ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – ВАРНА

КАТЕДРА „СИТ“

Дисциплина „ООП – Втора част“

Курсов проект

Програма за информационна система библиотека

*Изготвили:*

*Студент: Силвия Йорданова*

*Факултетен №: 18621721*

*Студент: Теодора Страхилова*

*Факултетен №: 18621696*

*Курс: III*

*Специалност: СИТ*

*Група: IIб*

Дата: 0.12.2020 год. Проверил: …………………………..

/ доц. Христо Ненов /

Задание:

Да се разработи информационна система – библиотека. Програмата съхранява и обработва данни за книги и читатели. Системата позволява множествен достъп.

Системата поддържа два вида потребители администратор и клиенти (оператори и читатели) с различни роли за достъп до функционалностите в системата.

Операции за работа с потребители.

* Създаване на оператори от администратор
* Създаване и отписване на читатели от оператор
* Формуляр за създаване на читателски профил
* Отдаване на книги

Системата поддържа операции за работа с книги:

* Добавяне на нови книги (Инвентарен номер, заглавие, автор, жанр..)
* Заемане на книги (различно ниво на сигурност при заемането на книги (читалня, изнасяне)
* Връщане на книга
* Бракуване на повредени книги
* Архивиране на стари издания (ползвани само в читалня)

Системата поддържа справки за:

* Подадени формуляри (дата, статус, съдържание на формуляра)
* Книги (състояние на книгата, информация за книгата)
* Потребители (дата на одобрение, списък с книги, потребителска информация)
* Рейтинг на потребителите (лоялни и нелоялни читатели)

Системата поддържа Известия за събития:

* Заявка за откриване на читателски профил
* Известяване за необходимост за архивиране на книга
* Просрочване връщането на книги

Изисквания:

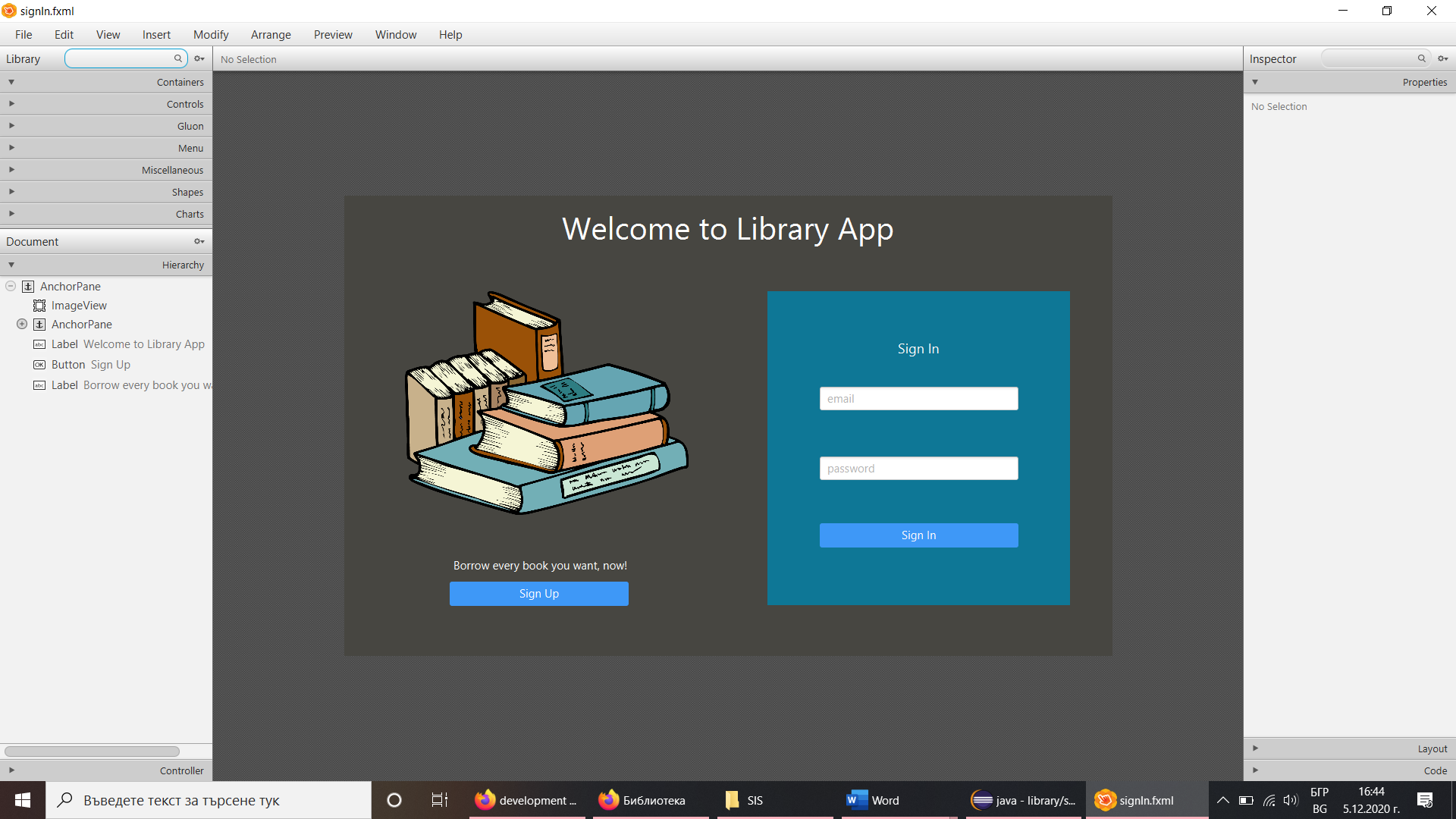
Документация – проекта трябва да е добре документиран

1. Заглавна страница съдържаща името на университета, факултета, катедрата, специалността. Темата на проекта, имената на студентите от екипа, факултетни номера.
2. Пълното задание (условие) на проекта.
3. Анализ на проблема:
   1. Функционални изисквания
   2. Структура на проекта
   3. Дефиниция на модулите на системата
4. Проектиране на системата:
   1. Проектиране на отделните модули (база данни, GUI, бизнес логика)
   2. UML – Use Case, Class Diagram, Sequence diagram и други
   3. Концептуален модел на базата от данни (ER диаграма (модел на Чен)
5. Реализация на системата:
   1. Реализация на базата от данни (PostGres, Oracle, MS SQL, My SQL) – Реализация схема, описание на таблиците
   2. Реализация на слоя за работа с базата данни (DAO, JPA (Hibernate)
   3. Реализация на бизнеслогика и графичен интерфейс – Java, JavaFX
   4. Реализация на модул за регистриране на събития в системата – log4J
6. Тестови резултати:
   1. Junit tests
   2. Функционални тестове
   3. Интеграционни тестове
7. **Анализ на проблема**

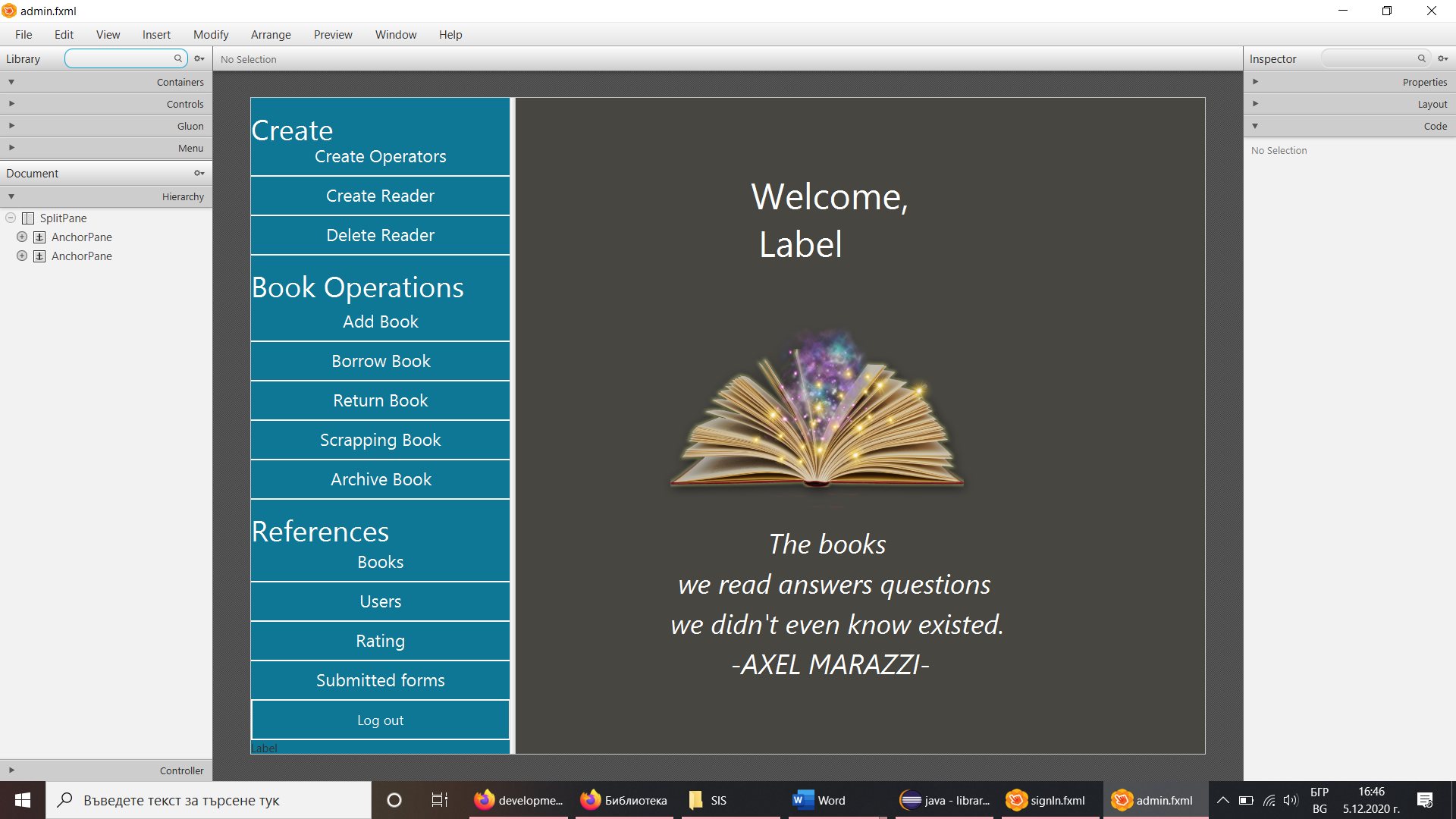
* *Функционални изисквания*
* В зависимост от ролята потребителите да имат различен достъп до функционалностите
* Читателите да могат да се регистрират в системата и да им се добавя роля
* Системата да има форма за вход
* Да поддържа различни прозорци според ролята на потребителя и според функционалностите
* При операциите с книги добавяне, бракуване и архивиране да се извършват от оператор или администратор
* Достъп до справките според ролята
* *Структура на проекта*
* База данни -Oracle Database
* Domain model- в него се намират Entities /обектите в системата/ отговарящи на таблиците в базата.
* DAO layer-за взаимодействие с базата данни. Изпълнява основните(CRUD) операции.
* Service layer – бизнес логиката. Комуникира с DAO за менажиране на моделите в базата.
* Потребителски интерфейс- в него се намират всички изгледи, контролери, които използват Service layer за изпълнение на зададените от потребителя функционалности.
* *Дефиниция на модулите на системата*
* Модул за работа с база данни
* GUI – модул, който съдържа презентационния слой на приложението
* Бизнес логика
* Модул, който да предоставя на потребителите лесна интеракция със системата или т.н. потребителски интерфейс

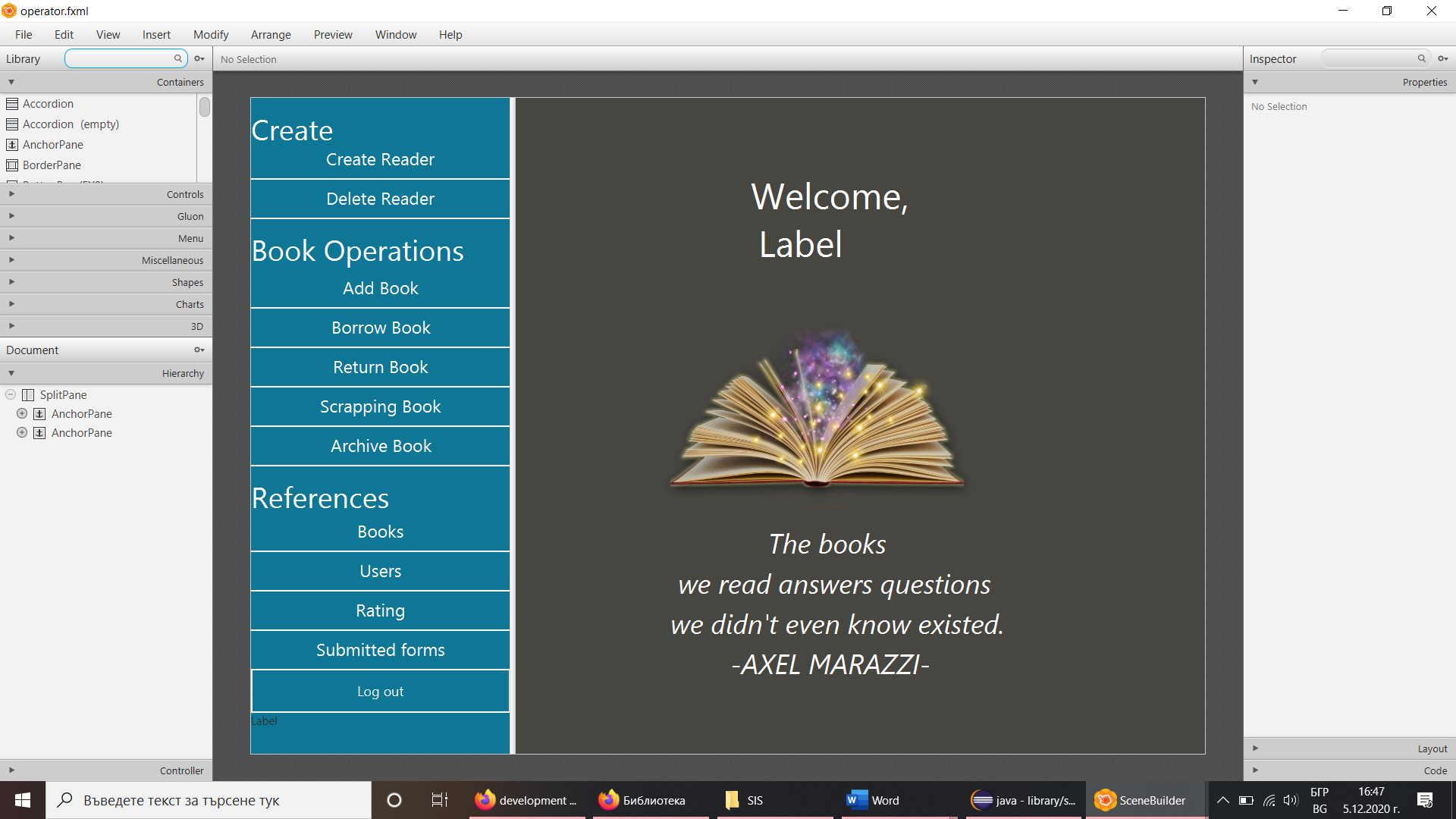
1. **Проектиране на системата**

* Графичен интерфейс изграден с помощта на SceneBuilder .
* Модулът за базата данни ще се изпълни чрез DAO & Hibernate. Трябва да поддържа основните CRUD /create, read,update,delete/функции.
* Бизнес логика Service layer който взаимодейства със слоя за базата и с графичния интерфейс чрез класовете контролери

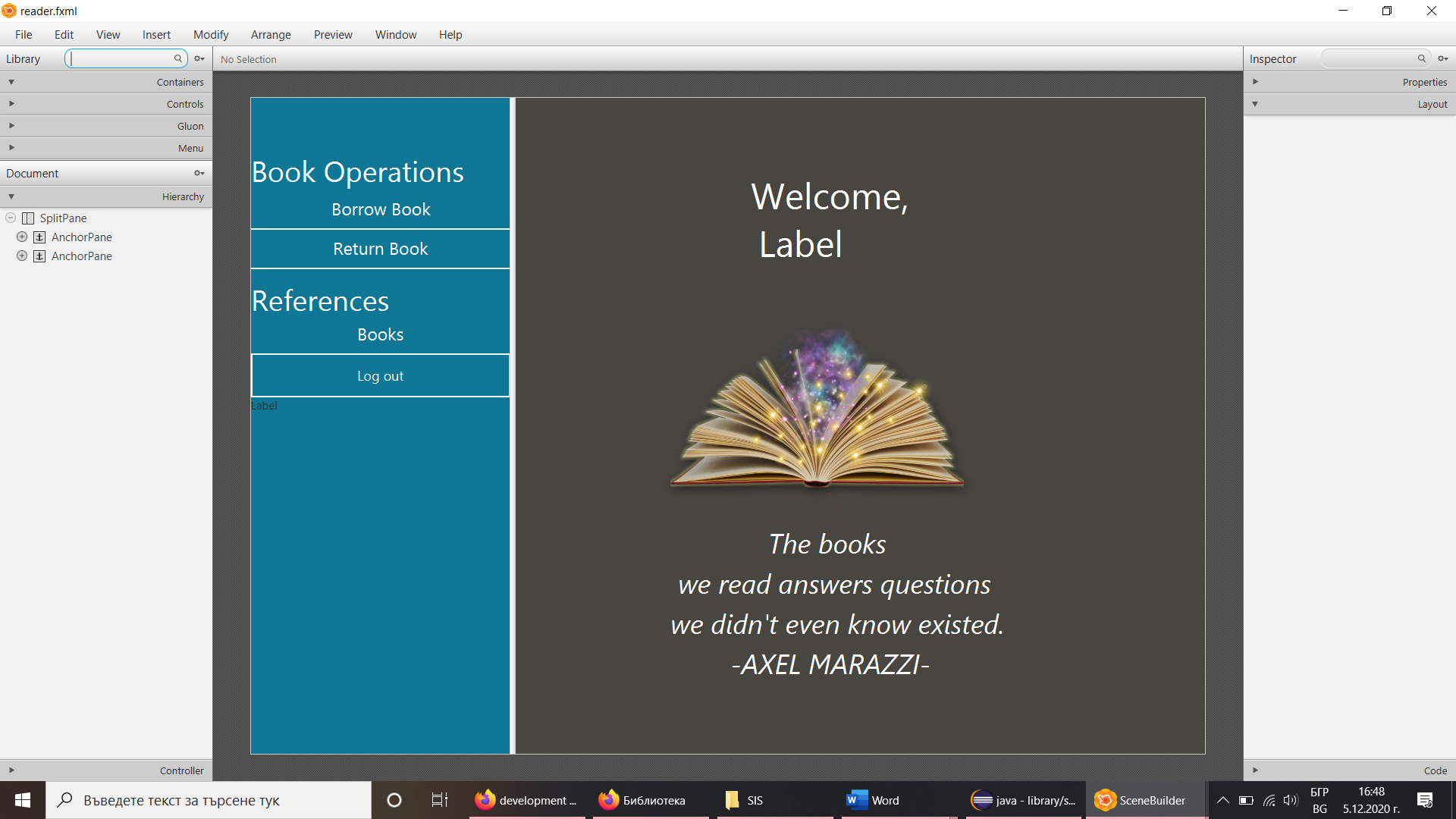
Изглед при влизане в системата signIn.fxml

Изглед на админа admin.fxml

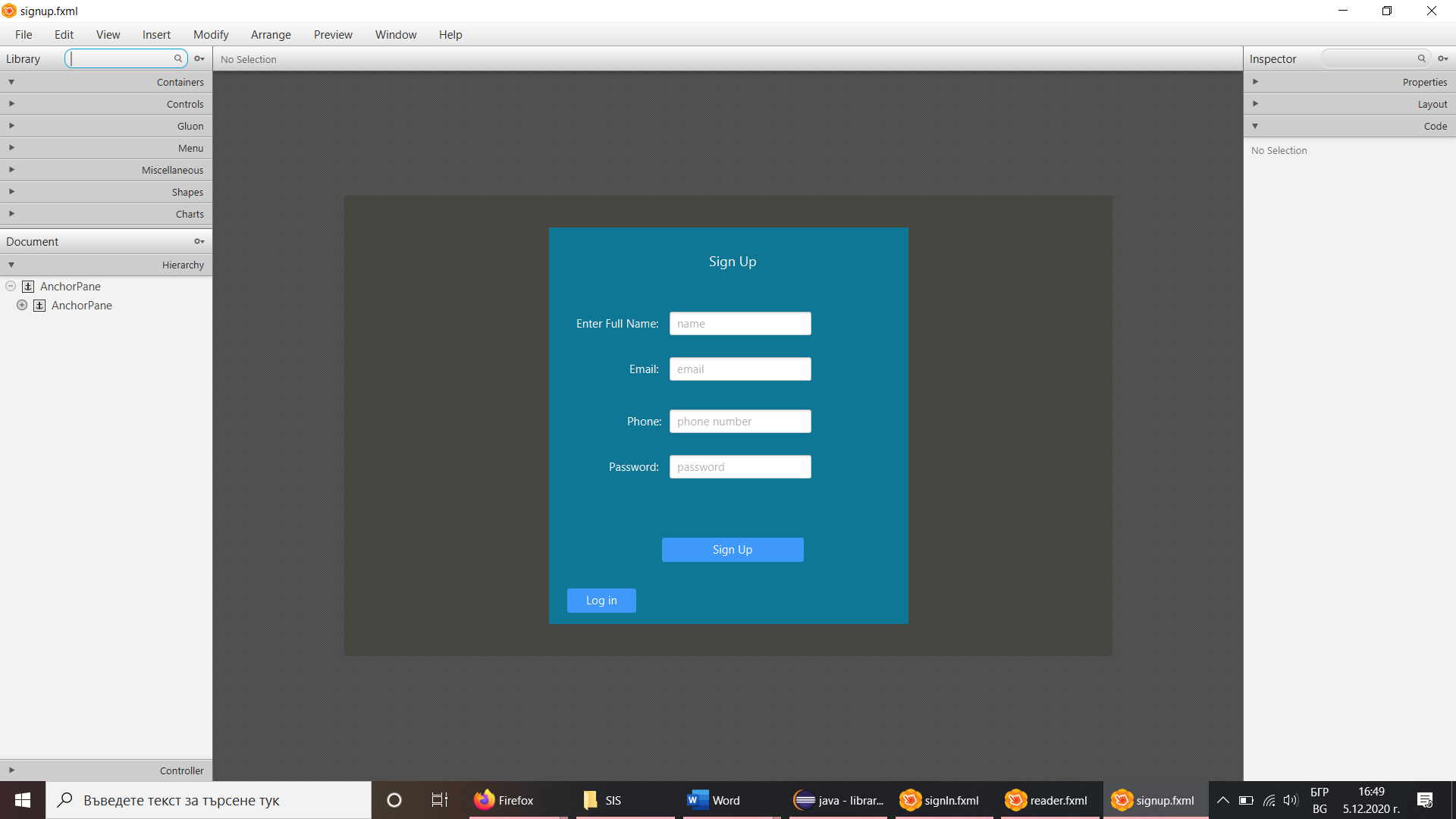


Изглед на оператора operator.fxml

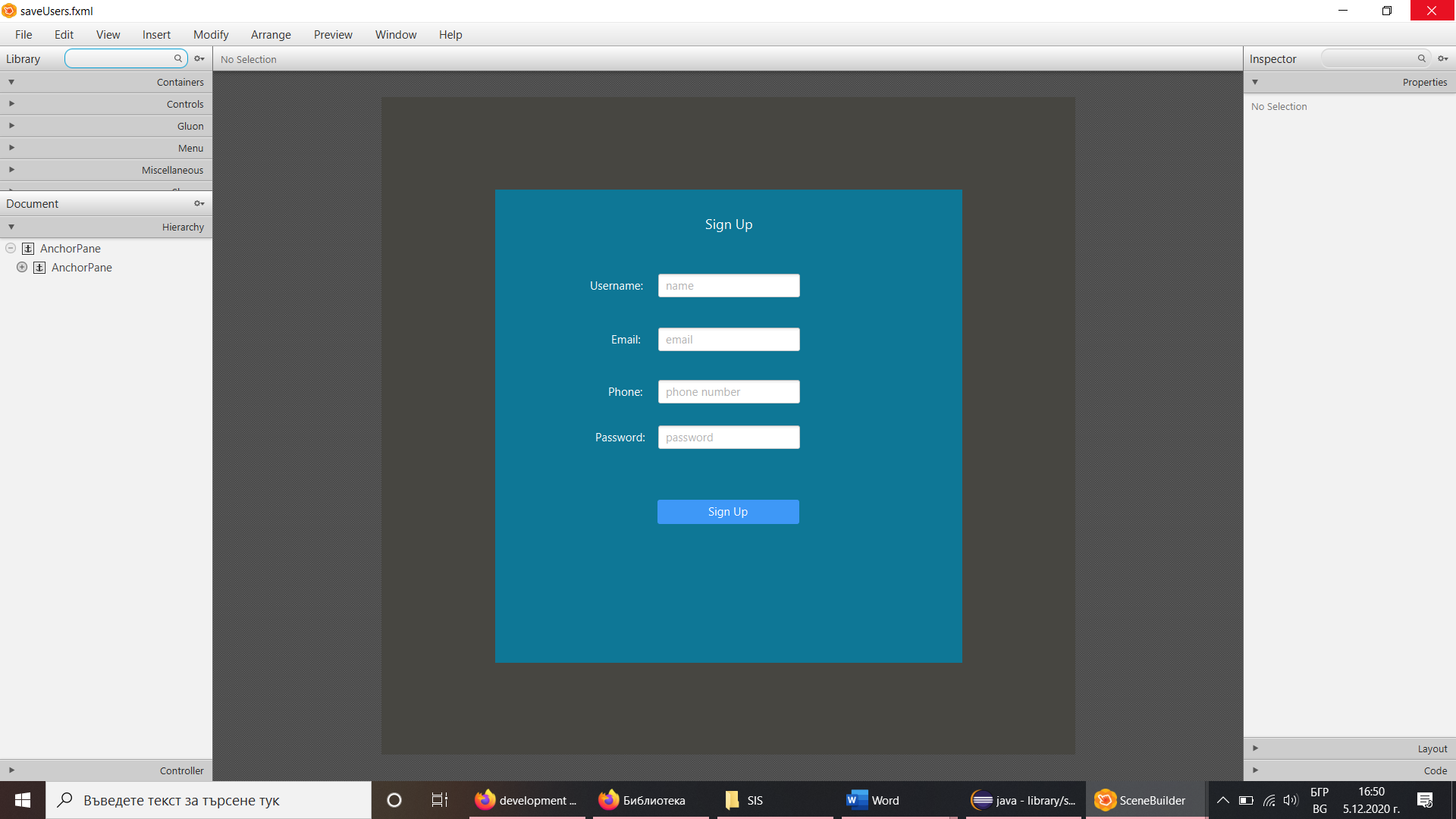
Изглед на читателя reader.fxml



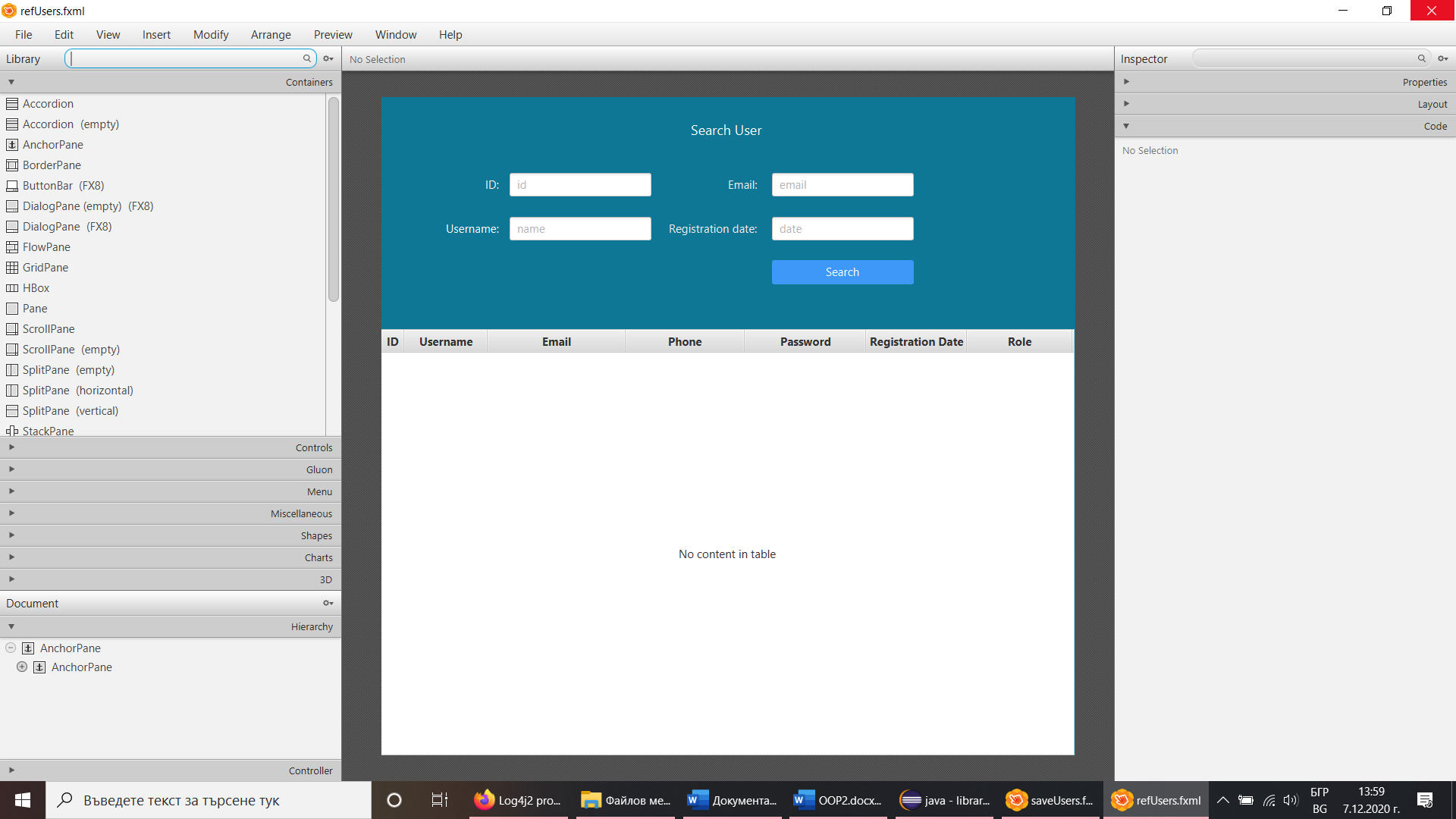
Изглед при регистрация на читател signup.fxml



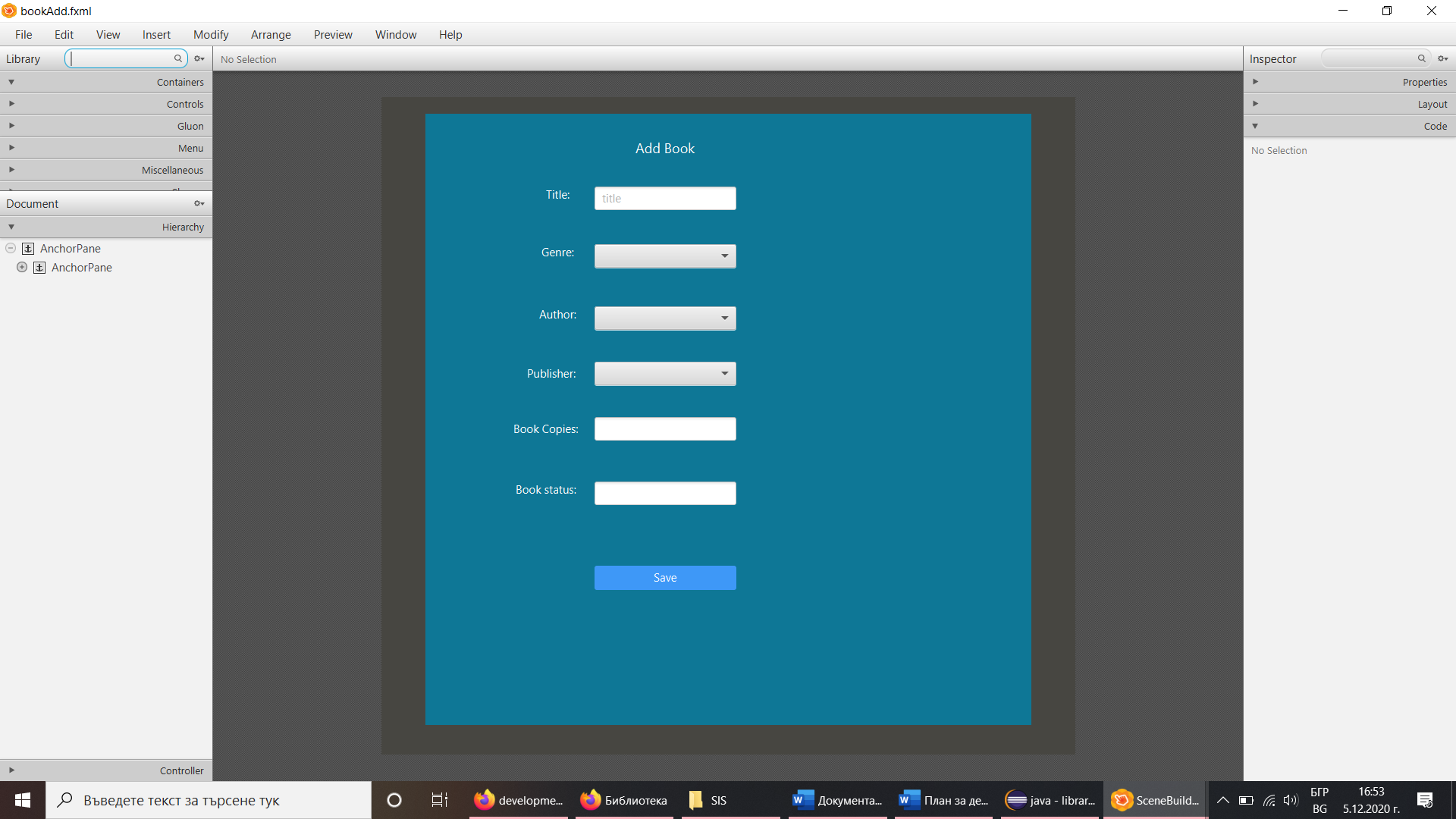
Изглед при запазване на опретор и читател от Админ и запазване на читател от Оператор saveUsers.fxml



Изглед при справки на потребител refUsers.fxml

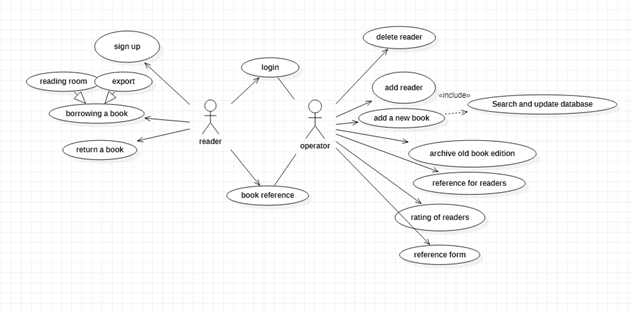


Изглед при добавяне на книга bookAdd.fxml



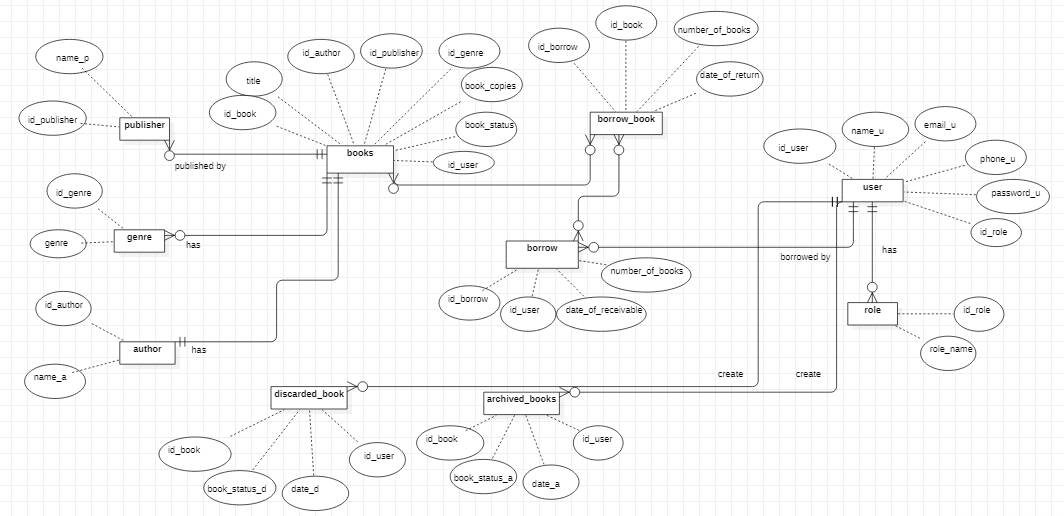
* UML диаграми:
* Use case - е техника за определяне на функционалните изисквания на една система. “Use case” (Случай на употреба, Случай на използване и др.) е независима част от функционалността на моделираната система, притежаваща резултантна ценност за своите изпълнители (актьори).

Пример за Use case:

Описва какви действия могат да извършват актьорите, които са reader(читател) и operator(оператор) в информационната система библиотека.

* ER diagram - диаграмата се използва за моделиране и проектиране на релационни бази данни. Тя отразява обектите, техните свойства и взаимодействията (връзки) между тях.

Пример за ER Diagram:

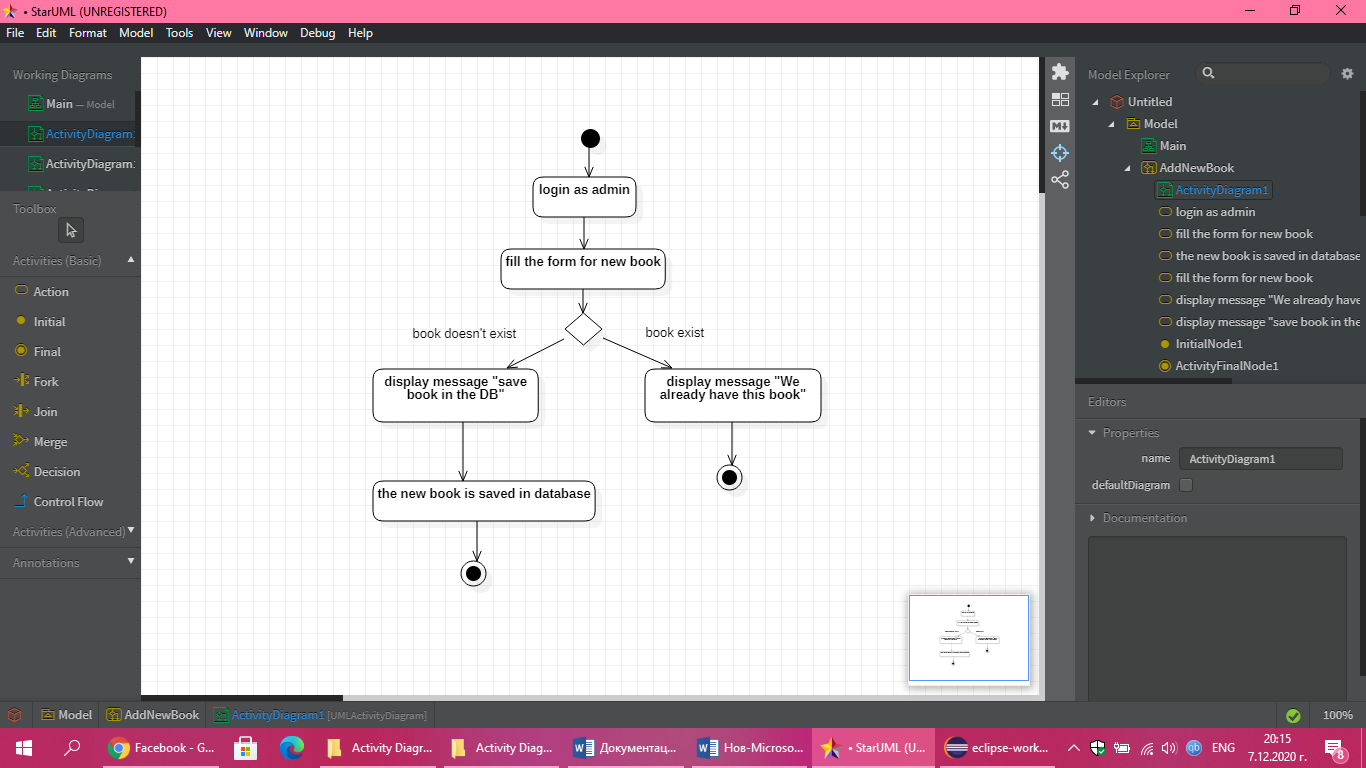


* Activity diagrams - „Диаграмата на активност” е поведенческа диаграма, която описва динамичните аспекти на системата.

Примери за Activity diagrams:

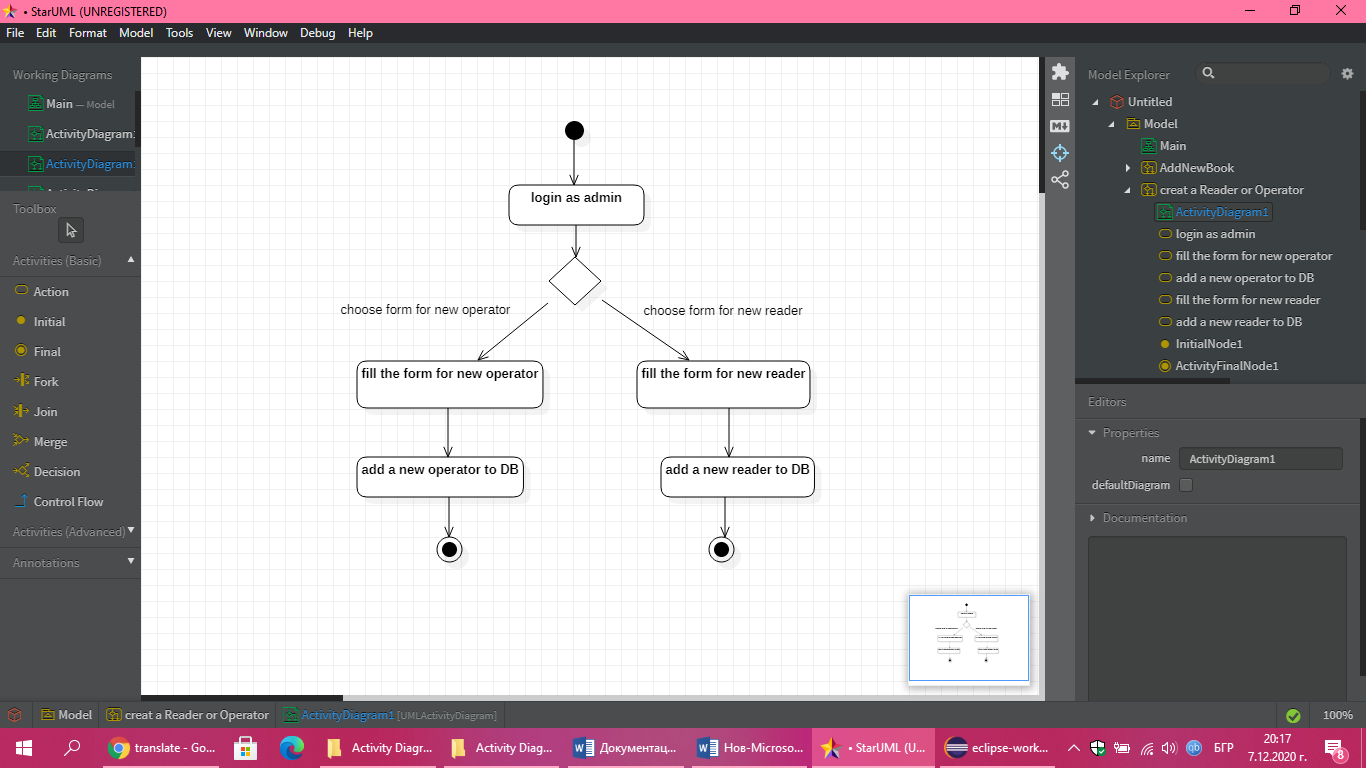
* + Добавяне на нова книга

Диаграмата отговаря на форма за добавяне на нова книга, след като администратор е влязъл в системата. Изписват се нужните съобщения, след проверка на попълнената форма за нова книга. Ако тя не съществува, се запазва в база данни.



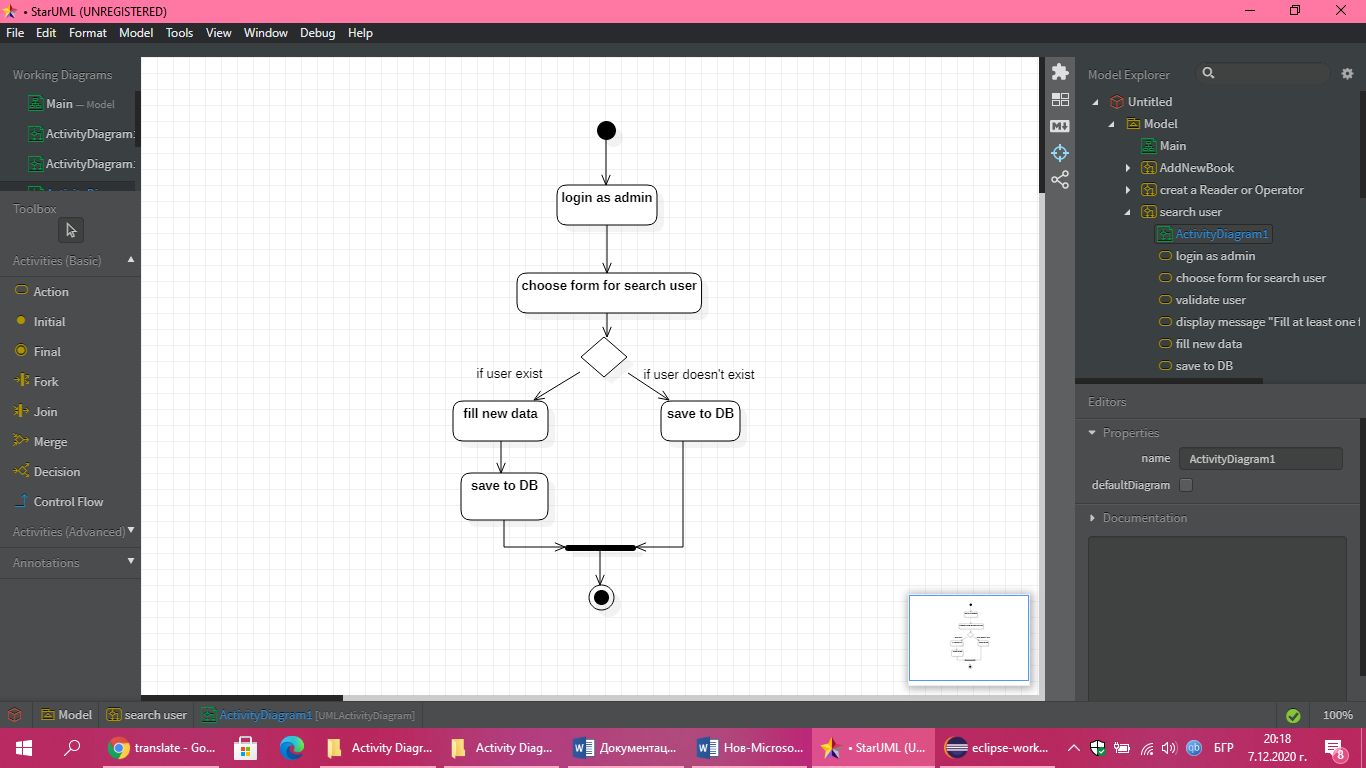
* + Създаване на оператор и читател

В тази диаграма е описано как администратор, може да избере форма за създаване на оператор или читател. Само той има това право. След като попълни нужните полета се създава запис в базата данни.



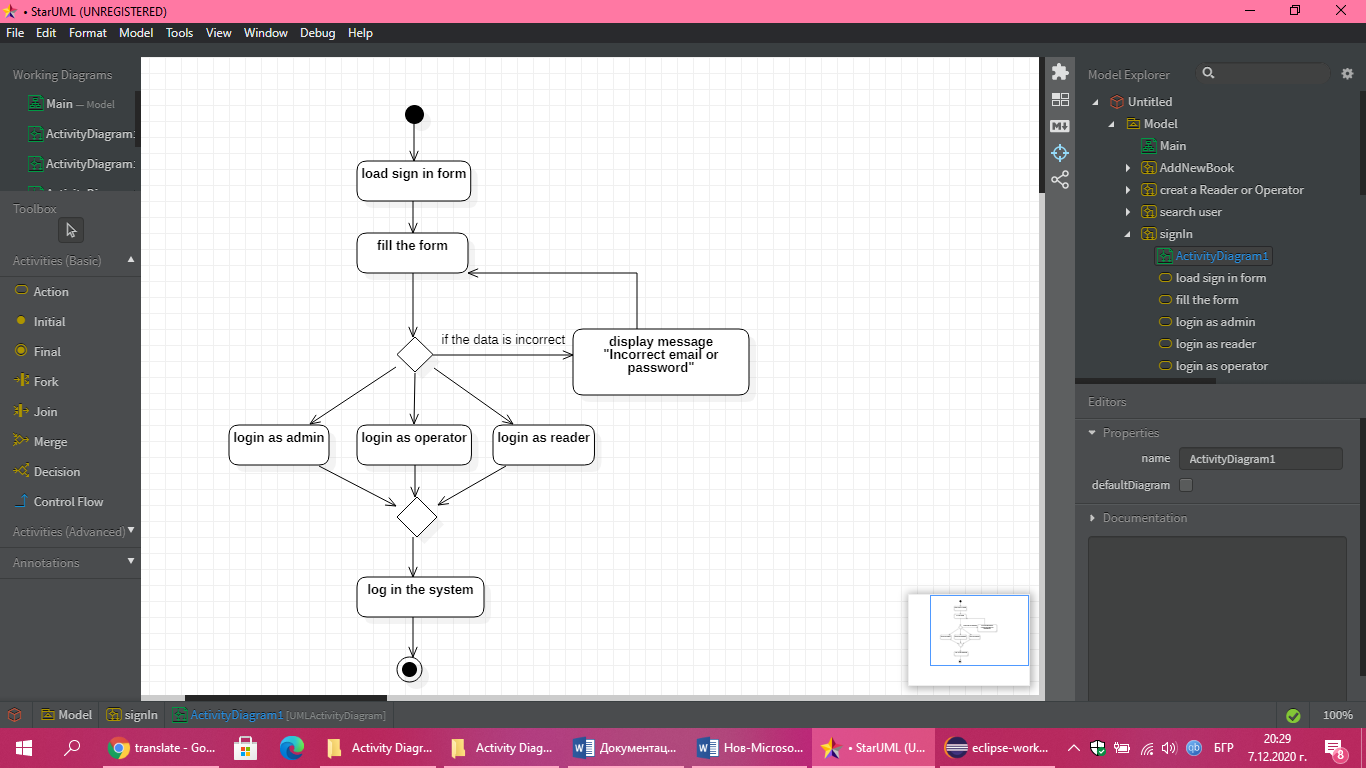
* + За търсене на потребител

Отговаря на формата за търсене на потребител.Когато въведеният потребител не съществува, се запазва в база данни. Ако той съществува, се попълват нови данни и тогава се запазва.



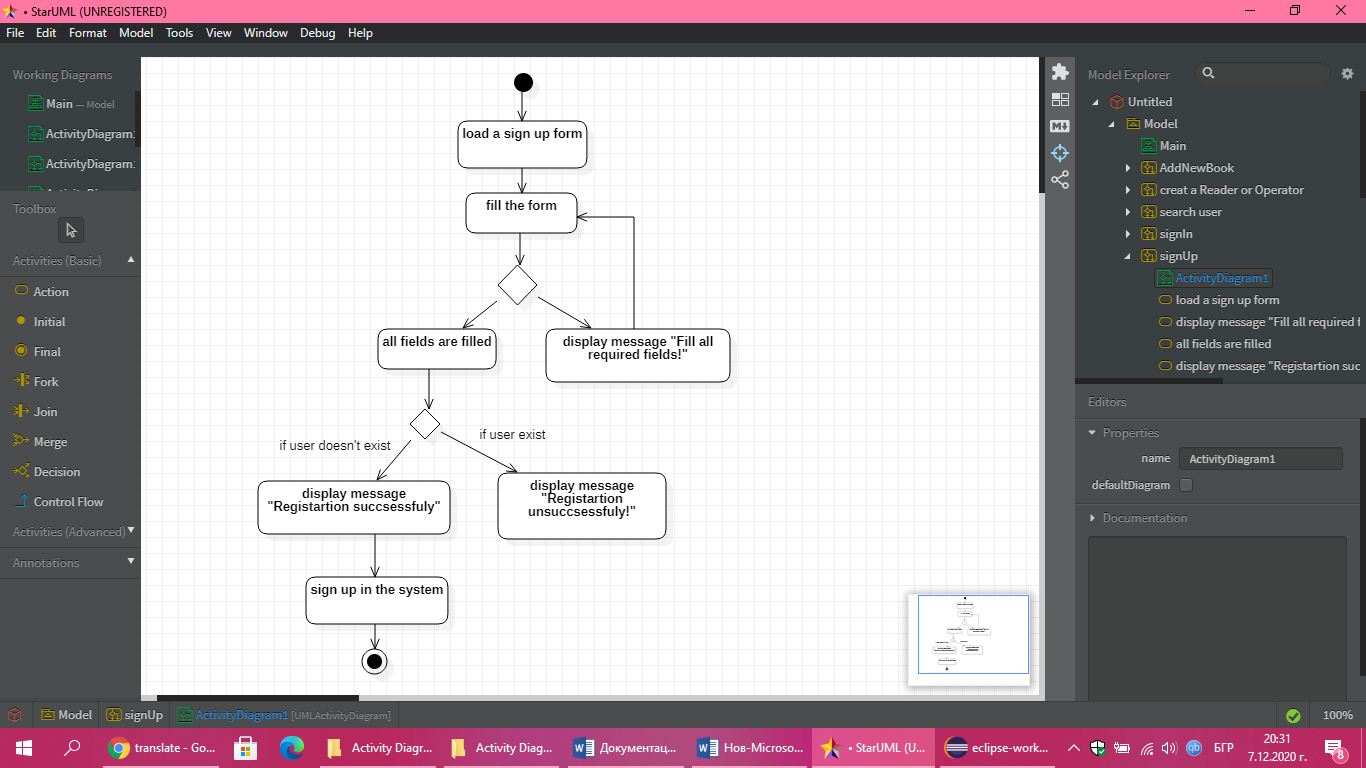
* + Вписване в системата

Във формата за вписване в системата, след попълването й, се проверяват въведените данни. Когато са грешни, се изписва необходимото съобщение и трябва да се въведат нови. Ако са верни въведените данни, има проверка за определяне на типа потребител, който влиза в системата.



* + Регистриране в системата

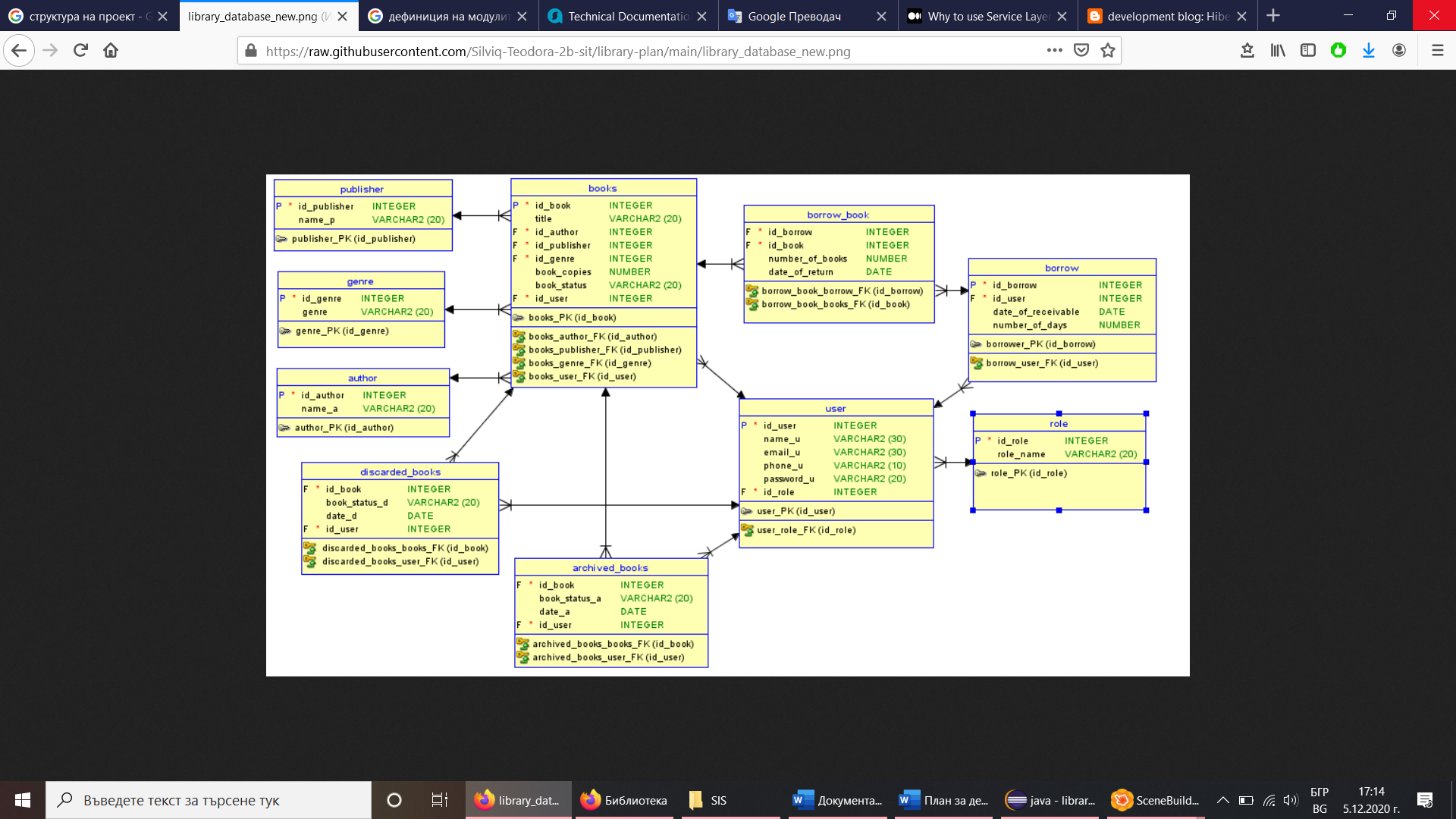
Тази диаграма представя регистриране на потребител. След попълване на полетата, се извършва проверка, която изписва съобщение, когато има пропуск. В зависимост от това потребителят съществува или не, се показват нужните съобщения.



1. **Реализация на системата**

* База данни Oracle Database

Relational diagram



На таблиците от базата данни отговарят и класове /Entities/. Всеки клас има анотации, които помагат на Hibernate да създаде таблиците в база данни.

**User клас отговаря на таблица user**

Този клас съдържа полета- имената на колоните в таблицата. Анотация @ManyToOne(cascade = CascadeType.ALL) , показва че много users могат да имат 1 роля.

Също така съдържа @NamedQueries за търсене по потребителско име и парола и за търсене само по потребителско.

import java.sql.Date;

@Entity

@NamedQueries( { @NamedQuery(name = "User.byEmailAndPass", query = "from User where userEmail=? and userPass=?"),

@NamedQuery(name = "User.Registration", query = "from User where userName=? and userEmail=?"),

@NamedQuery(name = "User.Search", query = "from User where userName=?")})

@Table(name="USERS")

public class User {

private int userId;

private String userName;

private String userEmail;

private String userPhone;

private String userPass;

private Role userRole;

private Date registrationDate;

public User() {

super();

// TODO Auto-generated constructor stub

}

public User(String userName, String userEmail, String userPhone, String userPass, Role userRole) {

super();

this.userName = userName;

this.userEmail = userEmail;

this.userPhone = userPhone;

this.userPass = userPass;

this.userRole = userRole;

}

@Id

@Column(name = "UserId")

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "user\_generator")

@SequenceGenerator(name = "user\_generator", sequenceName = "USERS\_SEQUENCE",allocationSize = 1,initialValue = 7)

public int getUserId() {

return userId;

}

public void setUserId(int userId) {

this.userId = userId;

}

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

public String getUserEmail() {

return userEmail;

}

public void setUserEmail(String userEmail) {

this.userEmail = userEmail;

}

public String getUserPhone() {

return userPhone;

}

public void setUserPhone(String userPhone) {

this.userPhone = userPhone;

}

public String getUserPass() {

return userPass;

}

public void setUserPass(String userPass) {

this.userPass = userPass;

}

@ManyToOne(cascade = CascadeType.ALL)

public Role getUserRole() {

return userRole;

}

public void setUserRole(Role userRole) {

this.userRole = userRole;

}

@Override

public String toString() {

return "User [userId=" + userId + ", userName=" + userName + ", userEmail=" + userEmail + ", userPhone=" + userPhone

+ ", userPass=" + userPass + ", userRole=" + userRole + "]";

}

public Date getRegistrationDate() {

return registrationDate;

}

public void setRegistrationDate(Date registrationDate) {

this.registrationDate = registrationDate;

}

}

Клас Role, Genre и Publisher са на същия принцип.

Клас Book и клас Author са свързани с връзка ManyToMany , а Genre , Publisher , User са свързани с книгата с ManyToOne.

Конфигурационен файл на hibernate намира се в \src\main\resources с име hibernate.cfg.xml. Той дава информация на на Hibernate за това къде да се намерят мапинг класовете, които отговарят за мапинга между Java класовете и таблиците на базата данни. Чрез тези класове автоматично се създават таблици в базата данни със съответния ред <property name=*"hibernate.hbm2ddl.auto"*>update</property> .Конфигурационният файл също така предоставя информация за подробности за базата данни, като URL адрес на базата данни, драйвър клас, потребителско име, парола, диалект , контекстов клас за сесията.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name=*"connection.driver\_class"*>oracle.jdbc.OracleDriver</property>

<property name=*"connection.url"*>jdbc:oracle:thin:@localhost:1521:xe</property>

<property name=*"connection.username"*>LIBRARY</property>

<property name=*"connection.password"*>SISITO21</property>

<property name=*"dialect"*>org.hibernate.dialect.Oracle8iDialect</property>

<property name=*"hibernate.current\_session\_context\_class"*>thread</property>

<property name=*"show\_sql"*>true</property>

<property name=*"hibernate.hbm2ddl.auto"*>update</property>

<mapping class=*"com.libapp.User"* />

<mapping class=*"com.libapp.Role"* />

<mapping class=*"com.libapp.Author"* />

<mapping class=*"com.libapp.Genre"* />

<mapping class=*"com.libapp.Publisher"* />

<mapping class=*"com.libapp.Book"* />

</session-factory>

</hibernate-configuration>

**За връзката с базата ползваме dependency за драйвер jdbc**

<dependency>

<groupId>com.oracle.database.jdbc</groupId>

<artifactId>ojdbc8</artifactId>

<version>19.3.0.0</version>

</dependency>

* **Слой за работа с база данни**

Слоят за работа базата от данни е реализиран посредством DAO(Data Access Object) или така наречения обект за достъпване на данни, който ни позволява да изпълняваме нужните ни действия върху базата от данни .

**DAOInt интерфейс**

Декларира методите за запис, ъпдейт, търсене по id, изтриване, заявка за взимане на всички записи от таблица, заявка с параметри

package com.libapp.dao;

import java.util.List;

import java.util.Map;

public interface DAOInt<T> {

T get(Class<T>cl,int id);

void save(T t);

void update(T t);

void delete(T t);

List<T> query(String from);

T namedQuery(String query, Map<Integer, Object> params);

List<T> createQuery(String hsql, Map<String, Object> params);

}

**DAOIpml клас имплементиращ интерфейс DAO**

import java.util.List;

import java.util.Map;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import org.hibernate.Query;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

public class DAOImpl<T> implements DAOInt<T>{

private SessionFactory sessionFactory;

private Logger LOGGER;

public DAOImpl(Class<T> cl, SessionFactory sessionFactory) {

this.LOGGER = LogManager.getLogger(cl);

this.sessionFactory = sessionFactory;

if (sessionFactory == null)

throw new RuntimeException("Session factory is null!!!");

}

@Override

public T get(Class<T> cl, int id) {

LOGGER.info("STARTED- Get user by ID");

Session session = sessionFactory.getCurrentSession();

session.beginTransaction();

T element = (T) session.get(cl, id);

session.getTransaction().commit();

LOGGER.info("FINISHED - Get user by ID");

return element;

}

@Override

public void save(T t) {

LOGGER.info("STARTED- Save user");

Session session = sessionFactory.getCurrentSession();

session.beginTransaction();

session.save(t);

session.getTransaction().commit();

LOGGER.info("FINISHED- Save user");

}

@Override

public void update(T t) {

LOGGER.info("STARTED- Update user");

Session session = sessionFactory.getCurrentSession();

session.beginTransaction();

session.update(t);

session.getTransaction().commit();

LOGGER.info("FINISHED- Update user");

}

@Override

public void delete(T t) {

LOGGER.info("STARTED- Delete user");

Session session = sessionFactory.getCurrentSession();

session.beginTransaction();

session.delete(t);

session.getTransaction().commit();

LOGGER.info("FINISHED- Delete user");

}

@Override

public List<T> query(String from){

LOGGER.info("STARTED- Query");

Session session = sessionFactory.getCurrentSession();

session.beginTransaction();

List<T> resultList=(List<T>) session.createQuery(from).list();

session.getTransaction().commit();

LOGGER.info("FINISHED- Query. Result suze="+resultList.size());

return resultList;

}

@Override

public T namedQuery(String query, Map<Integer, Object> params){

LOGGER.info("STARTED- Named Query");

Session session = sessionFactory.getCurrentSession();

session.beginTransaction();

Query q=session.getNamedQuery(query);

for (Integer i : params.keySet()) {

q.setParameter(i, params.get(i));

}

T result= (T) q.uniqueResult();

session.getTransaction().commit();

LOGGER.info("FINISHED- Named Query");

return result;

}

@Override

public List<T> createQuery(String hsql, Map<String, Object> params) {

LOGGER.info("STARTED - Create Query with parameters returns list");

Session session = sessionFactory.getCurrentSession();

session.beginTransaction();

Query query = session.createQuery(hsql);

if (params != null) {

for (String i : params.keySet()) {

query.setParameter(i, params.get(i));

}

}

List<T> result = null;

result = query.list();

LOGGER.info("FINISHED - Create Query. Result size="+result.size());

session.getTransaction().commit();

return result;

}

Класът съдържа две полета:

**private SessionFactory sessionFactory;** // за създаването на Session

private Logger LOGGER; // за регистриране на събитие в системата

Операции и конструктори:

**public DAOImpl(Class<T> cl, SessionFactory sessionFactory)//** за инициализиране на sessionFactory и LOGGER

**T get(Class<T>cl,int id);** търсене по id като приема за параметри темплейт клас /може да е всяко entity/ и id, създава сесия и намира записа със съответни данни

**void save(T t);** запис в базата данни параметър обект

**void update(T t);** ъпдейт на запис параметър обект

**void delete(T t)**; изтриване на запис параметър обект

**List<T> query(String from);** Получава параметър заявка под формата на стринг и връща лист със всички записи от таблицата на зададения клас

**T namedQuery(String query, Map<Integer, Object> params);** получава като параметри стринг на името на namedQuery и Map с параметрите ключът е номера на параметъра а стойността самия параметър

**List<T> createQuery(String hsql, Map<String, Object> params);** получава заявка като стринг и Map с параметрите ключът е името на параметъра/стренг/, а стойността самия параметър

* **Бизнес логика и графичен интерфейс**

Бизнес логиката е реализирана чрез Service слой

**Service интерфейс**

import com.libapp.dao.DAOInt;

public interface Service<T> extends DAOInt<T> {

}

Разширява ДАО интерфейса и тук могат да се добавят още методи. В нашия случай не сме добавили още.

**ServiceImpl имплементира Service**

package com.libapp.service;

import java.util.List;

import java.util.Map;

import org.hibernate.SessionFactory;

import com.libapp.dao.DAOImpl;

import com.libapp.dao.DAOInt;

public class ServiceImpl<T> implements Service<T> {

private DAOInt<T> dao;

private Class<T> cl;

public ServiceImpl(Class<T> cl, SessionFactory sessionFactory) {

this.cl=cl;

dao=new DAOImpl<T>(cl, sessionFactory);

}

@Override

public T get(Class<T> cl, int id) {

return (T) dao.get(cl, id);

}

@Override

public void save(T t) {

dao.save(t);

}

@Override

public void update(T t) {

dao.update(t);

}

@Override

public void delete(T t) {

dao.delete(t);

}

@Override

public List<T> query(String from){

return (List<T>) dao.query("from "+cl.getSimpleName());

}

@Override

public T namedQuery(String query, Map<Integer, Object> params){

return (T)dao.namedQuery(query, params);

}

@Override

public List<T> createQuery(String hsql, Map<String, Object> params) {

return (List<T>)dao.createQuery(hsql, params);

}

}

Извиква методите на DAOInt.

private DAOInt<T> dao;// обект на дао интерфейс

private Class<T> cl;// темплейт клас

public ServiceImpl(Class<T> cl, SessionFactory sessionFactory)// инициализира Class<T> cl и DAOInt<T> dao

public T get(Class<T> cl, int id)// по подаденото id и клас вика функцията на dao- get(Class<T> cl, int id) и връща съответния обект

**Клас HibernateUtil**

import org.hibernate.boot.MetadataSources;

import org.hibernate.boot.registry.StandardServiceRegistry;

import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

public class HibernateUtil {

private static final SessionFactory SessionFactory;

private static final StandardServiceRegistry serviceRegistry;

static {

try {

serviceRegistry = new StandardServiceRegistryBuilder()

.configure() // configures settings from hibernate.cfg.xml

.build();

SessionFactory = new MetadataSources(serviceRegistry).buildMetadata().buildSessionFactory();

} catch (Throwable ex) {

throw new ExceptionInInitializerError(ex);

}

}

public static SessionFactory getSessionFactory() {

return SessionFactory;

}

public static Session getSession() throws HibernateException {

return SessionFactory.openSession();

}

}

Този клас помага за работата с Hibernate. Той взима настройките от hibernate.cfg.xml файла и създава SessionFactory. От този клас чрез метода getSessionFactory() взимаме

SessionFactory за да работим със Service слоя, който от своя страна работи с DAO за да можем да извършваме операции с базата данни.

Формата за вход в системата е реализирана чрез signIn.fxml файл , който съдържа бутони за вход и регистрация и полета за имейл и парола. За да работи е създаден клас

SignInController.

Класът съдържа @FXML анотции на полетата и бутоните в fxml файла

public static User saveUser;// обект от класа User за запазване на потребителя който влиза в системата

private Logger LOGGER=LogManager.getLogger(SignInController.class);// за събитията в системата

private TextField tf\_email\_in; // поле за имейл

private PasswordField tf\_pass\_in;// поле за парола

void onSignINClick(ActionEvent event)// функция при натискане на бутона за вход , която проверява дали имейла и паролата са попълнени и ако не се извежда Alert прозорец. Ако са попълнени създава обект userService от тип Service интерфейс, който се инициализира с обект от ServiceImpl, чийто параметри са класа User и функцията на HibernateUtil класа, която взима SessionFactory. Създаден е един Map<Integer, Object> за ключ 0 се добавя имейла, а за 1 парола. userService вика функцията си за namedQuery, която взима заявката и параметрите и проверява в базата данни дали съществува такъв потребител и дава резултатите на user and saveUser /тип User/. Проверява се дали user е различен от null и ако е различен се отваря прозорец според ролята като текущия се скрива. Ако null тогава се извежда съобщение , че потребителя не съществува.

private void loadWindow(String title)// тази функция използваме при void onSignINClick(ActionEvent event) , когато отваряме прозорците според ролята като подаваме името на fxml файла.

void onSingUp(ActionEvent event)// функцията за бутона за регистрация скрива текущия прозорец и отваря формата за регистрация като задава App.setControllerRole("reader"); , която ще задава ролята на на потребителя при регистрация /тук може да се регистрират само читатели/

import javafx.event.ActionEvent;

public class SignInController {

public static User saveUser;

private Logger LOGGER=LogManager.getLogger(SignInController.class);

@FXML

private TextField tf\_email\_in;

@FXML

private PasswordField tf\_pass\_in;

@FXML

private Button btn\_signIN;

@FXML

void onSignINClick(ActionEvent event) {

String email=tf\_email\_in.getText();

String pass=tf\_pass\_in.getText();

if(email.isEmpty() || pass.isEmpty())

{

Alert alert=new Alert(AlertType.ERROR);

alert.setHeaderText(null);

alert.setContentText("Fill all required fields!");

alert.showAndWait();

}

else {

Service<User> userService = new ServiceImpl<User>(User.class, HibernateUtil.getSessionFactory());

Map<Integer, Object> data = new HashMap<Integer, Object>();

data.put(0, email);

data.put(1, pass);

User user=saveUser=userService.namedQuery("User.byEmailAndPass", data);

if(user!=null)

{

if(user.getUserRole().getRoleName().equals("admin"))

{

LOGGER.info("User: "+ user.getUserName()+" log in the system!");

((Node)event.getSource()).getScene().getWindow().hide();

loadWindow("admin");

}

else if(user.getUserRole().getRoleName().equals("operator")){

LOGGER.info("User: "+ user.getUserName()+" log in the system!");

((Node)event.getSource()).getScene().getWindow().hide();

loadWindow("operator");

}

else if(user.getUserRole().getRoleName().equals("reader")){

LOGGER.info("User: "+ user.getUserName()+" log in the system!");

((Node)event.getSource()).getScene().getWindow().hide();

loadWindow("reader");

}

}

else {

Alert alert=new Alert(AlertType.ERROR);

alert.setHeaderText(null);

alert.setContentText("Incorrect email or password!");

alert.showAndWait();

}

}

}

private void loadWindow(String title) {

Parent root = null;

try {

root= FXMLLoader.load(getClass().getResource(title+".fxml"));

} catch (IOException e) {

e.printStackTrace();

}

Scene scene=new Scene(root);

Stage stage=new Stage(StageStyle.DECORATED);

stage.setScene(scene);

stage.show();

}

@FXML

void onSingUp(ActionEvent event) {

App.setControllerRole("reader");

((Node)event.getSource()).getScene().getWindow().hide();

loadWindow("signup");

}

}

Формите за регистрация са реализирани с два fxml файла signup.fxml & saveUsers.fxml .

Първия е при натискане на бутона за регистрация от формата за вход, а другия използваме при регистриране на потребители от администратор или оператор. И двата файла имат един и същ управляващ клас контролер SignUpController.

Класът съдържа анотации за полетата/лейбъл,име,имейл,телефон,парола/ и операциите/регистрация, връщане към вход/ на бутоните още съдържа операция за проверка на роля и потребител. Създадено е и статично поле тип лейбъл **public** **static** Label *static\_label*; , което служи за променяне на текста на **private** Label lb\_register; според това какъв потребител се регистрира /оператор или читател/ . Също така класът имплементира Intitalizabel , в чиято функция **public** **void** initialize(URL location, ResourceBundle resources) задаваме лейбъла.

**void** SingUp(ActionEvent event) // бутон за запис на данните в базата създава обекти от тип Service интерфейса за User и Role /userService & roleSer/. Създава порменливи тип стринг в които поставя текста от полетата на fxml. Проверява дали потребителското име, телефона, паролата са със необходимата дължина и ако не са извежда Alert. Ако са с необходимата дължина проверява какво връща функцията checkUser(userService, name, email) ако връща true/ ако не се активира прозорец , че съществува такъв потребител/ се създават обекти тип User /подаваме имейла ,паролата ,телефона и потребителското име/ и Role/подаваме резултата от изпълнението на checkRole(roleSer) / . Взимаме днешна дата и „сетваме“ потребителската дата на регистрация. Проверяваме дали ролята е различна от null ако да „сетваме“ и ролята , ако не създаваме роля и така я добавяме на потребителския обект. Сейфаме потребителя в базата.

**private** **boolean** checkUser(Service<User> userService,String name,String email) // проверява дали имейла и са на потребител базата данни ако да връща true , ако не връща false

**private** Role checkRole(Service<Role> roleSer) //ако ролята съществува в базата връща Role ,ако не връща null

**void** goToLogIn(ActionEvent event) // връщане към форма за фход

public class SingUpController implements Initializable{

private CommonFunctions windowFunction;

private Logger LOGGER=LogManager.getLogger(SingUpController.class);

@FXML

private Label lb\_register;

public static Label static\_label;

@FXML

private TextField tf\_name;

@FXML

private TextField tf\_email\_up;

@FXML

private TextField tf\_phone;

@FXML

private PasswordField pf\_pass\_up;

@FXML

void SingUp(ActionEvent event) {

windowFunction=new CommonFunctionsImpl();

Service<User> userService = new ServiceImpl<User>(User.class, HibernateUtil.getSessionFactory());

Service<Role> roleSer = new ServiceImpl<Role>(Role.class, HibernateUtil.getSessionFactory());

String name=tf\_name.getText();

String email=tf\_email\_up.getText();

String pass=pf\_pass\_up.getText();

String phone=tf\_phone.getText();

if(name.length() <6 || email.isEmpty() ||

phone.length()<10 || pass.length()<6)

{

Alert alert=new Alert(AlertType.ERROR);

alert.setHeaderText(null);

alert.setContentText("Fill all required fields!Username and password -6 symbols,phone must be 10 numbers!");

alert.showAndWait();

}

else {

if(checkUser(userService, name, email)==true)

{

User user=new User(name,email,phone,pass,null);

Role role=new Role();

role=checkRole(roleSer);

Date utilDate=new Date();

java.sql.Date sqlDate = new java.sql.Date(utilDate.getTime());

user.setRegistrationDate(sqlDate);

if(role!=null)

{

user.setUserRole(role);

}

else {

role=new Role(App.getControllerRole());

user.setUserRole(role);

}

LOGGER.info("User: "+ user.getUserName()+" registered in DB!");

windowFunction.alertForSearchUser("Registartion succsessfuly!");

userService.save(user);

}

else {

Alert alert=new Alert(AlertType.ERROR);

alert.setHeaderText(null);

windowFunction.alertForSearchUser("Registartion unsuccsessfuly!");

alert.setContentText("This User already exists!");

alert.showAndWait();

}

}

}

private boolean checkUser(Service<User> userService,String name,String email) {

Map<Integer, Object> dataUser = new HashMap<Integer, Object>();

dataUser.put(0, name);

dataUser.put(1, email);

User user1= userService.namedQuery("User.Registration", dataUser);

if(user1==null)

{

return true;

}

else

return false;

}

private Role checkRole(Service<Role> roleSer) {

Map<Integer, Object> data = new HashMap<Integer, Object>();

data.put(0, App.getControllerRole());

Role role=roleSer.namedQuery("Role.Check", data);

if(role!=null)

{

return role;

}

else return null;

}

@FXML

void goToLogIn(ActionEvent event) {

windowFunction=new CommonFunctionsImpl();

LOGGER.info("From registration form to login!");

windowFunction.logOut(event,"\\com\\libapp\\signIn");

}

@Override

public void initialize(URL location, ResourceBundle resources) {

// TODO Auto-generated method stub

static\_label=lb\_register;

}

}

Формата за търсене на потребител в системата refUsers.fxml управляващ клас SearchUserController . Съдържа полета за име , id, дата на регистрация , имейл , таблица, за визуализиране на резултата и бутон за търсене.

**public** **void** putDataInTabel(List<User> users) //по подаден лист от потребители попълва таблицата. Проверява дали ролята на потребителя в системата е оператор/ако да , не показва информация за администраторите/.

**public** **void** init()// проверява кои полета са попълнени и ги поставя в Map<String,Object>, който ще ползваме за да създадем заявка.

**void** SearchUser(ActionEvent event) //извиква init() проверява размера на колекцията и ако е 0 издава съобщение. Ако е различно от 0 създава заявката чрез конкатенация на стрингове от стойностите на колекцията. Търси се потребителя в базата данни. Ако се намери се поставя в таблицата putDataInTabel(users);,ако не извежда съобщение

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import com.libapp.TabelViews.searchUserTabel;

import com.libapp.controllersFunctions.CommonFunctions;

import com.libapp.controllersFunctions.CommonFunctionsImpl;

import com.libapp.service.HibernateUtil;

import com.libapp.service.Service;

import com.libapp.service.ServiceImpl;

import javafx.fxml.FXML;

import javafx.scene.control.Label;

import javafx.scene.control.TableColumn;

import javafx.scene.control.TableView;

import javafx.scene.control.TextField;

import javafx.event.ActionEvent;

import javafx.collections.ObservableList;

import javafx.collections.FXCollections;

import javafx.beans.property.SimpleIntegerProperty;

import javafx.beans.property.SimpleStringProperty;

import javafx.scene.control.cell.PropertyValueFactory;

public class SearchUserController {

ObservableList<searchUserTabel> search = FXCollections.observableArrayList();

@FXML

private Label lb\_register;

@FXML

private TextField userName;

@FXML

private TextField userId;

@FXML

private TextField registrationDate;

@FXML

private TextField userEmail;

@FXML

private TableView<searchUserTabel> tb\_SearchUser;

@FXML

private TableColumn<searchUserTabel, Integer> search\_id;

@FXML

private TableColumn<searchUserTabel, String> search\_userName;

@FXML

private TableColumn<searchUserTabel, String> searchEmail;

@FXML

private TableColumn<searchUserTabel, String> search\_Phone;

@FXML

private TableColumn<searchUserTabel, String> search\_Password;

@FXML

private TableColumn<searchUserTabel, String> search\_RegDate;

@FXML

private TableColumn<searchUserTabel, String> search\_Role;

/\* @FXML

private TableColumn<searchUserTabel, String> search\_Books;\*/

private Map<String, Object>queryMap=new HashMap<>();

private CommonFunctions windowFunction;

@FXML

void SearchUser(ActionEvent event) {

windowFunction=new CommonFunctionsImpl();

init();

if(queryMap.size()==0)

{

windowFunction.alertForSearchUser("Fill at least one fild!");

}

else

{

Service<User> userService = new ServiceImpl<User>(User.class, HibernateUtil.getSessionFactory());

List<User> users=new ArrayList<>();

String criteriaString="";

int br=0;

for (String i : queryMap.keySet()) {

if(br==0)

{

criteriaString=i+" LIKE '%' || :"+i+" || '%'";

br++;

}

else {

criteriaString=criteriaString+" and "+i+" LIKE '%' || :"+i+" || '%'";

}

}

users=userService.createQuery("from User where "+criteriaString, queryMap);

if(users.size()==0)

{

windowFunction.alertForSearchUser("No such user!");

}

putDataInTabel(users);

}

}

public void putDataInTabel(List<User> users) {

int i=0;

while(i<users.size())

{

Role role=new Role();

role=users.get(i).getUserRole();

if((SignInController.saveUser.getUserRole().getRoleName()).equals("operator") && (role.getRoleName()).equals("admin"))

{

i++;

}

else

{

search.add(new searchUserTabel(

new SimpleIntegerProperty(users.get(i).getUserId()),

new SimpleStringProperty(users.get(i).getUserName()),

new SimpleStringProperty(users.get(i).getUserEmail()),

new SimpleStringProperty(users.get(i).getUserPhone()),

new SimpleStringProperty(users.get(i).getUserPass()),

new SimpleStringProperty(role.getRoleName()),

new SimpleStringProperty(users.get(i).getRegistrationDate().toString())

));

}

i++;

}

search\_id.setCellValueFactory(new PropertyValueFactory<>("Id"));

search\_userName.setCellValueFactory(new PropertyValueFactory<>("userName"));

searchEmail.setCellValueFactory(new PropertyValueFactory<>("userEmail"));

search\_Phone.setCellValueFactory(new PropertyValueFactory<>("userPhone"));

search\_Password.setCellValueFactory(new PropertyValueFactory<>("userPass"));

search\_RegDate.setCellValueFactory(new PropertyValueFactory<>("registrationDate"));

search\_Role.setCellValueFactory(new PropertyValueFactory<>("userRole"));

//search\_Books.setCellValueFactory(new PropertyValueFactory<>("userBooks"));

tb\_SearchUser.setItems(search);

}

public void init() {

queryMap=new HashMap<>();

if(!userId.getText().trim().isEmpty())

{

queryMap.put("userId", userId.getText());

}

if(!userName.getText().trim().isEmpty())

{

queryMap.put("userName", userName.getText());

}

if(!userEmail.getText().trim().isEmpty())

{

queryMap.put("userEmail", userEmail.getText());

}

if(!registrationDate.getText().trim().isEmpty())

{

queryMap.put("registrationDate", registrationDate.getText());

}

userId.clear();

userEmail.clear();

userName.clear();

registrationDate.clear();

//return queryMap;

}

}

Добавяне на книга става чрез bookAdd.fxml клас контролер- AddBookController

Полета за заглавие, копия, статус и комбо боксове за жанр автор и издател. Бутон за запис.

**void** SelectGenre(ActionEvent event) //избор на жанр от комбо бокса и запис в променлива тип Стринг която задаваме като стойност на колекция , чрез която задаваме параметри ни заявка от базата данни за съответния жанр

**void** SelectAuthor(ActionEvent event)

**void** SelectPublisher(ActionEvent event)

**public** **void** initialize(URL location, ResourceBundle resources) // класът имплементира методите на Initalizable . В този метод се инициализират комбо боксовете.

**void** SaveBook(ActionEvent event) // функция на бутона за търсене . Проверка на дължините на въведените данни в полетата за заглавие, копия и статус . Обект от клас book се инициализира с данните от попълнените полета и комбо боксове и ако книгата не съществува в базата се запазва

import javafx.fxml.FXML;

import javafx.fxml.Initializable;

import javafx.scene.control.ComboBox;

import java.net.URL;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.ResourceBundle;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import javafx.scene.control.TextField;

import com.libapp.controllersFunctions.CommonFunctions;

import com.libapp.controllersFunctions.CommonFunctionsImpl;

import com.libapp.service.HibernateUtil;

import com.libapp.service.Service;

import com.libapp.service.ServiceImpl;

import javafx.collections.FXCollections;

import javafx.collections.ObservableList;

import javafx.event.ActionEvent;

public class AddBookController implements Initializable {

private Logger LOGGER=LogManager.getLogger(SignInController.class);

@FXML

private TextField tx\_title;

@FXML

private TextField tx\_Copies;

@FXML

private TextField tx\_Status;

@FXML

private ComboBox cb\_genre;

ObservableList<String> searchGenre = FXCollections.observableArrayList();

ObservableList<String> searchAuthor = FXCollections.observableArrayList();

ObservableList<String> searchPublisher = FXCollections.observableArrayList();

@FXML

private ComboBox cb\_author;

@FXML

private ComboBox cb\_Publisher;

private String genreString=null;

private String authorString=null;

private String publisherString=null;

private Genre genre=null;

private Author author=null;

private Publisher publisher=null;

Service<Genre> genreService = new ServiceImpl<Genre>(Genre.class, HibernateUtil.getSessionFactory());

Service<Author> authorService = new ServiceImpl<Author>(Author.class, HibernateUtil.getSessionFactory());

Service<Publisher> publisherService = new ServiceImpl<Publisher>(Publisher.class, HibernateUtil.getSessionFactory());

Service<Book> bookService = new ServiceImpl<Book>(Book.class, HibernateUtil.getSessionFactory());

Map<Integer, Object> data = new HashMap<Integer, Object>();

private CommonFunctions windowFunction;

@FXML

void SelectGenre(ActionEvent event) {

genreString=(String) cb\_genre.getValue();

data.put(0, genreString);

genre=genreService.namedQuery("Genre.getByName", data);

data.clear();

}

@FXML

void SaveBook(ActionEvent event) {

windowFunction=new CommonFunctionsImpl();

if(tx\_title.getText().length()<2||tx\_Copies.getText().length() <1 || tx\_Status.getText().length()<4 || genre==null || author==null || publisher==null)

{

windowFunction.alertForSearchUser("Fill all filds!");

}

else {

User user=SignInController.saveUser;

Book book=new Book(tx\_title.getText(),genre,publisher,Integer.parseInt(tx\_Copies.getText()),tx\_Status.getText(),user);

data.put(0,tx\_title.getText());

if(bookService.namedQuery("Book.getByTitle", data)==null)

{

bookService.save(book);

windowFunction.alertForSearchUser("Save succsessfuly!");

LOGGER.info("User: "+ user.getUserName()+" save book in the DB!"+" Book "+book.getBookTitle());

}

else {

LOGGER.info("User: "+ user.getUserName()+" try to save in the DB but the book already exists!"+" Book "+book.getBookTitle());

windowFunction.alertForSearchUser("We already have this book!");

}

}

data.clear();

}

@FXML

void SelectAuthor(ActionEvent event) {

authorString=(String) cb\_author.getValue();

data.put(0, authorString);

author=authorService.namedQuery("Author.getByName", data);

data.clear();

}

@FXML

void SelectPublisher(ActionEvent event) {

publisherString=(String) cb\_Publisher.getValue();

data.put(0, publisherString);

publisher=publisherService.namedQuery("Publisher.getByName", data);

data.clear();

}

@Override

public void initialize(URL location, ResourceBundle resources) {

List<Genre> genres=genreService.query(null);

for (Genre g : genres) {

searchGenre.add(g.getGenreName());

}

cb\_genre.setItems(searchGenre);

List<Author> authors=authorService.query(null);

for (Author a : authors) {

searchAuthor.add(a.getAuthorName());

}

cb\_author.setItems(searchAuthor);

List<Publisher> publishers=publisherService.query(null);

for (Publisher p : publishers) {

searchPublisher.add(p.getPublisherName());

}

cb\_Publisher.setItems(searchPublisher);

}

}

Формите за администратор, оператор и читател са сходни. В зависимост от ролята поддържат различни операции , но контролерите са сходни.

admin.fxml, operator.fxml, reader.fxml

Класове контролери AdminController, OperatorController, ReaderController

Съдържат функции на различните бутони, който викат функцията за отваряне на различния fxml файл по подадено име на файла private void loadPage(String page).

Имплементират Initializable public void initialize(URL location, ResourceBundle resources) в която се задава потребителя. Функция за излизане и връщане към вход void logOut(ActionEvent event).

public class AdminController extends SingUpController implements Initializable{

private CommonFunctions windowFunction;

private Logger LOGGER=LogManager.getLogger(AdminController.class);

@FXML

private AnchorPane ap;

@FXML

private Pane pane;

@FXML

private Label userName;

@FXML

private Label user;

@FXML

private void ScrappBook(ActionEvent event) {

loadPage("");

}

@FXML

private void addBook(ActionEvent event) {

loadPage("bookAdd");

}

@FXML

private void archiveBook(ActionEvent event) {

loadPage("");

}

@FXML

private void borrowBook(ActionEvent event) {

loadPage("");

}

@FXML

private void createOperators(ActionEvent event) {

App.setControllerRole("operator");

loadPage("saveUsers");

static\_label.setText("Operator");

}

@FXML

private void createReader(ActionEvent event) {

App.setControllerRole("reader");

loadPage("saveUsers");

static\_label.setText("Reader");

}

@FXML

void deleteReader(ActionEvent event) {

loadPage("");

}

@FXML

private void refBook(ActionEvent event) {

loadPage("");

}

@FXML

private void refForms(ActionEvent event) {

loadPage("");

}

@FXML

private void refRating(ActionEvent event) {

loadPage("");

}

@FXML

private void refUsers(ActionEvent event) {

loadPage("refUsers");

}

@FXML

private void returnBook(ActionEvent event) {

loadPage("");

}

private void loadPage(String page) {

Parent root = null;

try {

root= FXMLLoader.load(getClass().getResource(page+".fxml"));

} catch (IOException e) {

e.printStackTrace();

}

pane.getChildren().add(root);

}

@FXML

void logOut(ActionEvent event) {

windowFunction=new CommonFunctionsImpl();

LOGGER.info("User: "+userName.getText()+" logout from the system!");

windowFunction.logOut(event,"\\com\\libapp\\signIn");

}

@Override

public void initialize(URL location, ResourceBundle resources) {

// TODO Auto-generated method stub

userName.setText(SignInController.saveUser.getUserName());

user.setText(SignInController.saveUser.getUserName());

}

В проекта има още помощен интерфейс CommonFunctions и клас , който го имплементира CommonFunctionsImpl . Имат два метода

**public** **void** logOut(ActionEvent event, String page) //затваря текущия прозорец и чрез основния клас App отваря отново формата за вход.

**public** **void** alertForSearchUser(String alertText) // функция за извеждане на прозорци съобщения

* **Модул за регистриране на събития в системата**

Реализиран е чрез log4j. Конфигурационен файл за log4j – log4j2.properties. В този файл сме указали, че събитията ще се записват във файл LibraryLogs.log и освен това ще се извеждат на конзолата. В класовете си създаваме променлива от тип Logger и с нея отразяваме събитията.

name=PropertiesConfig

property.filename = logs

appenders = console, file

appender.console.type = Console

appender.console.name = STDOUT

appender.console.layout.type = PatternLayout

appender.console.layout.pattern = [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c**{1}** - %msg%n

appender.file.type = File

appender.file.name = LOGFILE

appender.file.fileName=${filename}/LibraryLogs.log

appender.file.layout.type=PatternLayout

appender.file.layout.pattern=[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c**{1}** - %msg%n

appender.file.append=true

loggers=file

logger.file.name=com.libapp

logger.file.level = info

logger.file.appenderRefs = file

logger.file.appenderRef.file.ref = LOGFILE

rootLogger.level = info

rootLogger.appenderRefs = stdout

rootLogger.appenderRef.stdout.ref = STDOUT

1. **Тестови резултати**

* Junit tests – представлява framework за тестване на парчета от код (класове или методи) за Java, който също така се използва в много други езици.
* Функционални тестове – това са тестове, които се отнасят до видимата функция за крайния потребител и как това работи по начин, който има смисъл на ниво бизнес / домейн. Това са тестове, посочени в условията на домейна, а не такива, които се отнасят до това как се изпълнява функцията. Кодът, който се изпълнява не е от значение и в идеалния случай не трябва да се проваля, когато промените начина на изпълнение на функцията, без да променяте какво прави функцията.
* Интеграционни тестове - имат за цел да се проверят взаимовръзките и съвместната работа на отделните компоненти, когато са поставени в една среда. Компонентите може да се интегрират на вълни или накуп (известно още като Големият Взрив – „big bang“). Счита се, че варианта на вълни е по-добър, защото позволява новопоявили се дефекти да се открият и отстранят по-бързо преди да се интегрира следващата вълна и дефектите са се мултиплицират.