvarna.frontend.controller;  
  
import bg.tuvarna.api.BookGenre;  
import bg.tuvarna.api.operations.administrator.registerbook.RegisterBook;  
import bg.tuvarna.api.operations.administrator.registerbook.RegisterBookInput;  
import bg.tuvarna.api.operations.administrator.registerbook.RegisterBookResult;  
import bg.tuvarna.frontend.utils.SceneChanger;  
import javafx.fxml.FXML;  
import javafx.scene.control.Button;  
import javafx.scene.control.ComboBox;  
import javafx.scene.control.Label;  
import javafx.scene.control.TextField;  
import javafx.scene.text.Text;  
import javafx.stage.Stage;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
@Component  
@Slf4j  
public class RegisterBookController {  
  
 @FXML  
 private TextField authorForm;  
  
 @FXML  
 private Label authorLabel;  
  
 @FXML  
 private Button backButton;  
  
 @FXML  
 private Label failedRegistrationLabel;  
  
 @FXML  
 private ComboBox<BookGenre> genreForm;  
  
 @FXML  
 private Label genreLabel;  
  
 @FXML  
 private TextField isbnForm;  
  
 @FXML  
 private Label isbnLabel;  
  
 @FXML  
 private Button registerBookButton;  
  
 @FXML  
 private Text registerBookLabel;  
  
 @FXML  
 private TextField titleForm;  
  
 @FXML  
 private Label titleLabel;  
  
 @Value("${fxml.paths.adminForm}")  
 private String adminViewPath;  
 @Autowired  
 private SceneChanger sceneChanger;  
 @Autowired  
 private RegisterBook registerBook;  
  
 @FXML  
 void initialize() {  
 genreForm.getItems().setAll(BookGenre.*values*());  
 }  
 @FXML  
 void goBack() {  
 sceneChanger.changeScene((Stage) backButton.getScene().getWindow(), adminViewPath);  
 }  
  
 @FXML  
 void registerBook() {  
 RegisterBookInput input = RegisterBookInput.*builder*()  
 .title(titleForm.getText())  
 .isbn(isbnForm.getText())  
 .author(authorForm.getText())  
 .bookGenre(genreForm.getValue())  
 .build();  
  
 try {  
 RegisterBookResult result = registerBook.process(input);  
 *log*.info("Book registered successfully: {}", result);  
 } catch (Exception e) {  
 *log*.error("Error registering book: {}", e.getMessage());  
 failedRegistrationLabel.setText("Failed to register book: " + e.getMessage());  
 }  
 finally {  
 goBack();  
 }  
 }  
  
}

**RegisterBookController** е JavaFX контролер клас, който управлява формата за регистрация на нови книги в библиотечната система. Този клас предоставя функционалност за въвеждане на данни за нова книга, проверка на валидността на данните и регистрация на книгата чрез бизнес логиката, интегрирана със Spring.

**Компоненти и функционалности**

**Аннотации**

* **@Component**: Декларира класа като Spring Bean, което позволява той да бъде инжектиран и управляван от Spring контекста.
* **@Slf4j**: Добавя логиране към класа чрез Lombok, осигурявайки лесен достъп до логиращия механизъм.

**UI Компоненти (FXML полета)**

* **TextField**: Полета за въвеждане на заглавие, автор и ISBN на книгата.
* **ComboBox**: Комбинирано поле за избор на жанр на книгата.
* **Button**: Бутони за действия като връщане назад и регистрация на книга.
* **Label**: Етикети за полетата и съобщения за грешки.
* **Text**: Текст за заглавието на формата.

**Инжектирани услуги**

* **SceneChanger**: Услуга за смяна на сцените в JavaFX приложението.
* **RegisterBook**: Услуга за регистрация на нова книга.

**Конфигурационни стойности**

* **adminViewPath**: Път към формата за администрация, зададен чрез **@Value**.

package bg.tuvarna.frontend.controller;  
  
import bg.tuvarna.api.operations.operator.getbookitemsforreader.GetBookItemsForReaderInput;  
import bg.tuvarna.api.operations.operator.getbookitemsforreader.GetBookItemsForReaderResult;  
import bg.tuvarna.api.operations.operator.returnbook.ReturnBookItem;  
import bg.tuvarna.api.operations.operator.returnbook.ReturnBookItemInput;  
import bg.tuvarna.api.operations.util.BookItemDTO;  
import bg.tuvarna.core.service.GetBookItemsForReaderService;  
import bg.tuvarna.frontend.utils.SceneChanger;  
import javafx.fxml.FXML;  
import javafx.scene.control.Alert;  
import javafx.scene.control.Button;  
import javafx.scene.control.TableColumn;  
import javafx.scene.control.TableView;  
import javafx.scene.control.cell.PropertyValueFactory;  
import javafx.stage.Stage;  
import lombok.Setter;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
import java.util.List;  
  
@Component  
public class ReturnBookController {  
  
 @FXML  
 private TableView<BookItemDTO> booksTable;  
  
 @FXML  
 private TableColumn<BookItemDTO, String> idColumn;  
  
 @FXML  
 private TableColumn<BookItemDTO, String> isbnColumn;  
  
 @FXML  
 private Button returnBookButton;  
  
 @FXML  
 private Button backButton;  
  
 @FXML  
 private TableColumn<BookItemDTO, String> statusColumn;  
  
 @FXML  
 private TableColumn<BookItemDTO, String> titleColumn;  
  
 @Setter  
 private String readerEmail;  
  
 @Autowired  
 private GetBookItemsForReaderService getBookItemsForReader;  
 @Autowired  
 private ReturnBookItem returnBookItem;  
  
 @Autowired  
 private SceneChanger sceneChanger;  
 @Value("${fxml.paths.operatorForm}")  
 private String operatorFormPath;  
  
 @FXML  
 public void initialize() {  
 idColumn.setCellValueFactory(new PropertyValueFactory<>("id"));  
 titleColumn.setCellValueFactory(new PropertyValueFactory<>("title"));  
 isbnColumn.setCellValueFactory(new PropertyValueFactory<>("isbn"));  
 statusColumn.setCellValueFactory(new PropertyValueFactory<>("bookStatus"));  
  
 if (readerEmail != null && !readerEmail.isEmpty()) {  
 loadBookItems();  
 }  
 }  
  
 @FXML  
 void returnBook() {  
 BookItemDTO selectedBookItem = booksTable.getSelectionModel().getSelectedItem();  
 if (selectedBookItem != null) {  
 returnBookItem.process(ReturnBookItemInput.*builder*()  
 .bookItemId(selectedBookItem.getId())  
 .build());  
 } else {  
 // Show an alert if no book is selected  
 showAlert("No Selection", "No book selected", "Please select a book to return.");  
 }  
 initialize();  
 }  
  
 private void loadBookItems() {  
 GetBookItemsForReaderResult result = getBookItemsForReader.process(  
 GetBookItemsForReaderInput.*builder*()  
 .email(readerEmail)  
 .build()  
 );  
  
 List<BookItemDTO> bookItemDTOList = result.getBookItemDTOList();  
 booksTable.getItems().clear();  
 booksTable.getItems().addAll(bookItemDTOList);  
 }  
  
 @FXML  
 void goBack() {  
 sceneChanger.changeScene((Stage) backButton.getScene().getWindow(), operatorFormPath);  
 }  
  
 private void showAlert(String title, String header, String content) {  
 Alert alert = new Alert(Alert.AlertType.*INFORMATION*);  
 alert.setTitle(title);  
 alert.setHeaderText(header);  
 alert.setContentText(content);  
 alert.showAndWait();  
 }  
}

**ReturnBookController** е JavaFX контролер клас, който управлява интерфейса за връщане на книги от читатели в библиотечната система. Този клас предоставя функционалности за зареждане на заемани книги, избор на книга за връщане и връщане на избраната книга чрез бизнес логиката, интегрирана със Spring.

**Компоненти и функционалности**

**Аннотации**

* **@Component**: Декларира класа като Spring Bean, което позволява той да бъде инжектиран и управляван от Spring контекста.

**UI Компоненти (FXML полета)**

* **TableView**: Таблица за визуализация на заемани книги.
* **TableColumn**: Колони за атрибутите на заеманите книги.
* **Button**: Бутони за действия като връщане на книга и връщане назад.
* **Alert**: Диалогов прозорец за показване на съобщения към потребителя.

**Инжектирани услуги**

* **GetBookItemsForReaderService**: Услуга за получаване на списък с заемани книги от читател.
* **ReturnBookItem**: Услуга за връщане на книга.
* **SceneChanger**: Услуга за смяна на сцените в JavaFX приложението.

**Конфигурационни стойности**

* **operatorFormPath**: Път към формата за оператори, зададен чрез **@Value**.

**Полета**

* **readerEmail**: Имейл на текущия читател, зададен чрез сетър.

**UML Диаграми:**

A diagram of a diagram

Description automatically generated

A screenshot of a computer

Description automatically generated