ДОКУМЕНТАЦИЯ

**ТЕМА**: Проект „Складове“

***Изготвен от****: Калоян Димитров и Славян Христов,*

*учащи в „Технически университет“ – Варна*

Информация за студентите:

*Студент 1: Студент 2:*

*име: Славян Христов име: Калоян Димитров*

*фак. номер: 18621793 фак.номер: 18621798*

*факултет: ФИТА факултет: ФИТА*

*специалност: СИТ специалност: СИТ*

*група: 5б група: 5б*

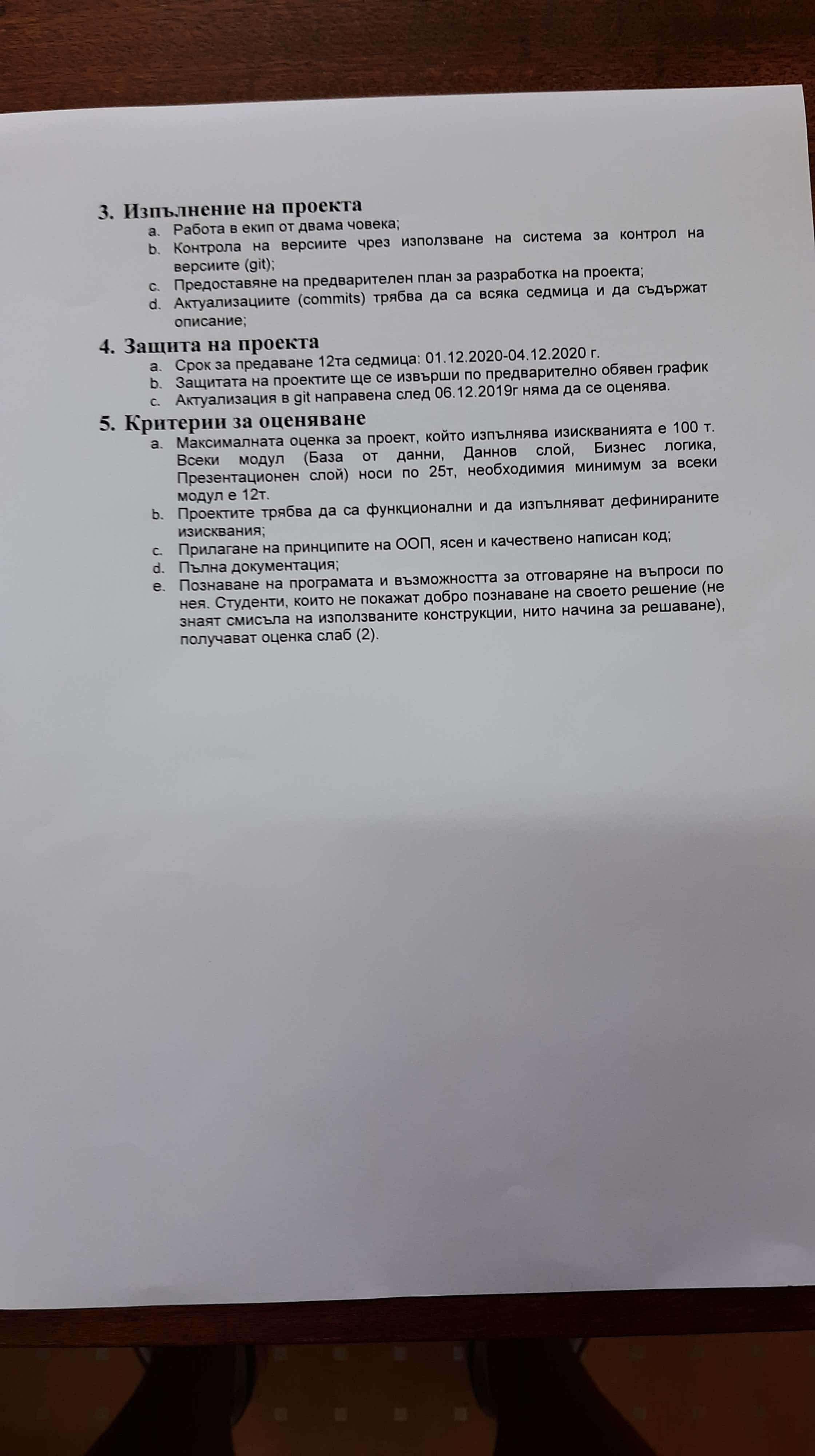
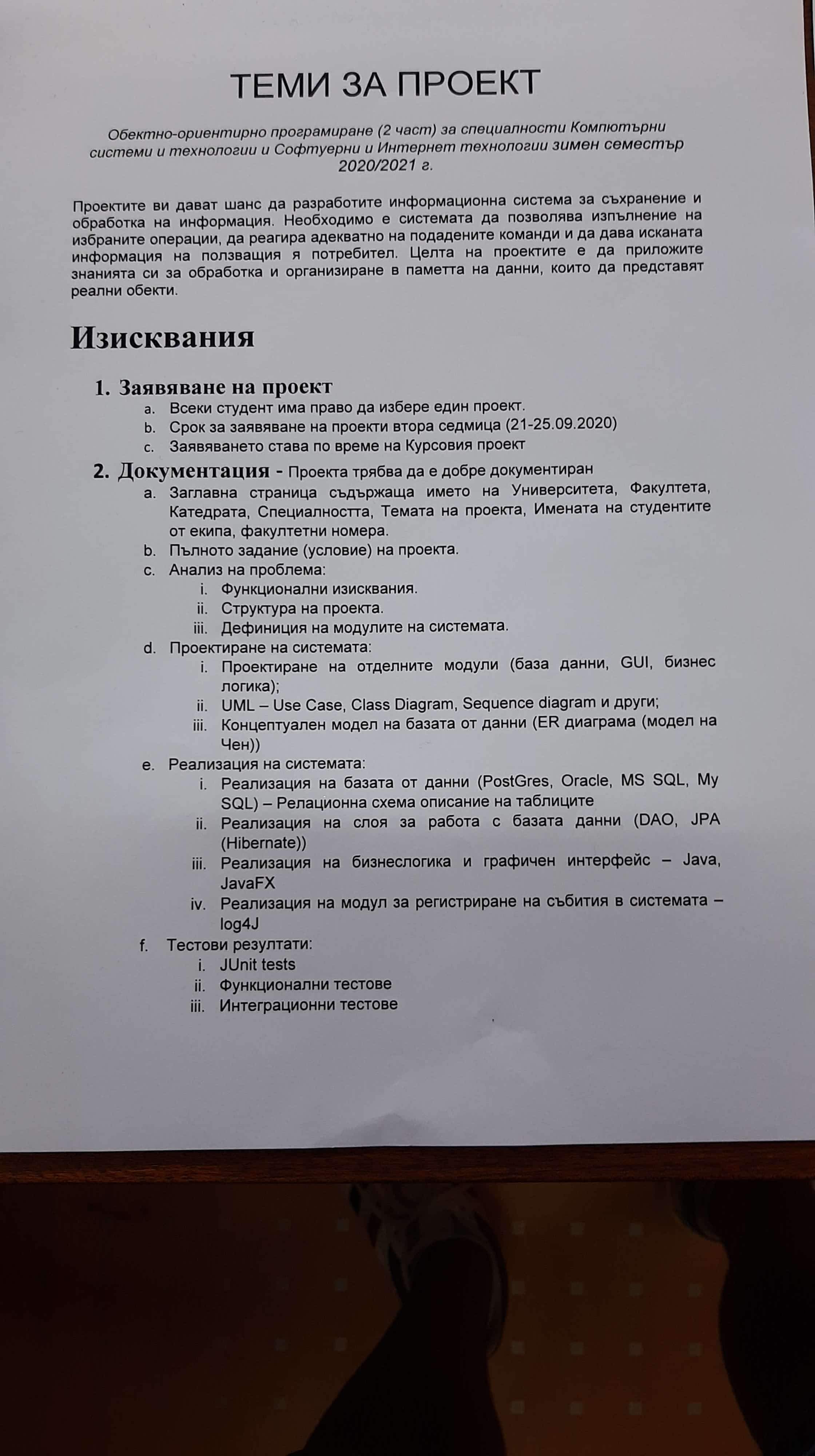
Съдържание*:*

1. *Заглавна страница*
2. *Задание*
3. *Анализ на проблема*
4. *Структура и проектиране на системата*
5. *Реализация на системата*

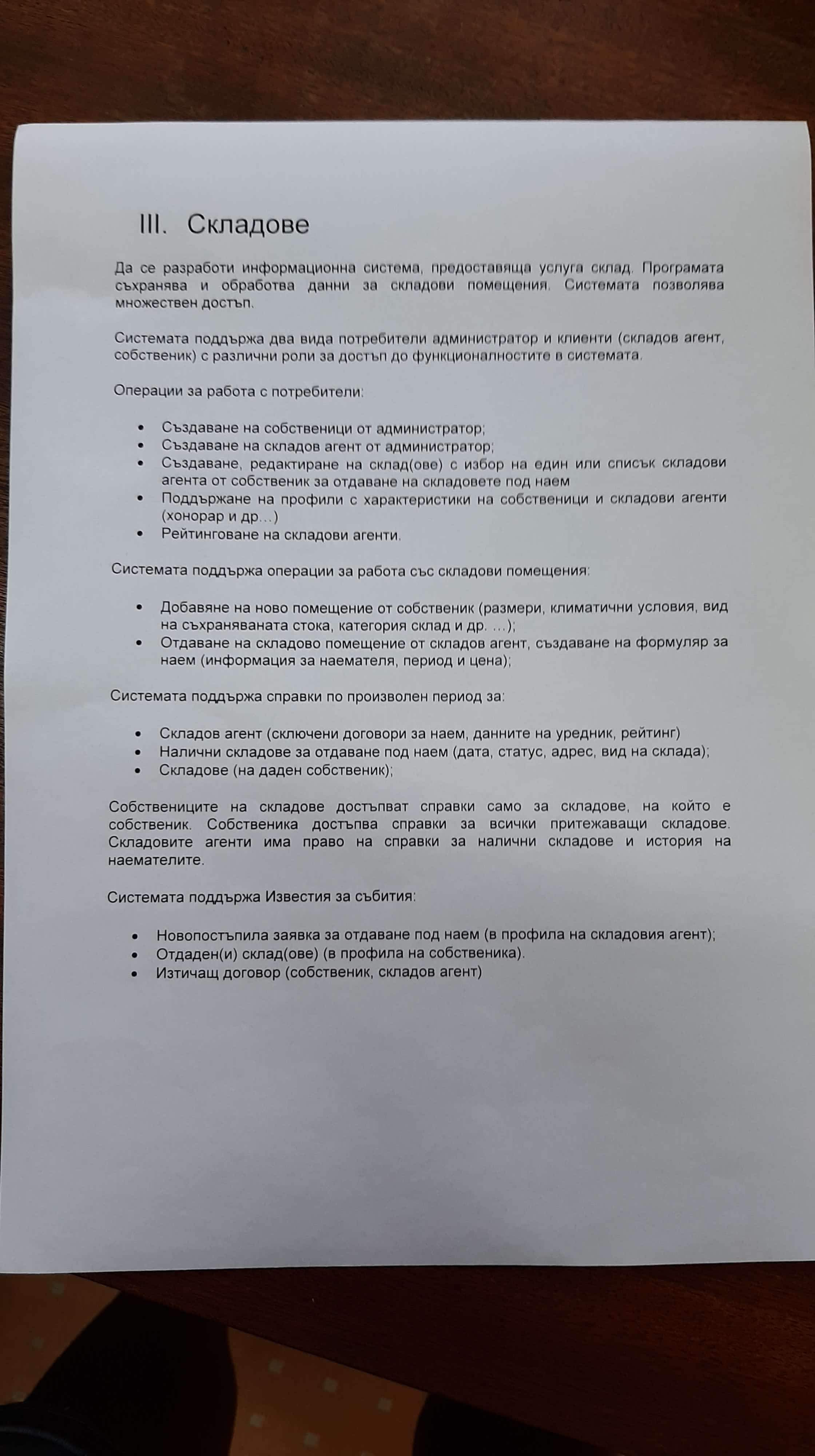
* *Потребителски интерфейс(Front-End)*
* *Работа с база данни и сървърни операции(Back-end)*

1. *Тестови резултати*
2. *Източници*

Изисквания



Задание



Анализ на проблема

1. Функционални изисквания

Функционалността на системата трябва да бъде съобразена с нуждите на потребителите, които ще я използват. Интерфейсът на системата трябва да бъде прост, приятен и да позволява лесно и безпрепятсвено използване, без грешки в процеса на работа.

Потребителите, които ще използват системата са 3 вида: администратор, собственик на склад и складов агент. Те трябва да могат да изпълняват следните действия:

1. Администратор

* влизане в системата
* създаване на собственици
* създаване на складови агенти
* преглеждане на профили(максимални права)
* редактиране на профили(максимални права)
* изтриване на профили(максимални права)

1. Собственик

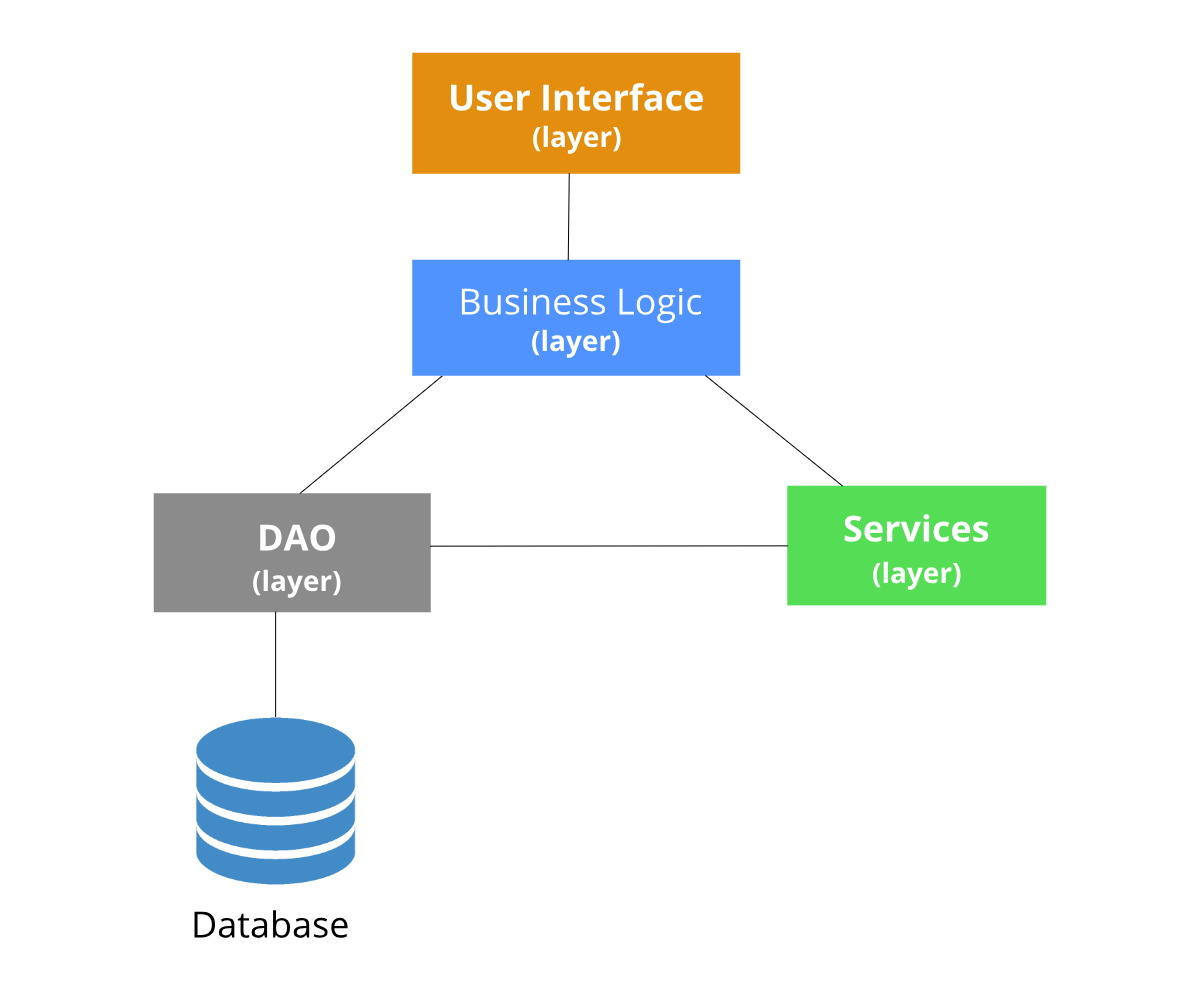
* влизане в системата
* създаване на складове
* преглеждане на собствени складове
* редактиране на собствени складове
* изтриване на собствени складове
* назначаване на складови агенти за даване под наем на складове
* достъп до справка за всички собствени складове
* получава известия за отдадени складове
* получава известия за изтичащи договори за наем
* преглеждане на собствения си профил(лимитирани права)
* редактиране на собствения си профил(лимитирани права)

1. Складов агент

* влизане в системата
* отдаване на склад под наем(договор за наем с информация, период, цена)
* притежава рейтинг
* достъп до справка за всички налични складове за даване под наем
* достъп до справка за история на наемателите(договори, дата, рейтинг)
* получава известия за искове за отдаване под наем на склад
* получава известия за изтичащи договори за наем
* преглеждане на собствения си профил(лимитирани права)
* редактиране на собствения си профил(лимитирани права)

Системата трябва да предостави възможност за изпъленение на всички действия описани горе. Допълнителни функционалности трябва да бъде имплементирани за валидиране на данните при регистрация или при актуализация на профил и за автентикация при влизане в системата.

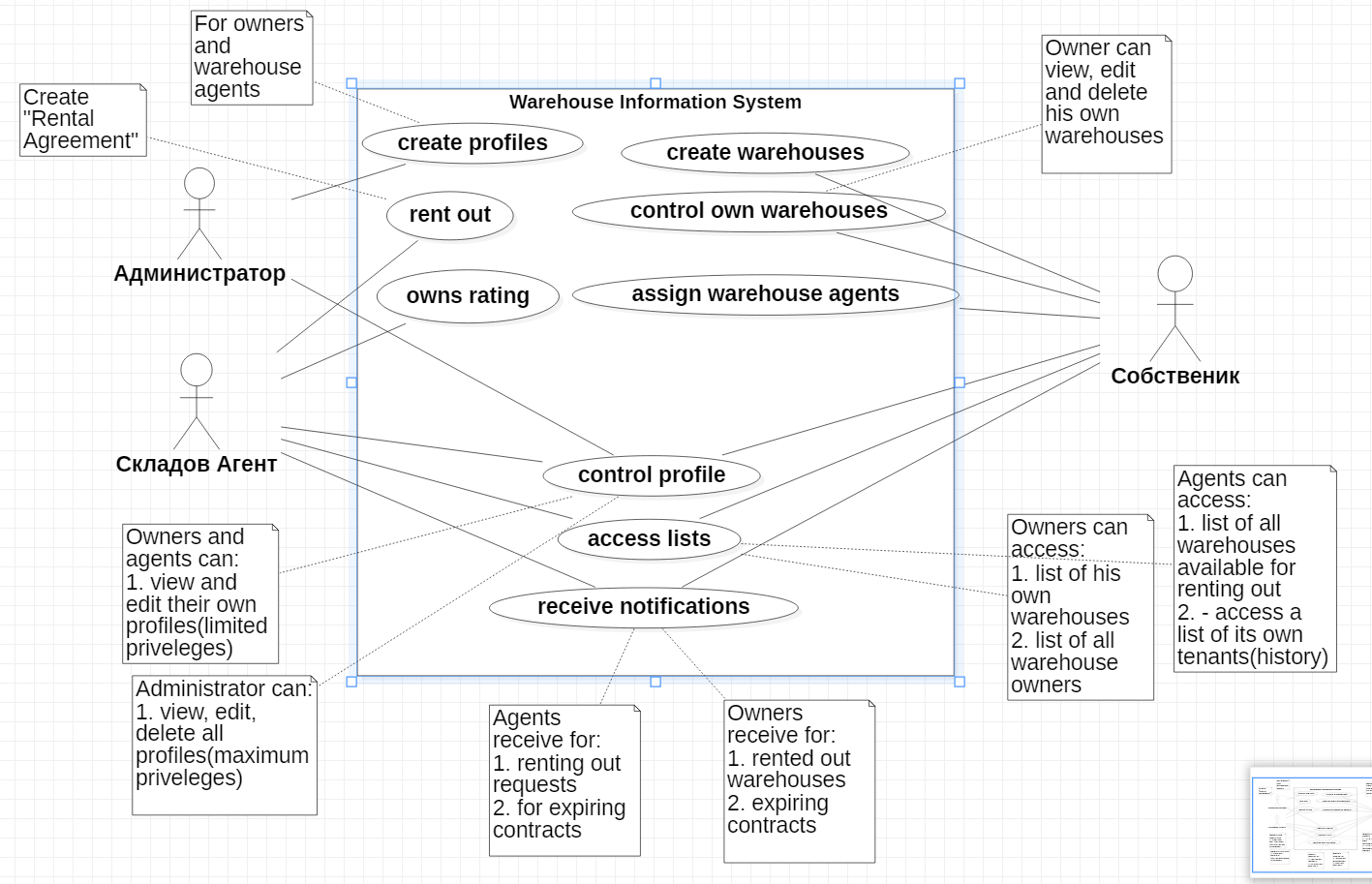
1. Структура на проекта



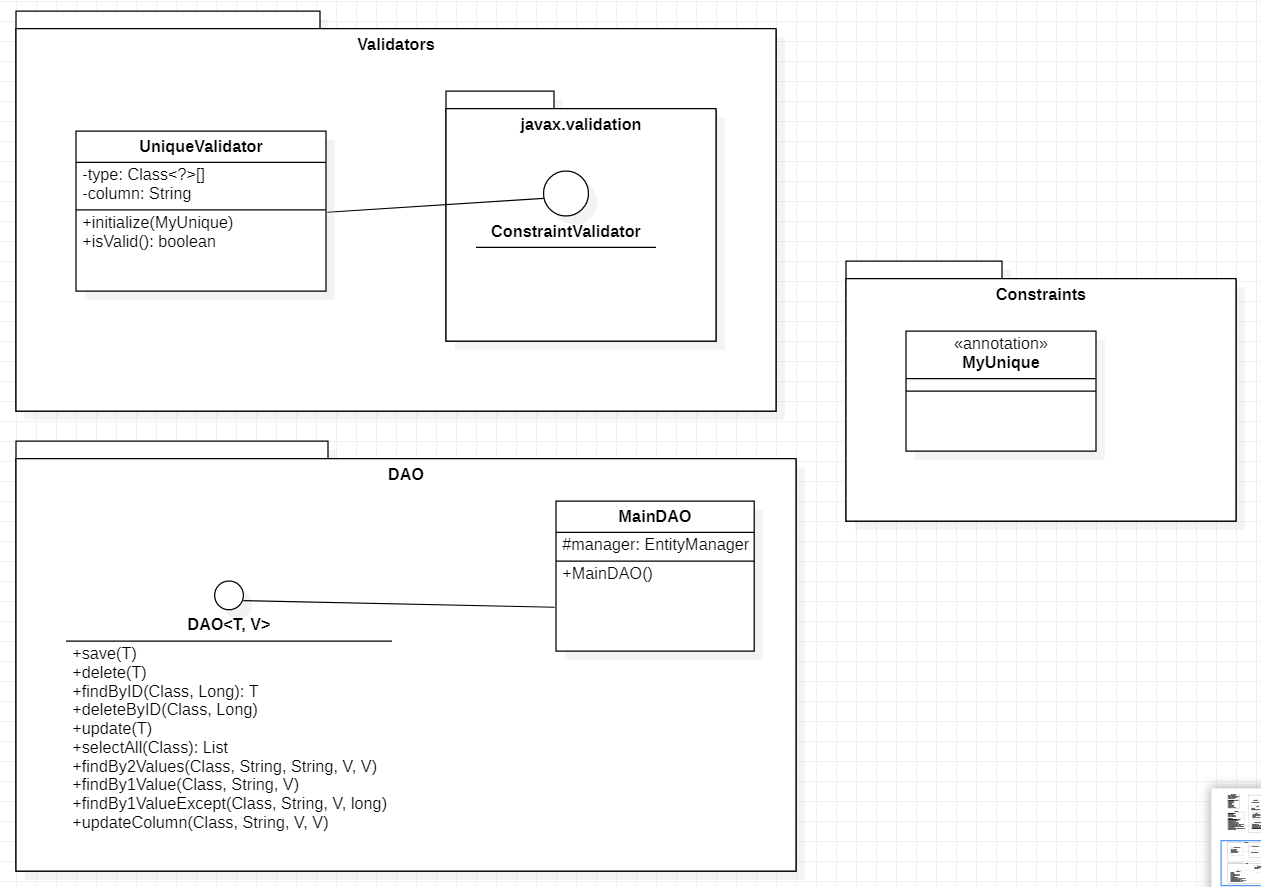
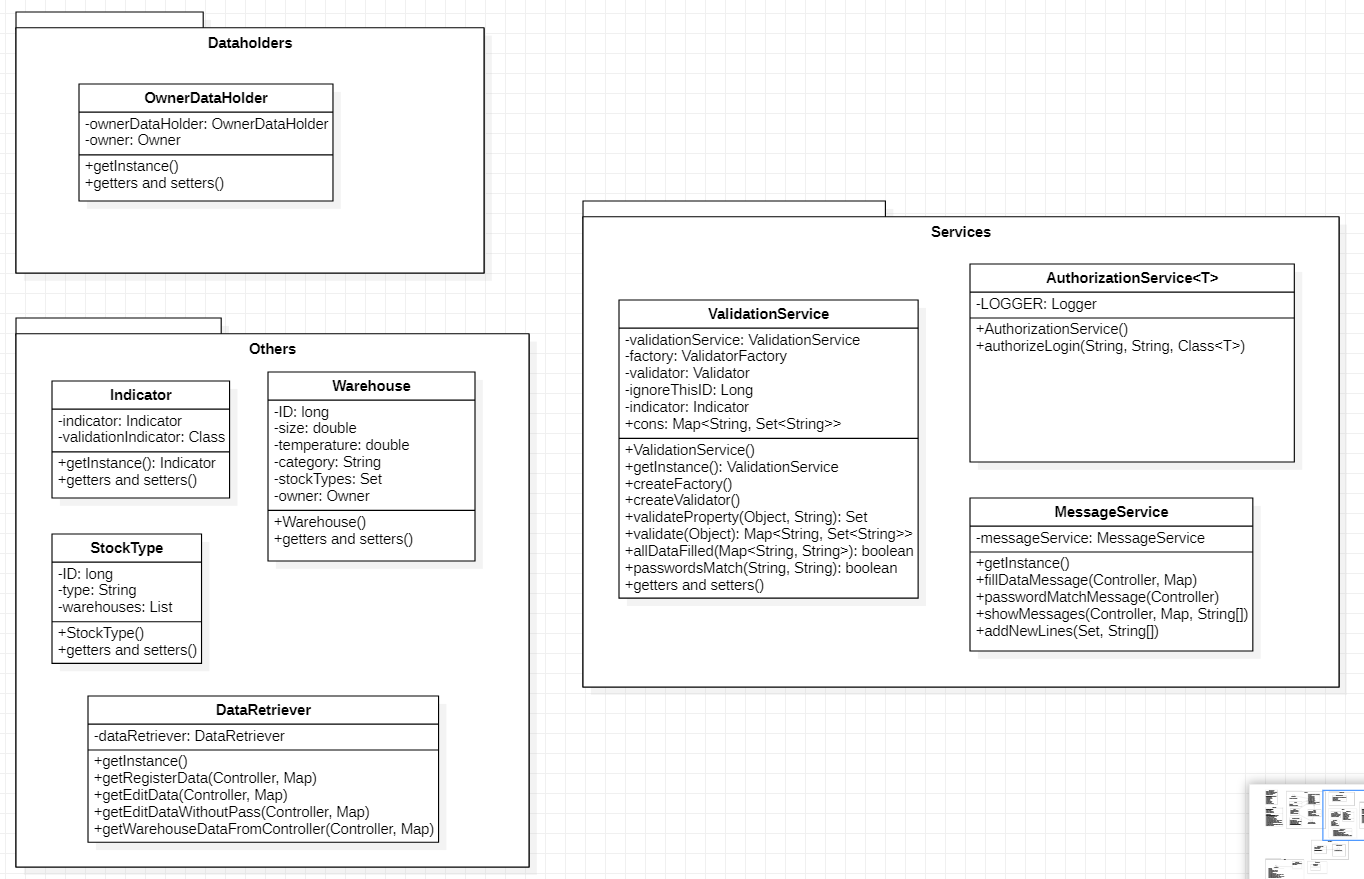
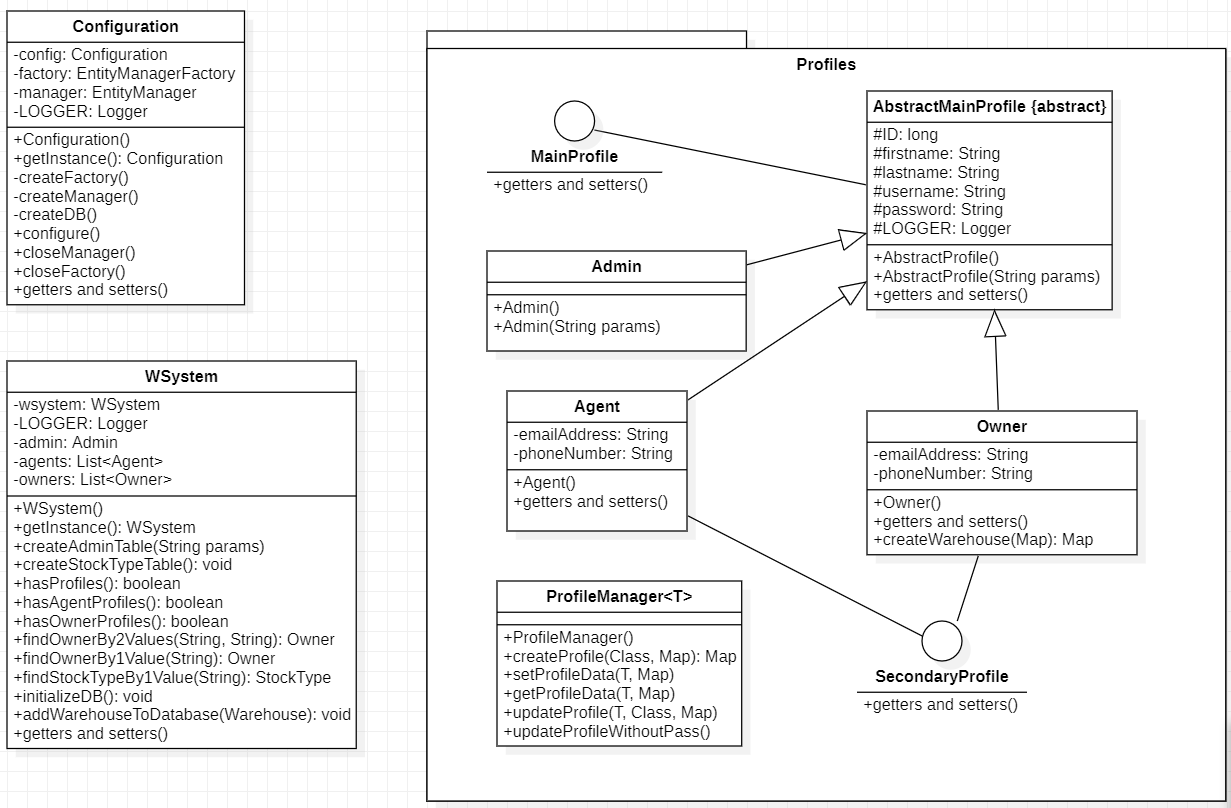
1. Дефиниция на модулите на системата
2. User Interface – Потребителски интерфейс, който служи за навигиране на потребителя в системата
3. Business Logic – Това е модулът, който служи за имплементация на бизнес идеите и правилата. Служи си с обекти от високо ниво и бизнес обекти и отразява ясна представа за работния процес на програмата.
4. Services – Това е модул от по-ниско ниво, който служи за извършване на нещо странично, което е нужно в процеса на работа
5. DAO – Тове е модул от по-ниско ниво, който служи само и единствено за извършване на операции с базата данни

Проектиране на системата

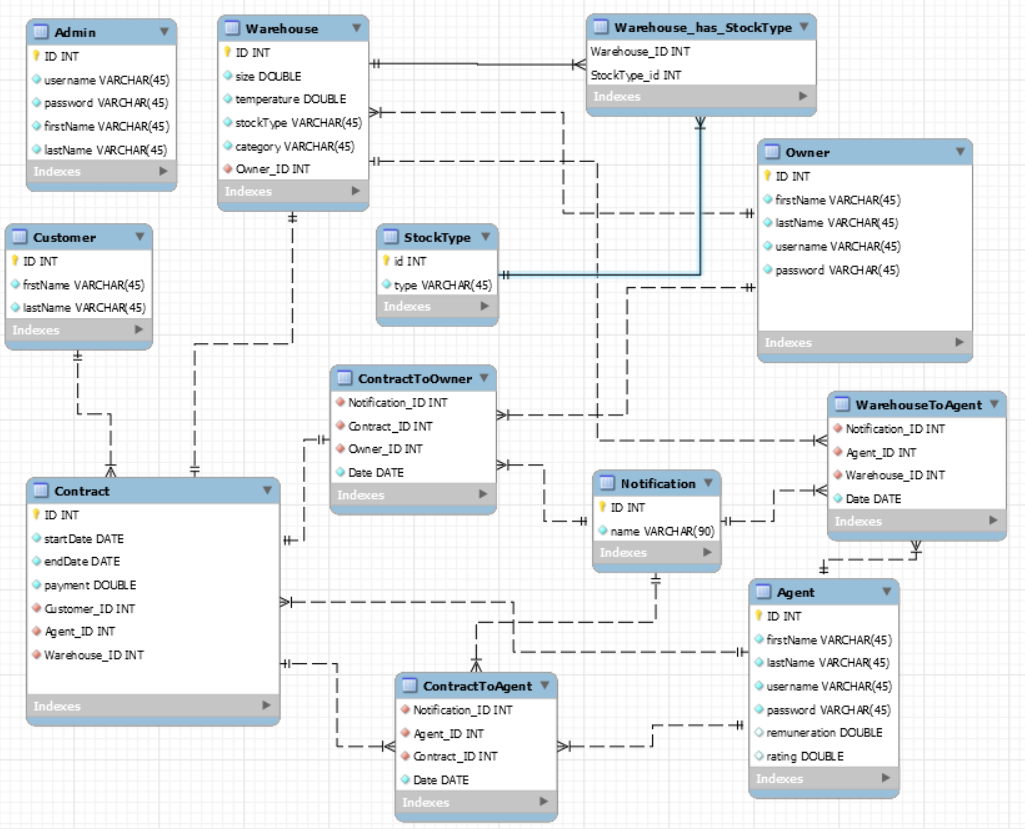
Use case Diagram



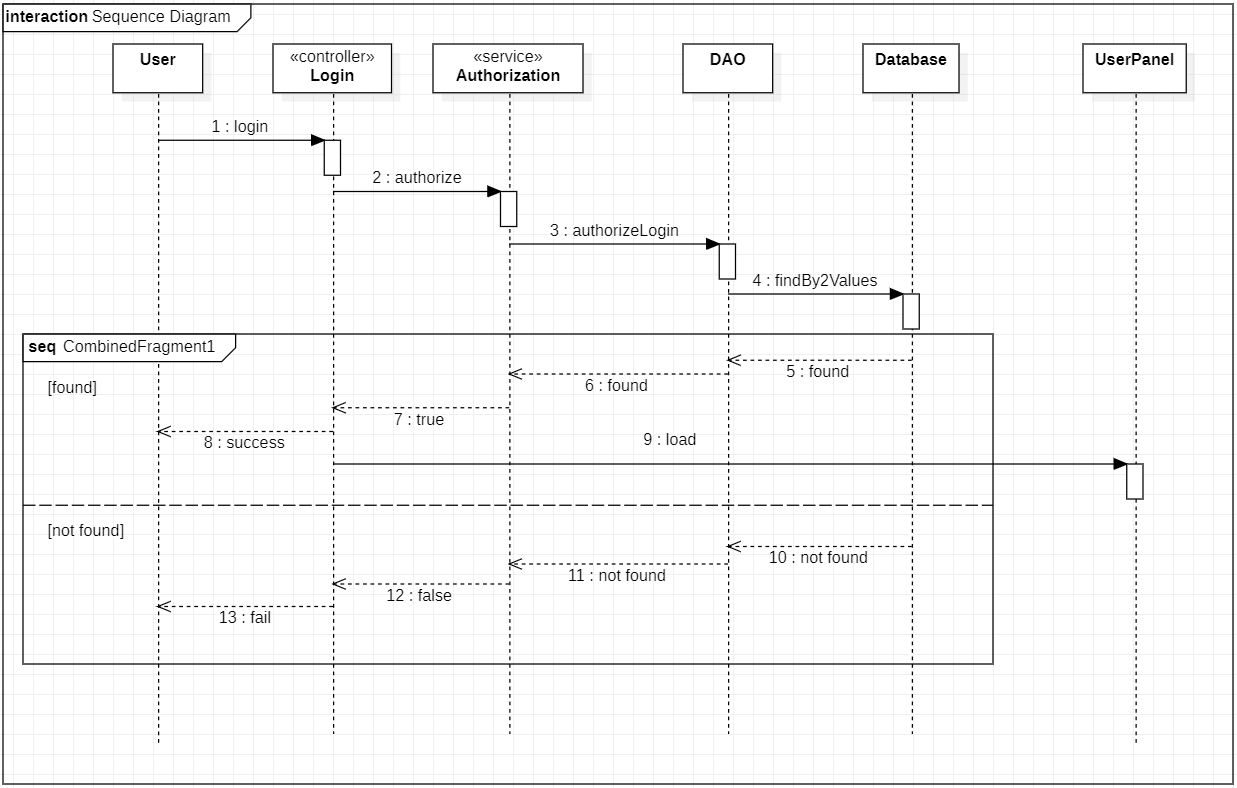
Class Diagram



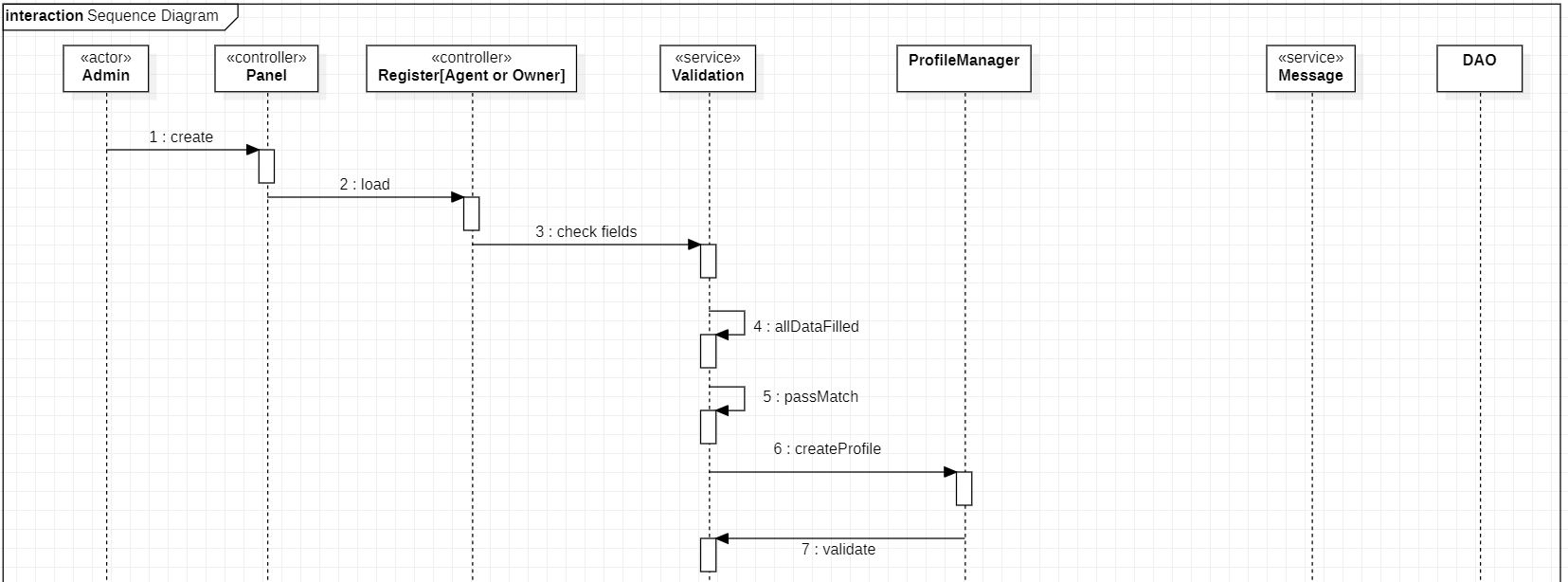
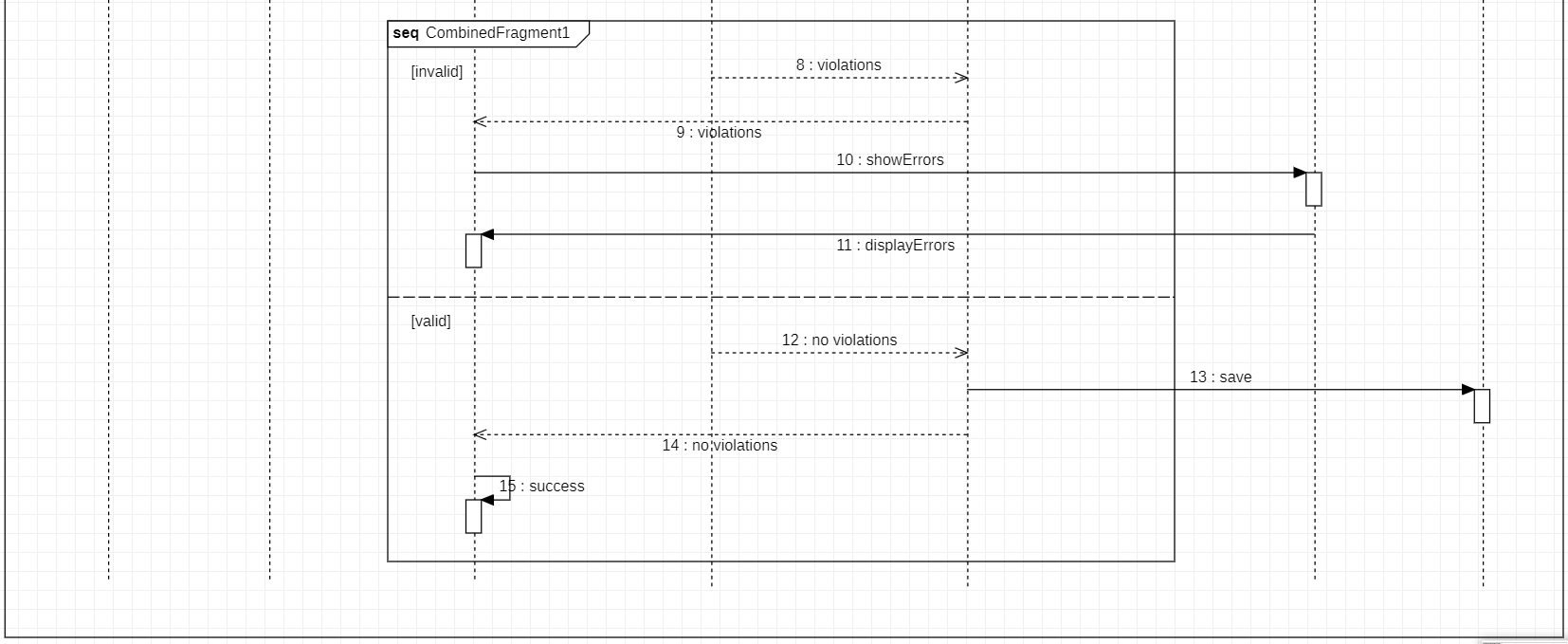
EER Diagram



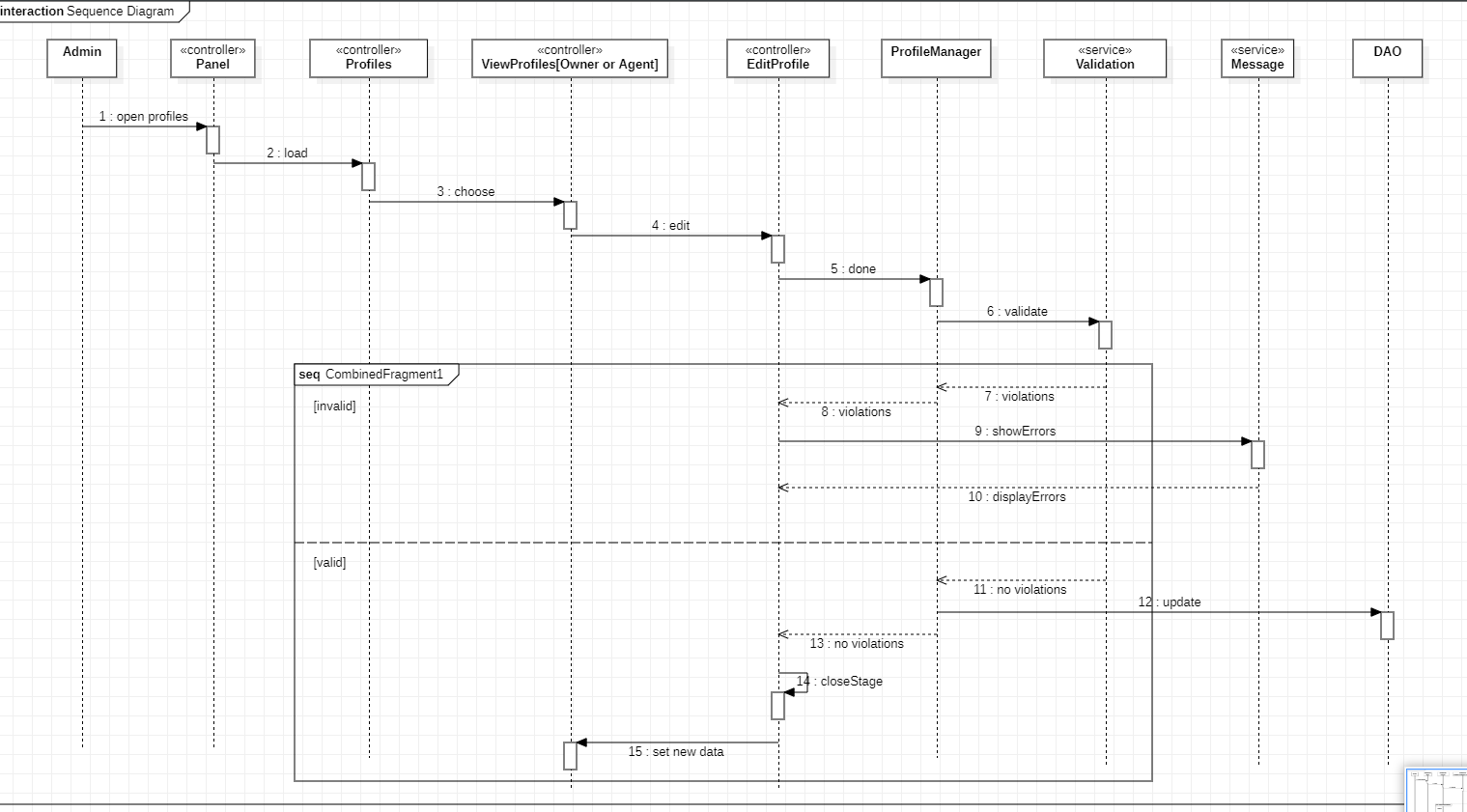
Sequence Diagram for Login Scenario



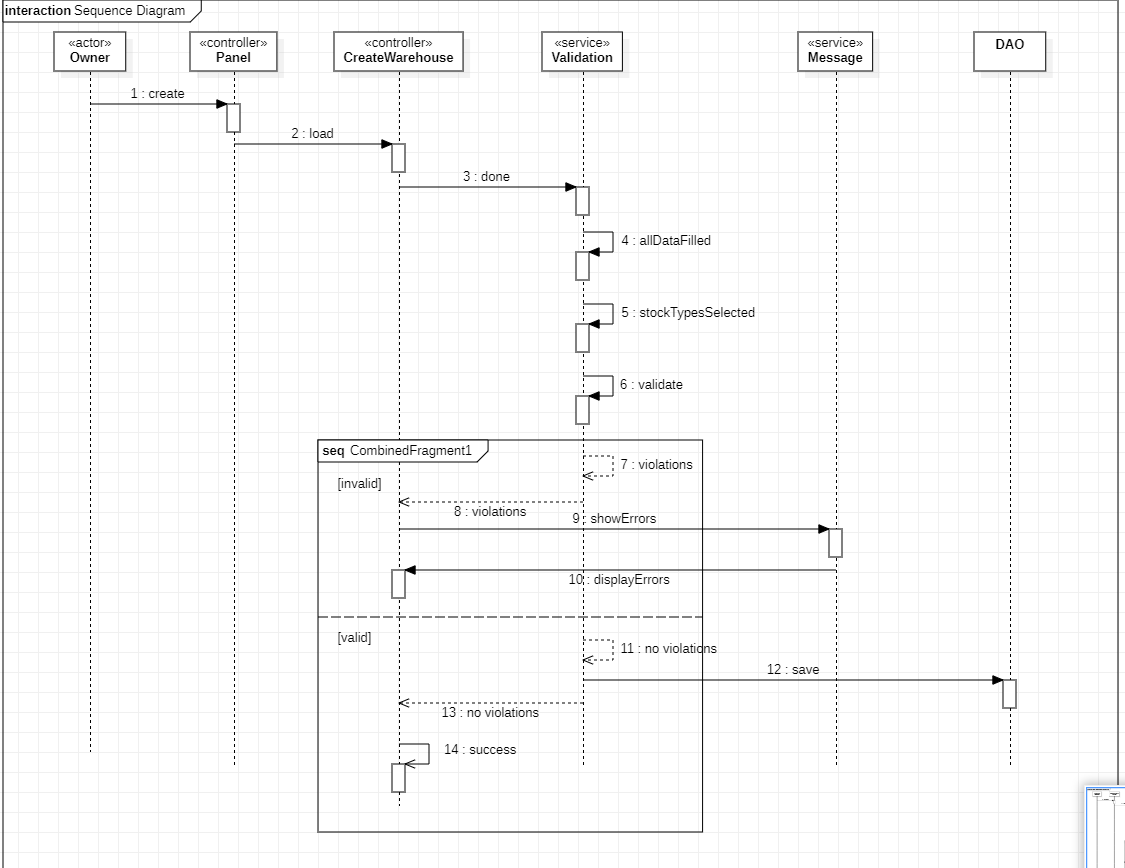
Sequence Diagram for Register User Scenario



Sequence Diagram for Admin Update Profiles Scenario

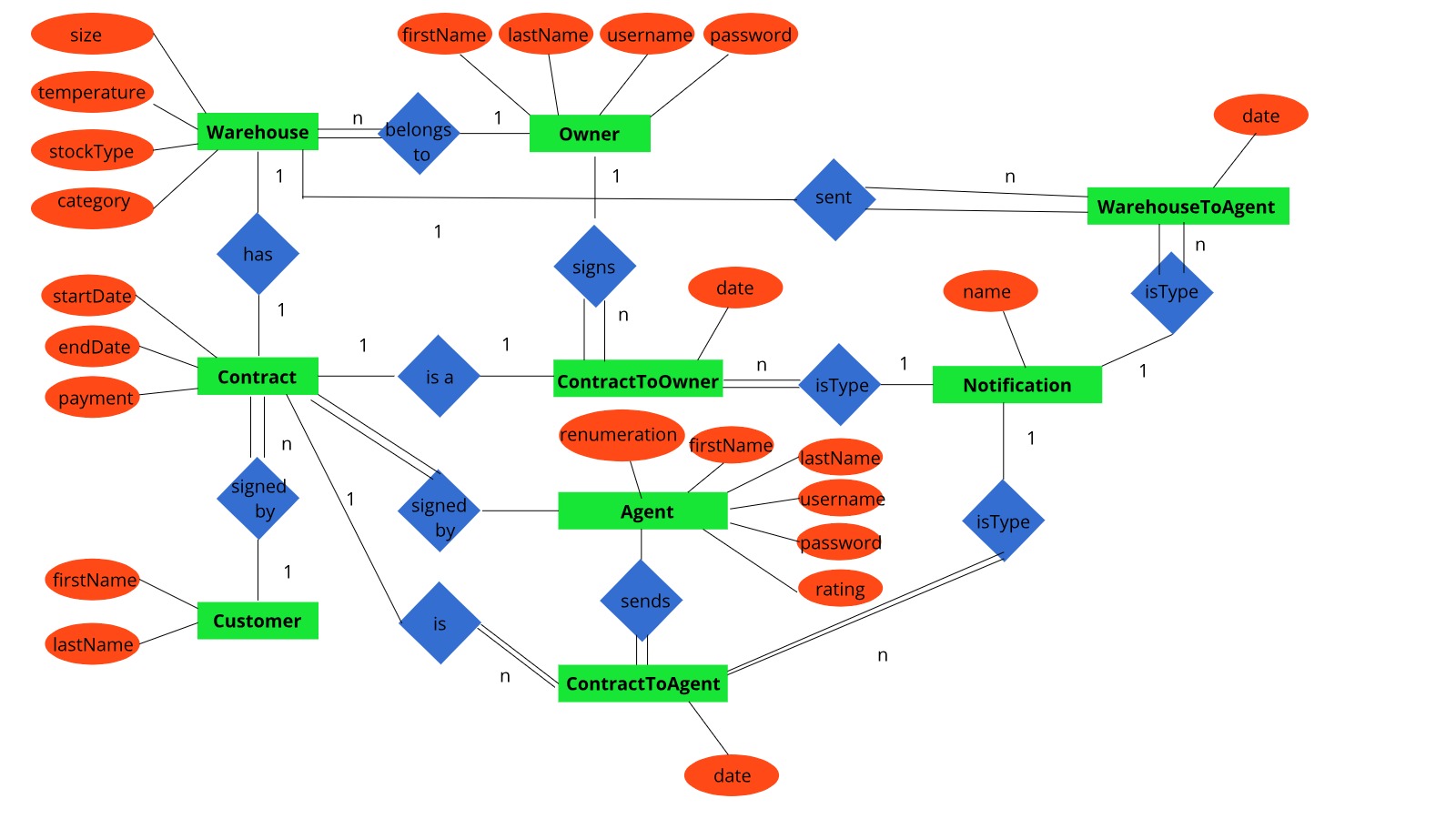


Sequence Diagram for Create Warehouse Scenario



Някои сценарии не са включени в документацията.

Модел на Чен  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Реализация на системата



1. **Използвана технология**
2. JavaFX – Реализация на потребителски и графичен интерфейс
3. JPA + Hibernate – Реализация на слоя за работа с база данни
4. MySQL database – база данни
5. Java – използван език
6. Log4J2 – за регистриране на събитията
7. JUnit Jupiter – за тестови резултати
8. **Потребителски интерфейс**

Потребителският интерфейс е изграден с помоща на ***JavaFX 15.***

***JavaFX*** може да се добави към проекта като първо се изтегли библиотеката от интернет и след това в IDE-то, което се използва(в нашия случай IntelliJ) се добавя конфигурация. Като се отвори прозореца за добавяне на конфигурация в текстовото поле за VM Options се добавя следния ред:

--module-path ***"\path\to\javafx-sdk-15.0.1\lib****"* --add-modules javafx.controls,javafx.fxml

Като това, което е в кавички е пътя към самата папката, която съдържа базовите библиотеки за JavaFX.

Също библиотеката трябва да се добави и в *Project Structure -> Libraries -> избира се папката с библиотеките на JavaFX.*

Използвани са .fxml файлове, които значително улесняват продуктивността и реализацията на Front-End частта от проекта. Тези файлове автоматично се попълват с генериран код, който код е на език XML и може да се управлява с помоща на “Scene Builder”. Това е средство, което спомага редактирането, управлението на дизайна на един графичен потребителски интерфейс.

За да се добави Scene Builder към IDE-то, първо се изтегля от съответния сайт и после се добавя от *Settings - > Preferences - > Language and Frameworks и се въвежда пътят към scenebuilder.exe.*

С помоща на Scene Builder е възможно бързо да се добавят бутони, текстови полета, лейбъли, контейнери в определна сцена(форма) без програмистът да пише код, тоест се генерира XML код в fxml файлове.

Вече манипулацията на тези контроли се управлява с Java класове, които още се наричат контролери. В тези контролери програмистът е вече длъжен да изработи код, който да управлява дадените контроли.

Например:

При натискането на бутон примерно да се изведе съобщение.

При инициализация на сцената да се конкатинират контроли и т.н.

Контролерите в този проект за разделени на групи.Това са група login, admin и група оwner. Всяка група съдържа контролери, които контролират действията(функциите) на дадена група.

* 1. Обяснение на потребителския интерфейс в проект „Скалдове“.

А) Клас App

Наследява Application класа, който осигурява три основни компонента:

**Stage** – Или прозореца, съдържа всички обекти в една JavaFX апликация. Основния прозорец е създаден от самата платформа. Обекта на този прозорец се подава на start() метода, който се изпълнява винаги пръв при стартирането на JavaFX програма.

**Scene** – Това е вътрешността на прозореца. Сцената от своя страна

Съдържа всички възли(nodes или бутоните, лейбълите и т.н.)

**Nodes** – Възлите са всички контроли в една сцена. Те се делят на Root Node, Branch Node и Leaf Node.

Root Node – e първия контрол в една сцена, той винаги се предава в графа на сцената като корен.

Branch Node – Тези възли винаги имат деца, обикновено те са контейнери като GridPane, VBox, HBox, AnchorPane, Pane и т.н.

Leaf Node – Това са възли деца, задължително имат родител, но нямат деца. Това са например: Бутон, лейбъл, текстово поле и т.н.

В този класа App се съдържа метода start(), който е викан винаги първи при стартирането на приложението. В него е подаден прозореца като аргумент, който първи ще се покаже, а сцената му е декларирана в тялото на метода. Кореновия възел също е деклариран в тялото на start метода и му е присвоен обект от тип FXMLLoader клас, който се парсва към тип Parent. Tози обект зарежда fxml файла за дадената сцена чрез подаване като аргумент към Scene обекта.

FXMLLoader loader = new FXMLLoader(getClass().getResource("/fxml/loadingScreenFXML.fxml"));

Parent loadingScreenRoot = (Parent) loader.load();

Scene loadingScreenScene = new Scene(loadingScreenRoot);

loadingStage.setScene(loadingScreenScene); // задаване на сцена за дадения прозорец

loadingStage.show(); // Показване на прозореца

*Обясненото по-горе в код.*

При стартирането на апликацията се създава прозорец , със съответната си сцена и възли, който прозорец съдържа възел Progress Bar. Този Progress Bar се увеличава с 1-ца при всяка итерация на for цикъл. При достигането на 10-тата итерация прозореца loadingStage се затваря и се отваря следващия прозорец , който е деклариран в тялото на start() метода.

*Декларация на втория прзорец със сцената му и кореновия възел:*

FXMLLoader loader2 = new FXMLLoader(getClass().getResource("/fxml/loginScreenFXML.fxml"));

Parent loginScreenRoot = (Parent) loader2.load();

Scene loginScreenScene = new Scene(loginScreenRoot);

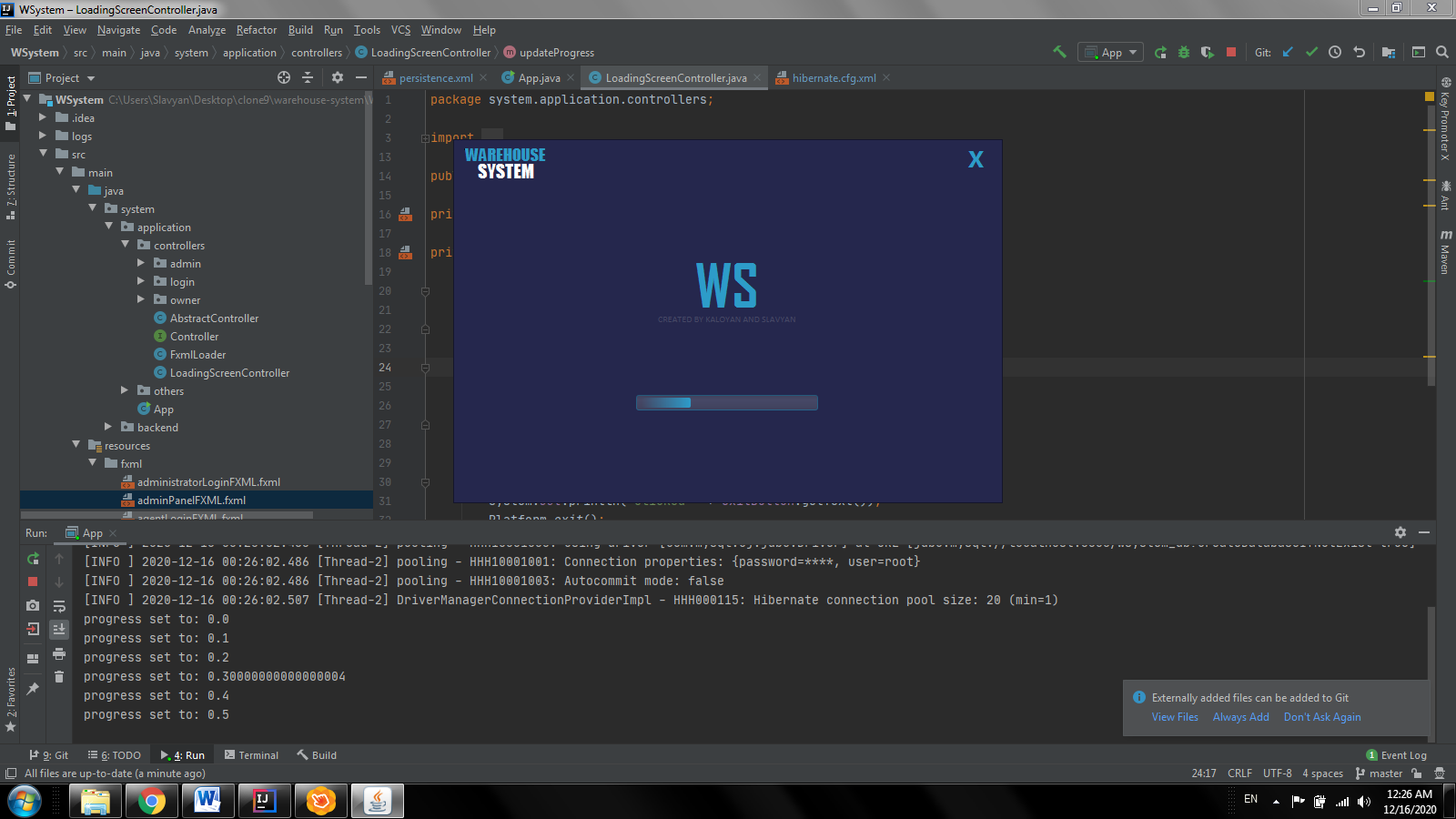
Stage loginStage = new Stage();

loginStage.setScene(loginScreenScene);

*Увеличаване на прогрес бара и затварянето и отварянето на прозорци чрез Thread*

new Thread(() -> { // създаване на thread чрез lambda еxpression, Thread-a позволява паралелната работа на многократни действия, тоест в нашия случай този thread ъпдейтва прогрес бара и затваря и отваря стейдж при достигане на 10-тата итерация, а другия Thread паралено създава/инициализира базата данни на апликацията.  
 for (int i = 0; i <= 10; i++) {  
 try {  
 Thread.*sleep*(750); // забавяне с милисекудни  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 final double progress = i \* 0.1; // I се умножава по 0.1 на всяка итерация  
 loadingScreenController.updateProgress(progress); // викане на метод от контролер  
  
 }  
 Platform.*runLater*(new Runnable(){  
 @Override  
 public void run() { // кода, който ще се изпълни при стартирането на thread  
 loadingStage.close(); // затваря първия прозорец  
 Platform.*setImplicitExit*(false); // задава флага на false = т.е. при затварянето на последния прозорец да не се изключва приложението, а само при викане на exit()  
 loginStage.show(); // отваря следващия  
 }  
 });  
}).start(); // стартиране на thread

Б) Клас LoadingScreenController



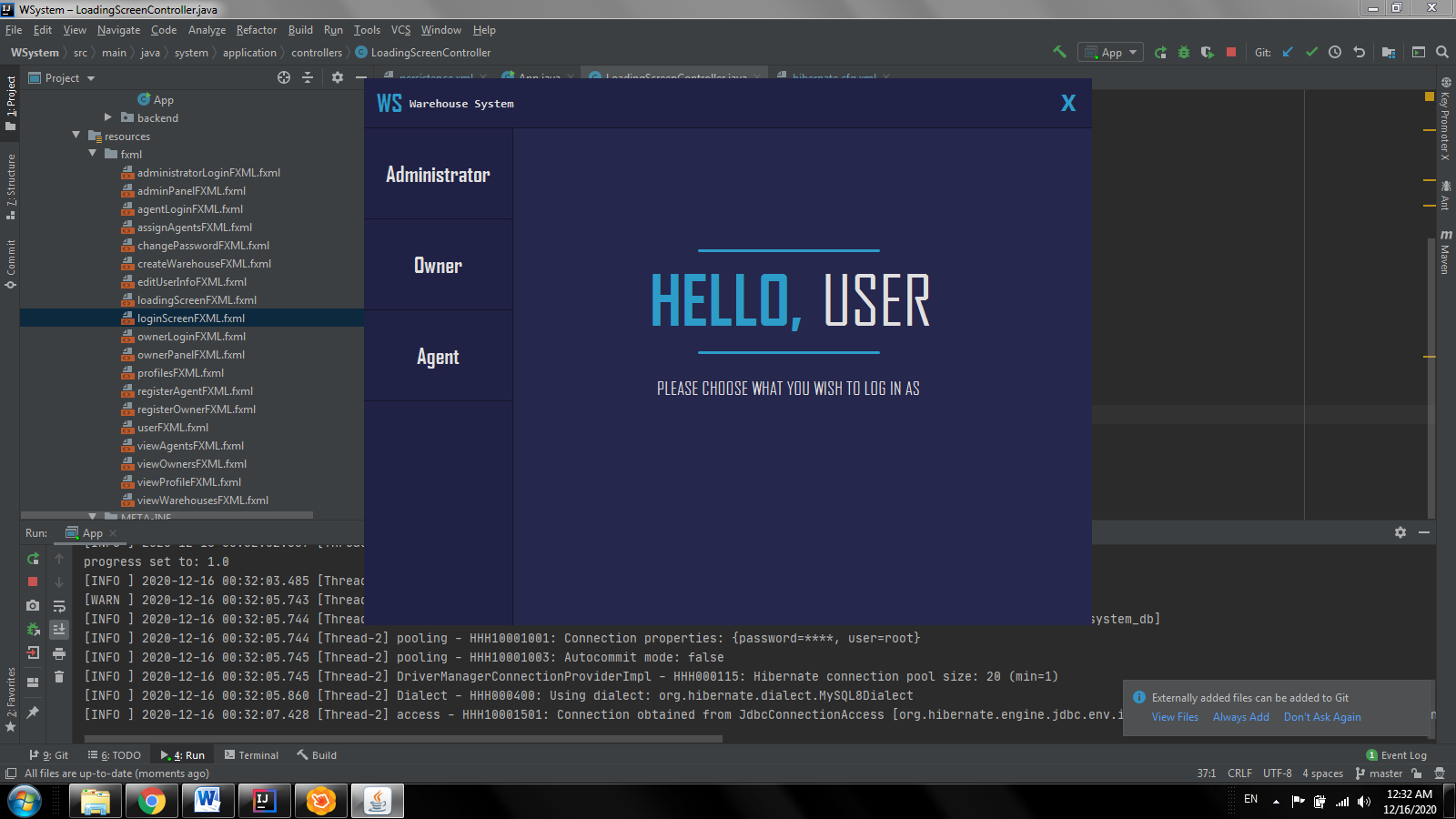
В този клас се увеличава прогреса на прогрес бара и се управлява exitButton.

@FXML  
private Button exitButton = null;  
 @FXML  
private ProgressBar progressBar = new ProgressBar();

public void updateProgress(double v){  
 progressBar.setProgress(v); // задаване на прогрес  
 System.*out*.println("progress set to: " + progressBar.getProgress()); // извеждане в конзолата  
 }

public void onClickEvent(MouseEvent mouseEvent) {  
 System.*out*.println("Clicked " + exitButton.getText());  
 Platform.*exit*(); // при натискане на изход бутона се спира приложението  
 System.*exit*(0);  
}

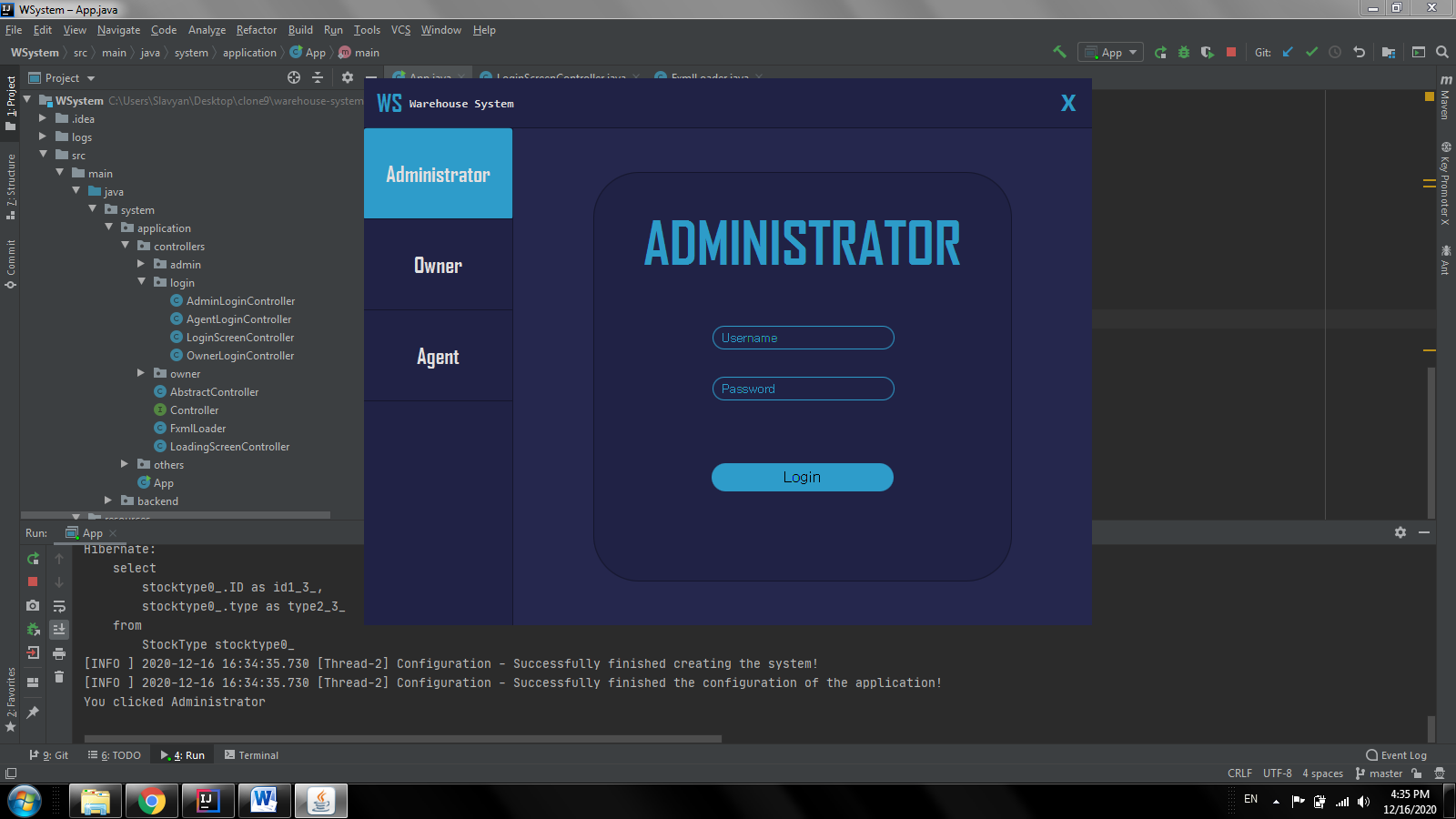
В) Клас LoginScreenController



След затварянето на loading прозореца се показва login stage. Тук потребител може да се логне като администратор, собственик и скалдов агент.

При натискането на бутон от лявата част на сцената се зарежда друг fxml

файл на мястото на “Hello, User” контейнера. Ако натиснем примерно “administrator” бутона, ще се зареди сцената за логване на администратор. Същото се отнася за останалите два бутона, зареждат съответния си fxml файл.



*Зареждане на главния контейнер с друг fxml файл при натискането на бутон от лявата част на основния контейнер.*

public void handleButton1Action(ActionEvent actionEvent) { // метода, който управлява събитието при натискане на бутона administratorButton  
 System.*out*.println("You clicked " + administratorButton.getText()); // извеждане в конзолата  
 setLoader("administratorLoginFXML"); // викане на метода, който осъществява зареждането на другия fxml файл.  
}

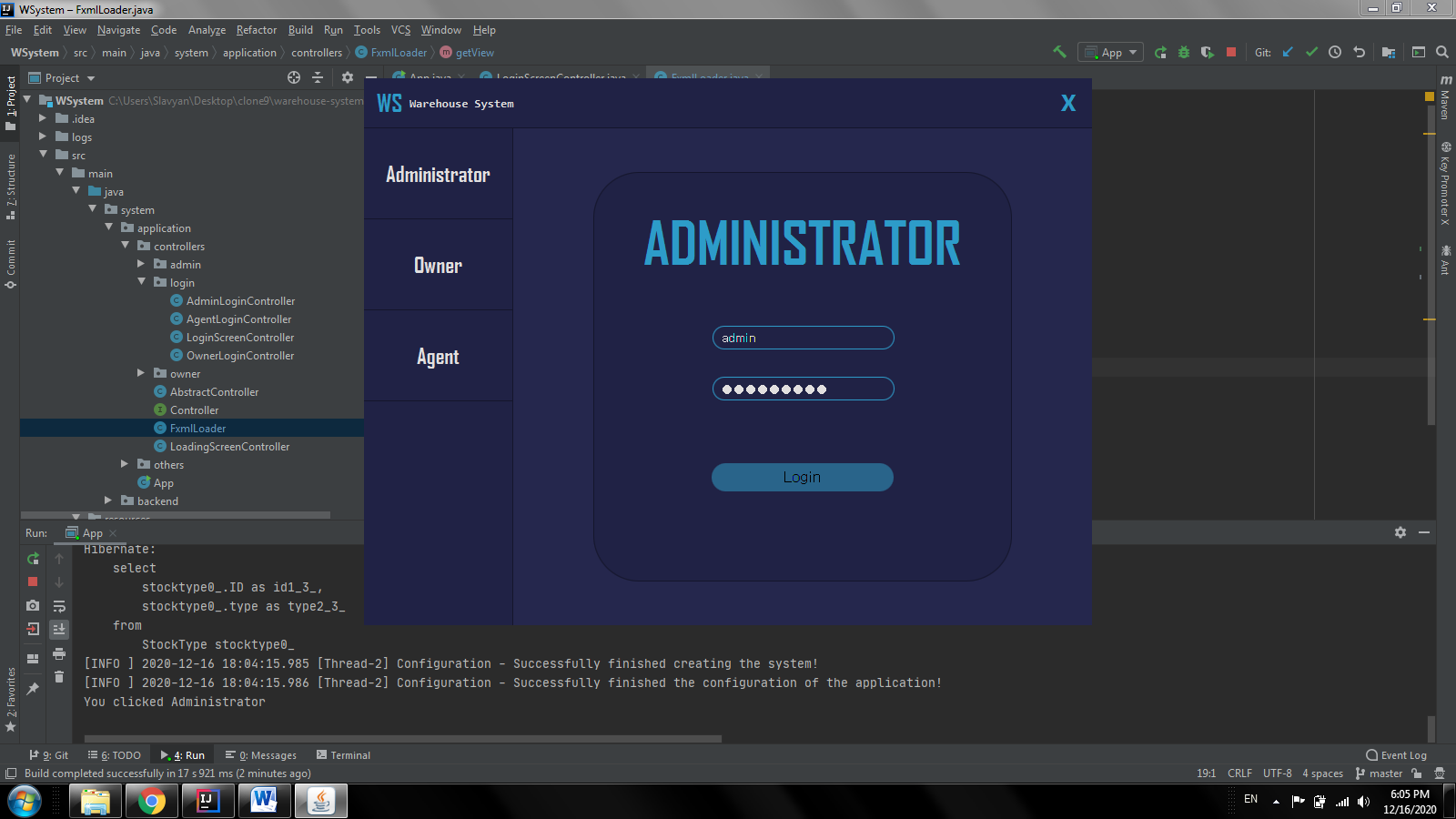
public void setLoader(String fxmlFile){  
 FxmlLoader object = new FxmlLoader(); // създаване на обект от клас FxmlLoader, който ще вика метода getView(path), който пък ще добавя контейнер с нови възли към главния контейнер loaderPane. Новите възли на главния контейнер се съдържат в контейнера, които ще се конкатинира към главния контейнер. Възлите са описани в fxml файла  
 AnchorPane view = object.getView(fxmlFile); // създаване на новия контейнер и му се присвоява контейнера с fxml файла   
 loaderPane.getChildren().clear(); // изчистване на старите възли на главния контейнер  
 loaderPane.getChildren().add(view); // добавяне на нов възел(контейнер в нашия случай) към главния контейнер на сцената. Този нов възел съдържа вече други нови възли, които са описани в fxml файла, който ще се зарежда

}

public class FxmlLoader { // чрез този клас ще се създават обекти, които ще зареждат контейнери в сцени с fxml файлове, които съдържат възли(nodes)  
 private AnchorPane view; // контейнерът, който ще зарежда открития fxml файл  
  
 public AnchorPane getView(String FXMLName) { // getter с аргумент име на fxml файла  
  
 try { // управление на exception  
 URL fileUrl = getClass().getResource("/fxml/" + FXMLName + ".fxml"); // обект от тип URL , който ще държи пълния път на fxml файла  
  
 if (fileUrl == null)  
 throw new java.io.FileNotFoundException("FXML file can't be found");  
  
 view = new FXMLLoader().*load*(fileUrl); // зареждане на контейнера с новия fxml файл  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 return view; // връщане на контейнера вече зареден с нови възли чрез подаден път към fxml файл  
 }  
}

*Класът FxmlLoader, който се използва в контролери за зареждане на нови възли в контейнер*

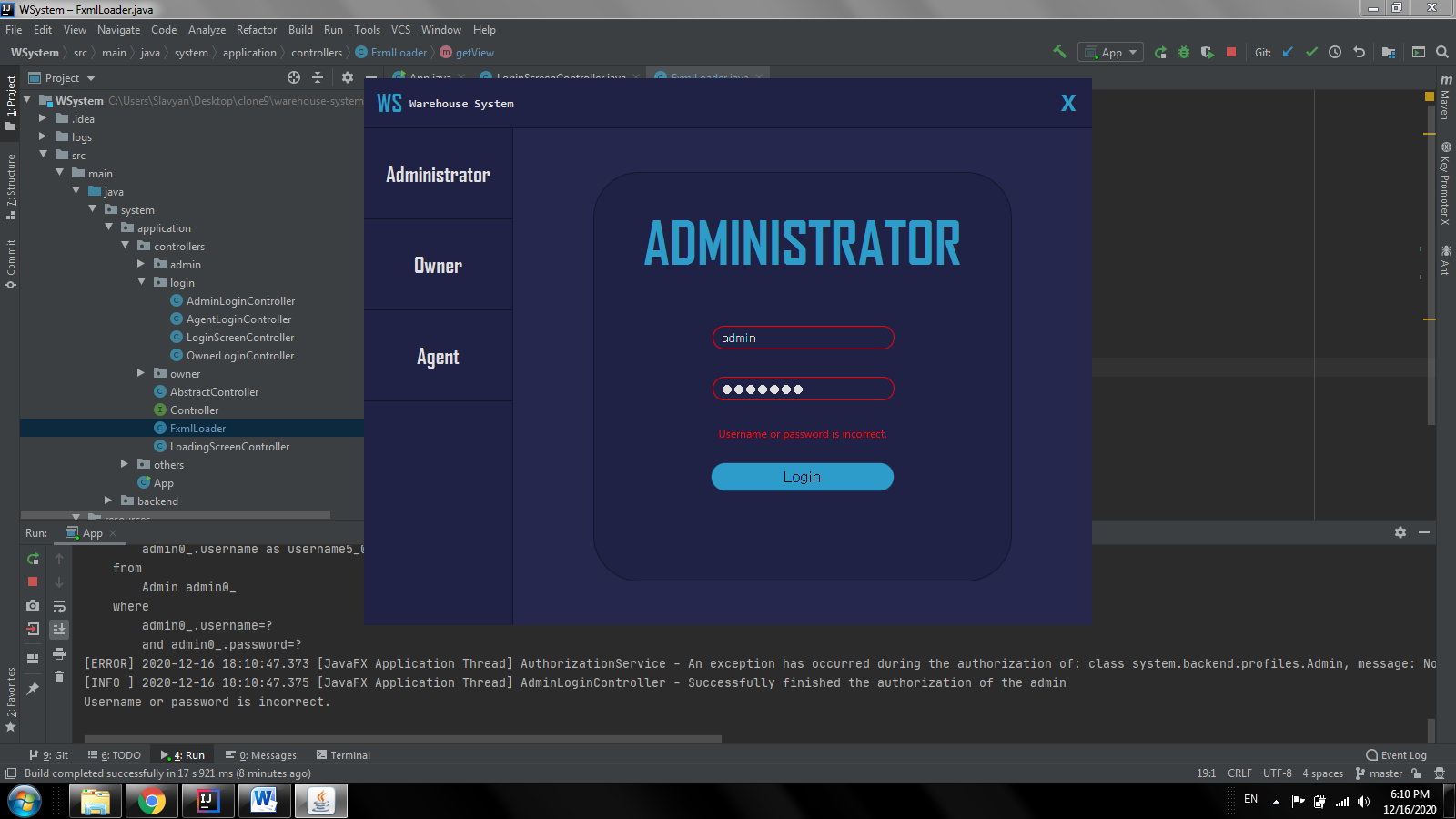
Г) Клас AdminLoginController



Този контролер осъществява логването на потребител като администратор.

Успешното логване се осъществява при правилното въвеждане на данни, които са

поискани от потребителя. При натискането на бутона Login се определя дали данните са правилни. Ако данните не са правилни и има грешка в текстовите полета, потребителят се уведомява, че има грешка:



*Излиза лейбъл за некоретни данни и рамките на текстовите полета се*

*оцветяват в червен цвят*

Следователно потребителят за да успее да се логне е приканен да въведе верните данни. След тяхното въвеждане се натиска отново Login бутона, прави се проверка в метода, който управлява събитието на Login бутона и се затваря сегашния прозорец и се отваря прозореца на администратора, където може да прави неговите си операции.

@FXML  
private TextField usernameField = null;  
@FXML  
private PasswordField passwordField = null;

@FXML  
private Button loginButton = null;

@FXML  
private Label loginFalseLabel = null;

private static final PseudoClass *fieldsCheck\_pseudoClass* = PseudoClass.*getPseudoClass*("fieldsCheck");

*Декларация на основните възли в този клас*

public void setPseudoClassState(boolean state){  
 usernameField.pseudoClassStateChanged(*fieldsCheck\_pseudoClass*, state);  
 passwordField.pseudoClassStateChanged(*fieldsCheck\_pseudoClass*, state);  
}

*Метод, който използва PsuedoClass, който според булевата подадена като аргумент*

*оцветява рамките на текстовите полета чрез CSS*

.usernameField:fieldsCheck{  
-fx-border-color: red;  
-fx-border-radius: 50px;  
-fx-border-width: 1px;  
}

*CSS код, който оцветява бордера на текстовото поле usernameField*

public void setOnAction(ActionEvent actionEvent) {  
 String username = usernameField.getText().trim();  
 String password = passwordField.getText();  
 AuthorizationService<Admin> service = new AuthorizationService<>();  
 boolean success = service.authorizeLogin(username, password, Admin.class);  
 LOGGER.info("Successfully finished the authorization of the admin");  
  
 if(success){ // при правилни данни  
 System.*out*.println("Successfully logged in.");  
 setPseudoClassState(false); // викане на PsuedoClass метода, който в този случай рамките на текстовите полета не се оцветяват в червено, защото е задеден на false  
 closeStage(actionEvent); // викане на метода closeStage, който затваря сегашния прозорец  
 loadStage("/fxml/adminPanelFXML.fxml"); // отваряне на нов прозорец с подаден път на fxml файл  
 } else{ // при неправилни въведени данни  
 System.*out*.println("Username or password is incorrect.");  
 loginFalseLabel.setVisible(true); // лейбъла за грешка се показва  
 setPseudoClassState(true); // викане на PsuedoClass метода, който в този случай рамките на текстовите полета се оцветяват в червено, защото е задеден на true  
 }  
}

*методът, който управлява събитието при натискане на loginButton възела*

public void closeStage(ActionEvent event){  
 loginButton = (Button) event.getSource(); // loginButton присвоява обекта, който е предизвикал събитието и се каства към бутон тип  
 Stage stage = (Stage) loginButton.getScene().getWindow(); // взимане на прозореца на логин бутона и се assign-ва към обекта stage  
 stage.close(); // затваряне на прозореца  
}

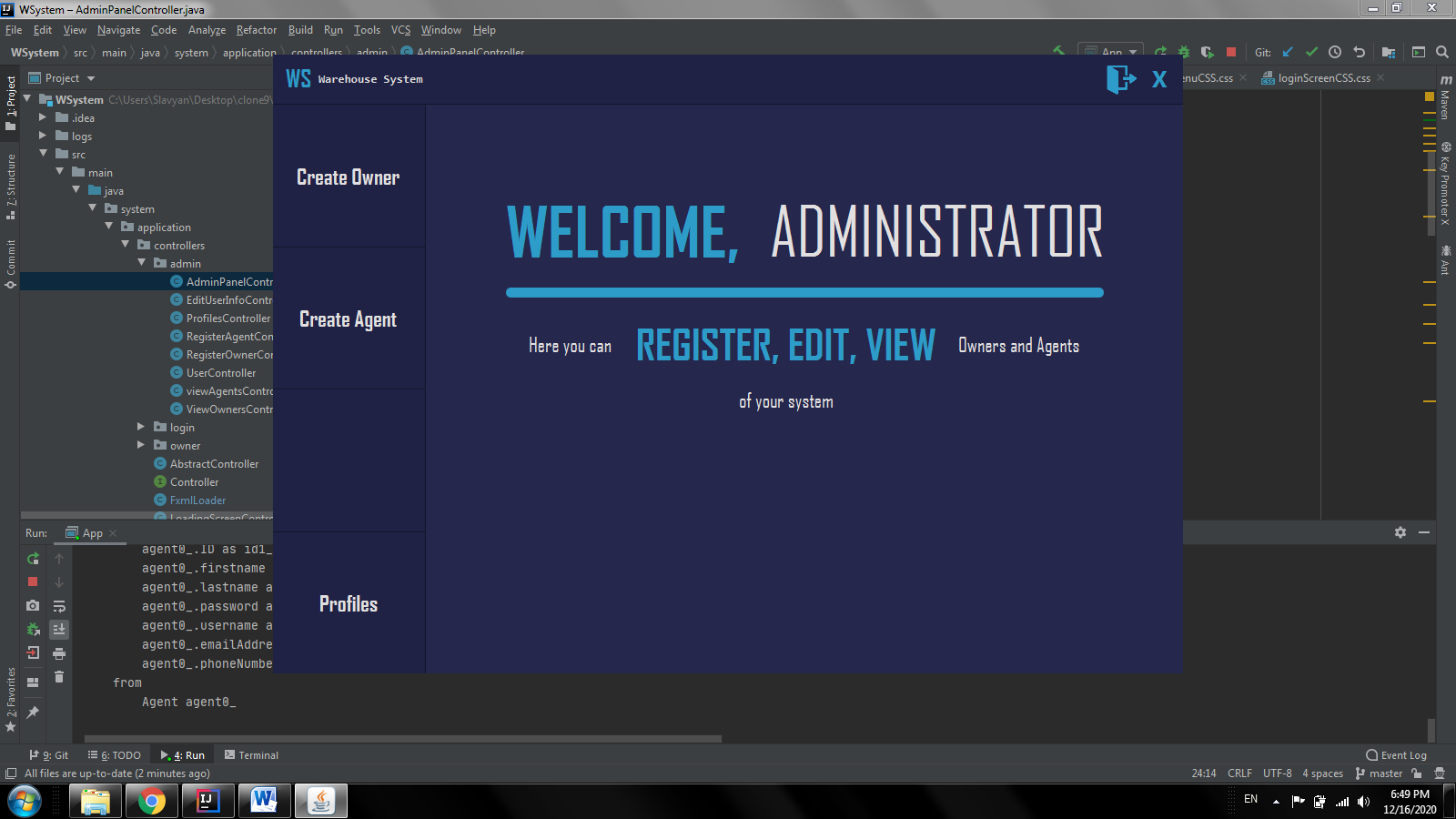
*методът, който се вика при въведени правилни данни.*

*Той затваря сегашния прозорец*

public void loadStage(String fxmlPath) {  
 FXMLLoader loader = new FXMLLoader(getClass().getResource(fxmlPath)); // деклариране на обект FXMLLoader, който ще се използва за зареждане на новия fxml файл  
 Parent adminPanelRoot = null; // деклариране на първия възел за сцената(root node)  
  
 try {  
 adminPanelRoot = (Parent) loader.load(); // задаване fxml файл в кореновия възел  
 }  
 catch (IOException e) { // контролиране на exception  
 e.printStackTrace();  
 System.*out*.println("Couldn't load the stage.");  
 LOGGER.error("Couldn't load the stage.");  
 }  
  
 if (adminPanelRoot == null) // ако кореновия възел не е зададен  
 throw new AssertionError();  
  
 Scene adminPanelScene = new Scene(adminPanelRoot); // деклариране на сцената с кореновия възел  
 Stage adminPanelStage = new Stage(); // деклариране на новия прозорец, който ще се отвори при въведени правилни данни  
 adminPanelStage.setScene(adminPanelScene); // задава сцената на новия прозорец  
 adminPanelStage.initStyle(StageStyle.*UNDECORATED*);  
  
 adminPanelStage.show(); // показва новата сцена  
}

*методът, който зарежда новия прозорец при въведени правилни данни.*

Д) Клас AdminPanelController



Чрез правилните въведени данни в login прозореца може вече да се отвори новия прозорец, който е главния прозорец на администратора. Тук администраторът има възможността да отваря сцени, които имат следните цели:

добавяне собственици и складови агенти към системата,

преглед на профилите им и също тяхното редактиране.

Зареждането на тези сцени става по еднакъв начин като LoginScreenController-a.

При натискането на съответен бутон от лявата част на прозореца се зареждат нови

възли със съответните си child nodes, които са описани в fxml файл, който файл се зарежда с помоща на горе описания клас FxmlLoader.

В най-горната част на сцената са logout бутона и exit бутона. Logout бутонът затваря сегашния прозорец и отваря предишния, който дава възможност да се логнеш като

друг тип потребител. Exit бутонът изключва програмта със System.exit(1).

Profiles бутонът е направен да се вижда само когато има регистрирани агенти или собственици.

@FXML  
private Button exitButton = null;

@FXML  
private Button createOwnerButton = null;  
@FXML  
private Button createAgentButton = null;  
@FXML  
protected Button profilesButton = new Button();  
protected static Button *profilesButton\_static* = new Button();

@FXML  
private Button logoutButton = null;

*Декларация на основните възли в този клас*

public void initialize() {  
 *profilesButton\_static* = profilesButton; // профил бутона се assign-ва към статичен бутон, статичният обект се използва за комуникация между няколко контролера  
 System.*out*.println("AdminPanel initialized.");  
 profilesState(); // викане на метода, който определя дали има профили в системата  
}

*метод, който се изпълнява първи при инициализация на сцената*

public void profilesState(){  
 WSystem wSystem = WSystem.*getInstance*(); // деклариране на обект WSystem, чрез който ще се вика метод hasProfiles, който връща true или false при проверката дали има профили   
 if (wSystem.hasProfiles()) // ако системата има профили  
 *profilesButton\_static*.setVisible(true); // бутонът се активира  
 else  
 *profilesButton\_static*.setVisible(false); // бутонът е неактивен  
}

*метод, който скрива или показва бутона за профили*

public void handleButton1Action(ActionEvent actionEvent) {  
 System.*out*.println("You clicked " + createOwnerButton.getText());  
 setLoader("registerOwnerFXML");  
}

*метод, който контролира събитието на createOwnerButton*

използва по-горе обяснения метод setLoader(fxmlPath)

public void setLoader(String fxmlFile) {  
 FxmlLoader object = new FxmlLoader();  
 AnchorPane view = object.getView(fxmlFile);  
 loaderPane.getChildren().clear();  
 loaderPane.getChildren().add(view);  
 initialize();  
}

*изчиства старите възли на главния котейнер и добавя нови чрез getView(path) метода,*

*който е част от FxmlLoader класа*

public void handleButton2Action(ActionEvent actionEvent) {  
 System.*out*.println("You clicked " + createAgentButton.getText());  
 setLoader("registerAgentFXML");  
}  
  
public void handleButton3Action(ActionEvent actionEvent) {  
 System.*out*.println("You clicked " + profilesButton.getText());  
 setLoader("profilesFXML");  
}

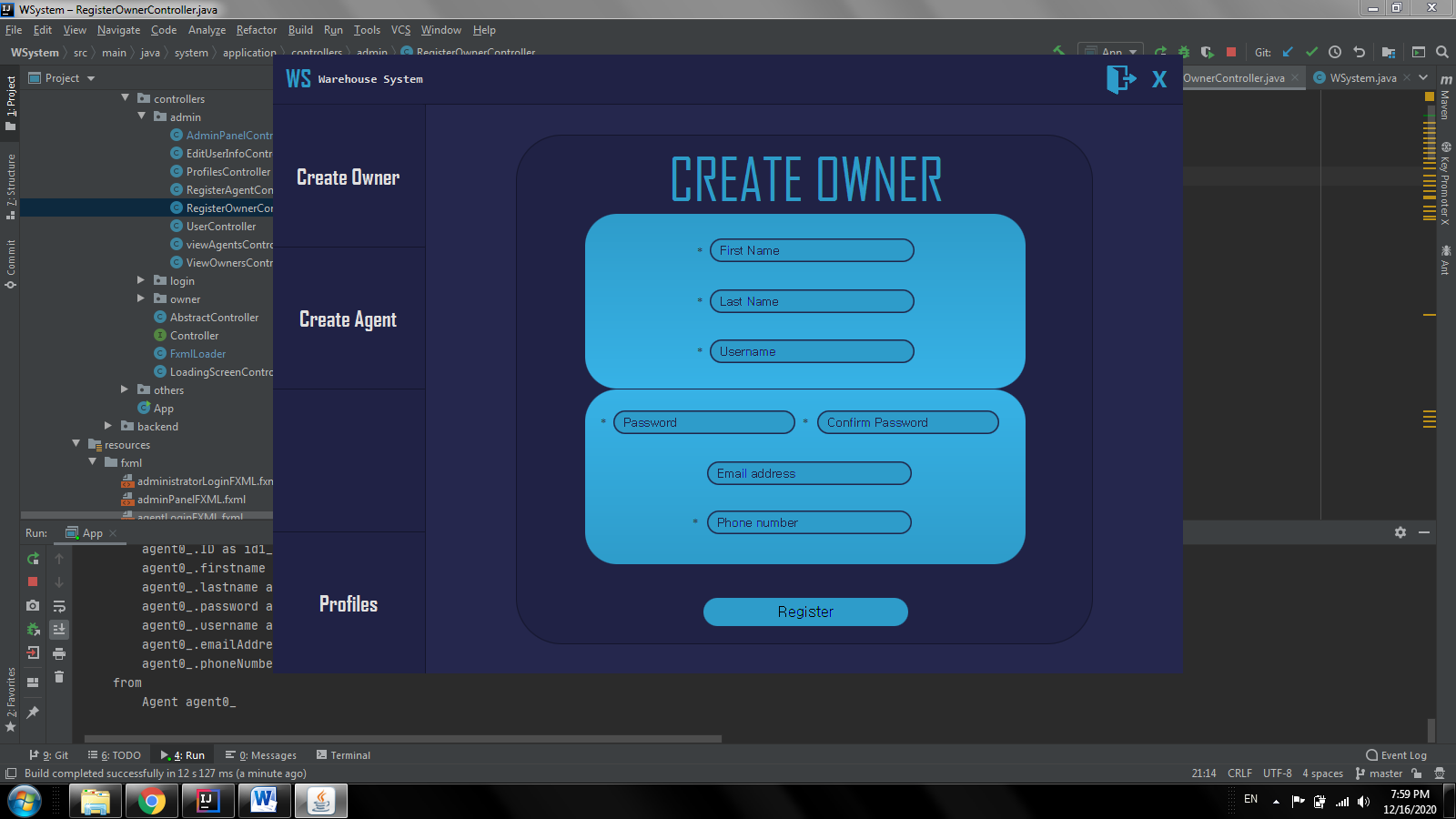
*контролиране на събитието на бутоните profilesButton и createAgentButton*

public void handleLogoutButton(ActionEvent event) {  
 closeStage(event);  
 loadStage("/fxml/loginScreenFXML.fxml");  
}

*управление на събитието на logoutButton, който при натискане затваря сегашния прозорец и отваря*

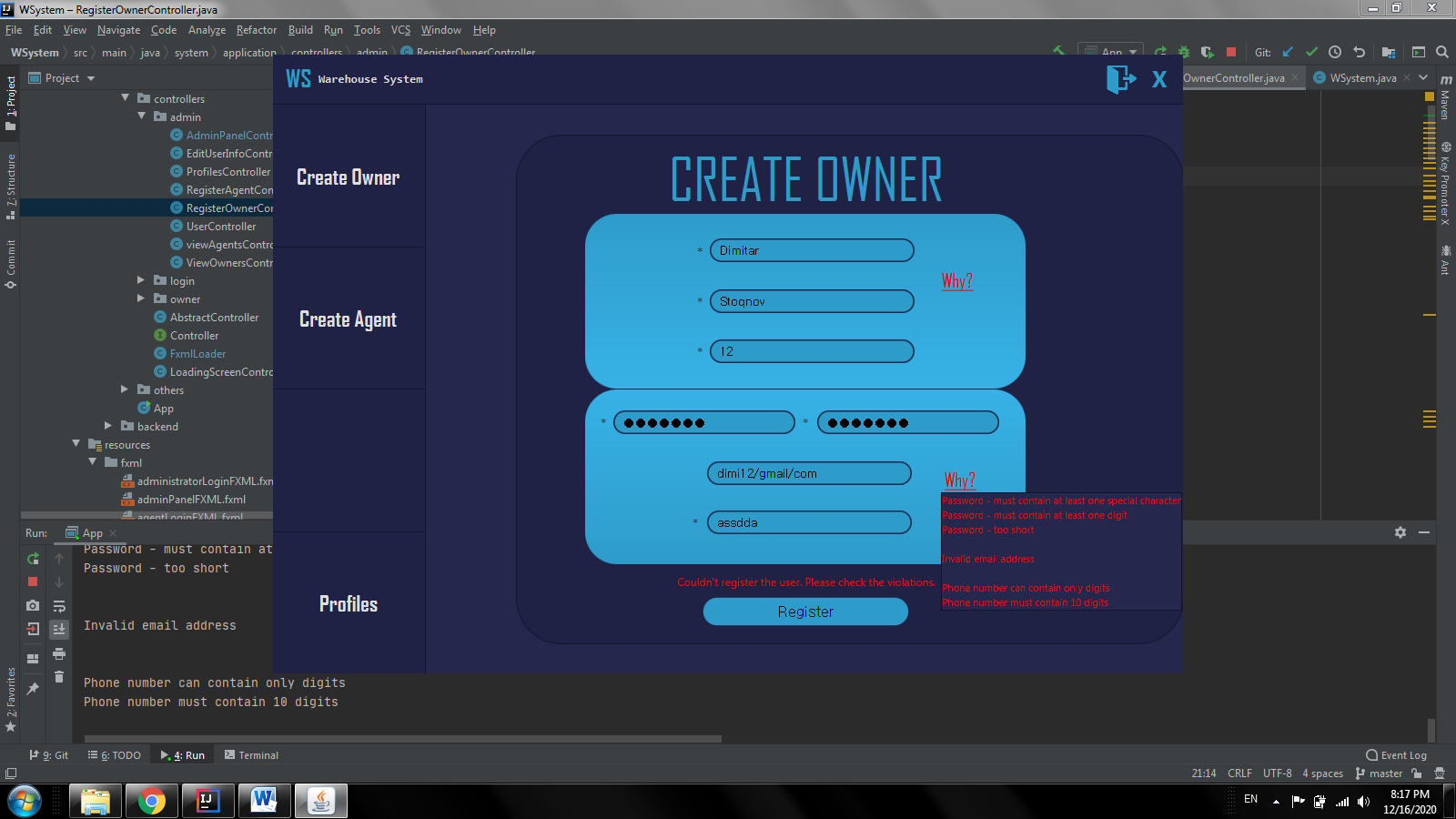
*login прозореца*

Е) Клас RegisterOwnerController



Тази сцена се зарежда, когато createOwnerButton е натиснат. Целта на RegisterOwnerController е да създаде собственик в складовата система при въведени правилни данни. Тези данни се проверяват в метода, който управлява събитието на Register бутона. При правилни въведени данни се извежда лейбъл за успех и собственика на склад е добавен в базата данни на апликацията.

От друга страна ако има някъде въведени невалидни данни в текстовите полета и се натисне бутона за регистрация, се извежда по приличен начин, че администраторът някъде е сбъркал при регистрация на собственик. Съответно, определените грешки се извеждат в сцената.



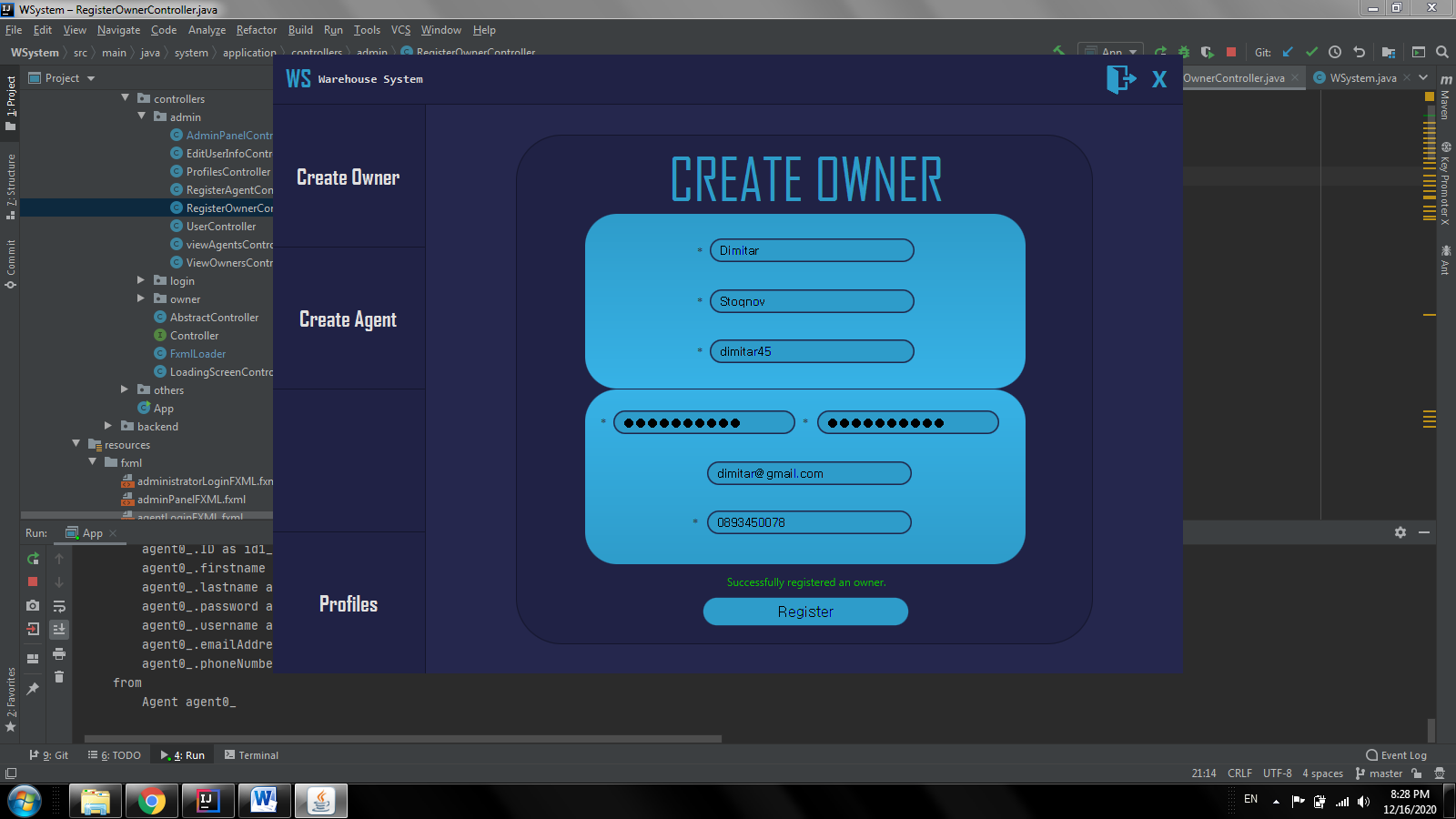
*В случай на грешка, извеждат се съответните грешки в правилата на регистрация*

*на собственик в двете полета*

Когато мишката влезе в областта на HyperLink-a „Why?” се показва малък контейнер с лейбъли, които описват грешките в текстовите полета.

Най-горния HyperLink се отнася за First Name, Last Name и username. Тоест там показва грешките само за тези текстови полета. А долния хиперлинк за останалите 4 текстови полета.

От друга страна, при правилно въведени данни се изчистват възлите(nodes) за грешка и се показва лейбъл за успех(регистрацията е успешна). Също собственика на складове се добавя към базата данни и ако е първият профил в системата, profiles бутонът се показва.



*Правилно въведени данни => Успешна регистрация на собственик*

@Override  
public void initialize(URL url, ResourceBundle resourceBundle) {

// Тук двата контейнера, които включват лейбълите за грешка си регулират размерите според броя, и големината на всички възли в тях  
 consVbox1.setMinWidth(Region.*USE\_COMPUTED\_SIZE*);   
 consVbox2.setMinWidth(Region.*USE\_COMPUTED\_SIZE*);   
 consVbox1.setPrefWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox2.setPrefWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setMaxWidth(Region.*USE\_PREF\_SIZE*);  
 consVbox2.setMaxWidth(Region.*USE\_PREF\_SIZE*);  
  
 consVbox1.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);   
 consVbox2.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setPrefHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox2.setPrefHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setMaxHeight(Region.*USE\_PREF\_SIZE*);  
 consVbox2.setMaxHeight(Region.*USE\_PREF\_SIZE*);  
}

*Методът, който първи се изпълнява при показването на сцената*

public void fillConsBox1(String message) { // съдържа аргумент, който държи текста на грешката  
 Label consLabel = new Label(); // деклариране на лейбъл  
 consLabel.setText(message); // задаване на текста на лейбъла  
 consLabel.setStyle("-fx-text-fill: red; -fx-font-size: 11px"); // задаване на стил на лейбъла  
 consVbox1.getChildren().add(consLabel); // добавяне на лейбъла към първия контейнер за грешки  
 System.*out*.println(message);  
 why1.setVisible(true); // задаване на хиперлинка да се вижда  
}

public void fillConsBox2(String message){  
 Label consLabel = new Label();  
 consLabel.setText(message);  
 consLabel.setStyle("-fx-text-fill: red; -fx-font-size: 11px");  
 consVbox2.getChildren().add(consLabel);  
 System.*out*.println(message);  
 why2.setVisible(true);  
}

*два метода, които запълват малките контейнери с лейбъли при грешка в регистрацията*

Тези контейнери с грешки се показват и скриват чрез следните методи:

public void showConsPane1(MouseEvent mouseEvent) { consVbox1.setVisible(true); }  
  
public void hideConsPane1(MouseEvent mouseEvent) { consVbox1.setVisible(false); }  
  
public void showConsPane2(MouseEvent mouseEvent) { consVbox2.setVisible(true); }  
  
public void hideConsPane2(MouseEvent mouseEvent) { consVbox2.setVisible(false); }

При натискането на Create бутон се вика следния метод:

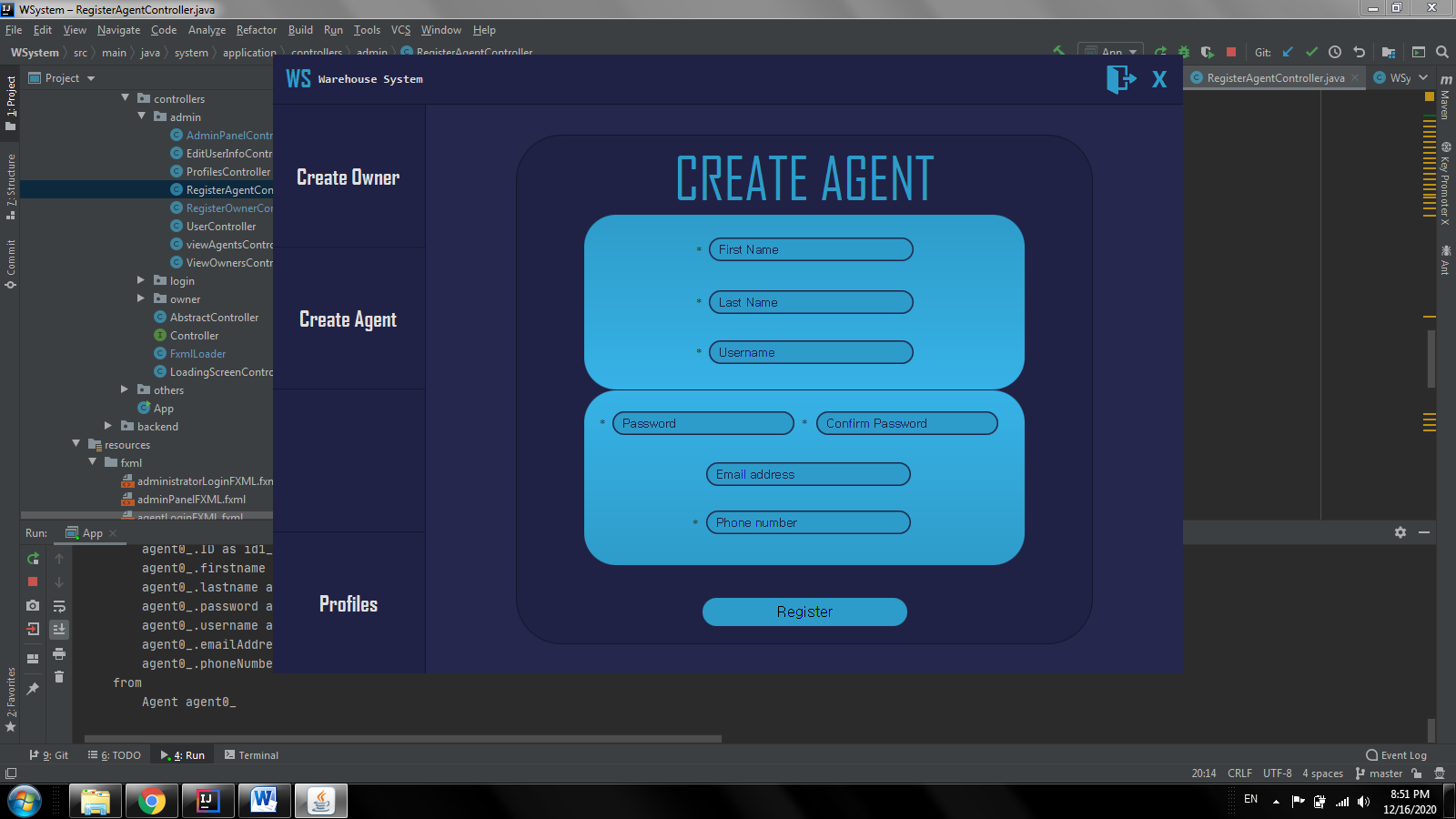
public void registerButtonAction(ActionEvent actionEvent) {  
 consVbox1.getChildren().clear(); // изчиства се първия контейнер с грешки  
 consVbox2.getChildren().clear(); // изчиства се втория контейнер с грешки  
 why1.setVisible(false); // скрива се хиперлинка  
 why2.setVisible(false); // скрива се хиперлинка  
  
 System.*out*.println("Register Button Clicked.");  
  
 DataRetriever dataRetriever = DataRetriever.*getInstance*();  
 dataRetriever.getRegisterData(this, data);  
  
 ValidationService validationService = ValidationService.*getInstance*();  
  
 if(validationService.allDataFilled(data)) {  
 if (validationService.passwordsMatch(data.get("password"), data.get("confirmPassword")))  
 registerOrViolate();  
 else messageService.passwordMatchMessage(this);  
 } else messageService.fillDataMessage(this, data);  
}

*проверява дали има грешки в текстовите полета, ако има извежда дадените грешки,*

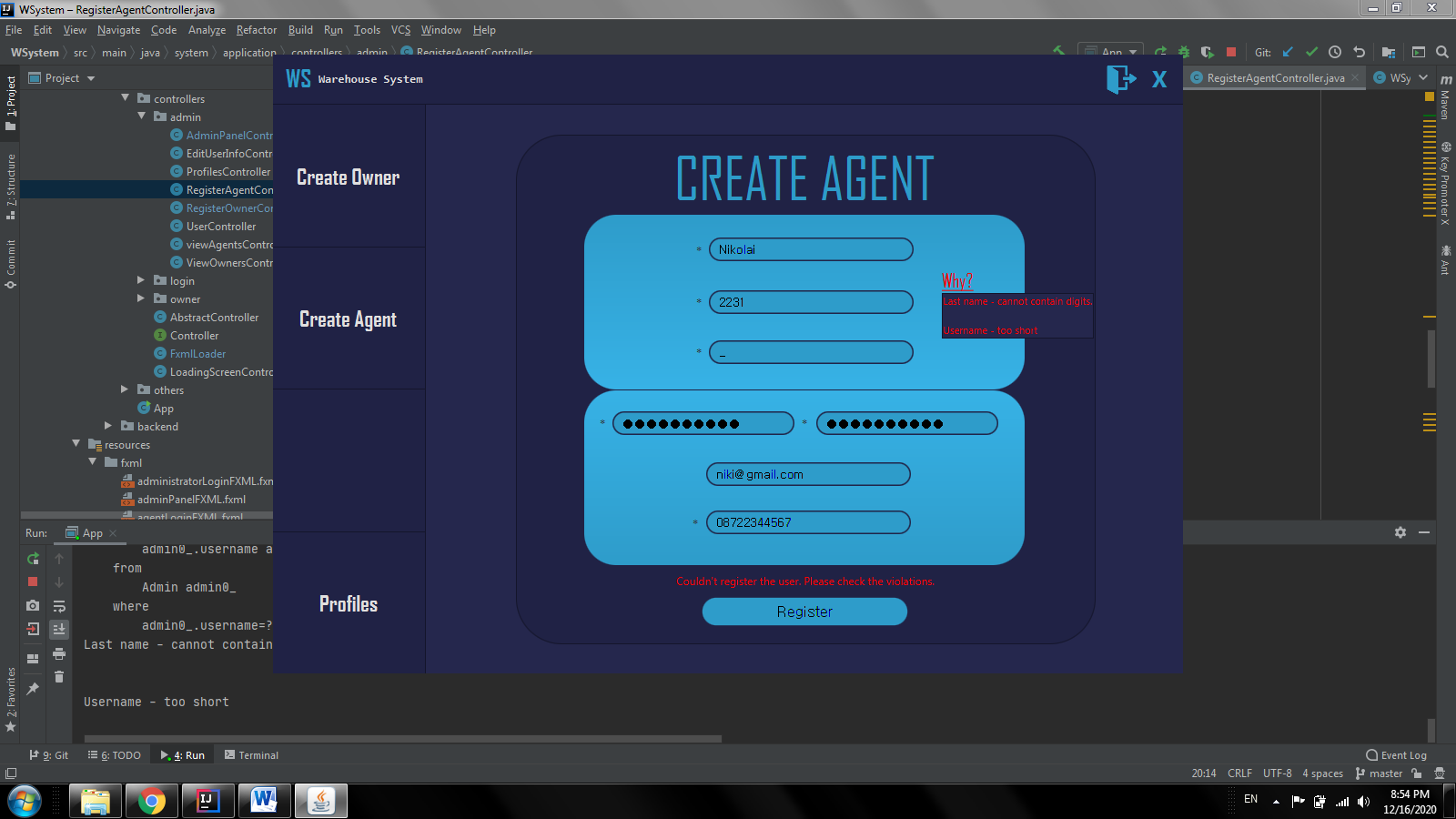
*ако няма грешки добавя собственика в базата данни*

public void registerOrViolate(){  
 cons.clear(); // изчистване на грешките  
  
 ProfileManager<Owner> profileManager = new ProfileManager<>();  
 cons = profileManager.createProfile(Owner.class, data);  
  
 if (cons.isEmpty()) { // ако няма грешки  
 violationsLabel.setVisible(false); // скрива лейбъла за неуспех  
 successLabel.setVisible(true); // показва лейбъла за успех  
 profilesState(); // checks if there are profiles. true => sets button visible. false => sets button invisible. Методът се вика без да се създава обект от друг контролер, тъй като наследява AdminPanelController  
 } else { // ако има грешки  
 violationsLabel.setVisible(true); // показва лейбъла за грешки  
 successLabel.setVisible(false);  
 messageService.showMessages(this, cons, keys);  
 }  
}

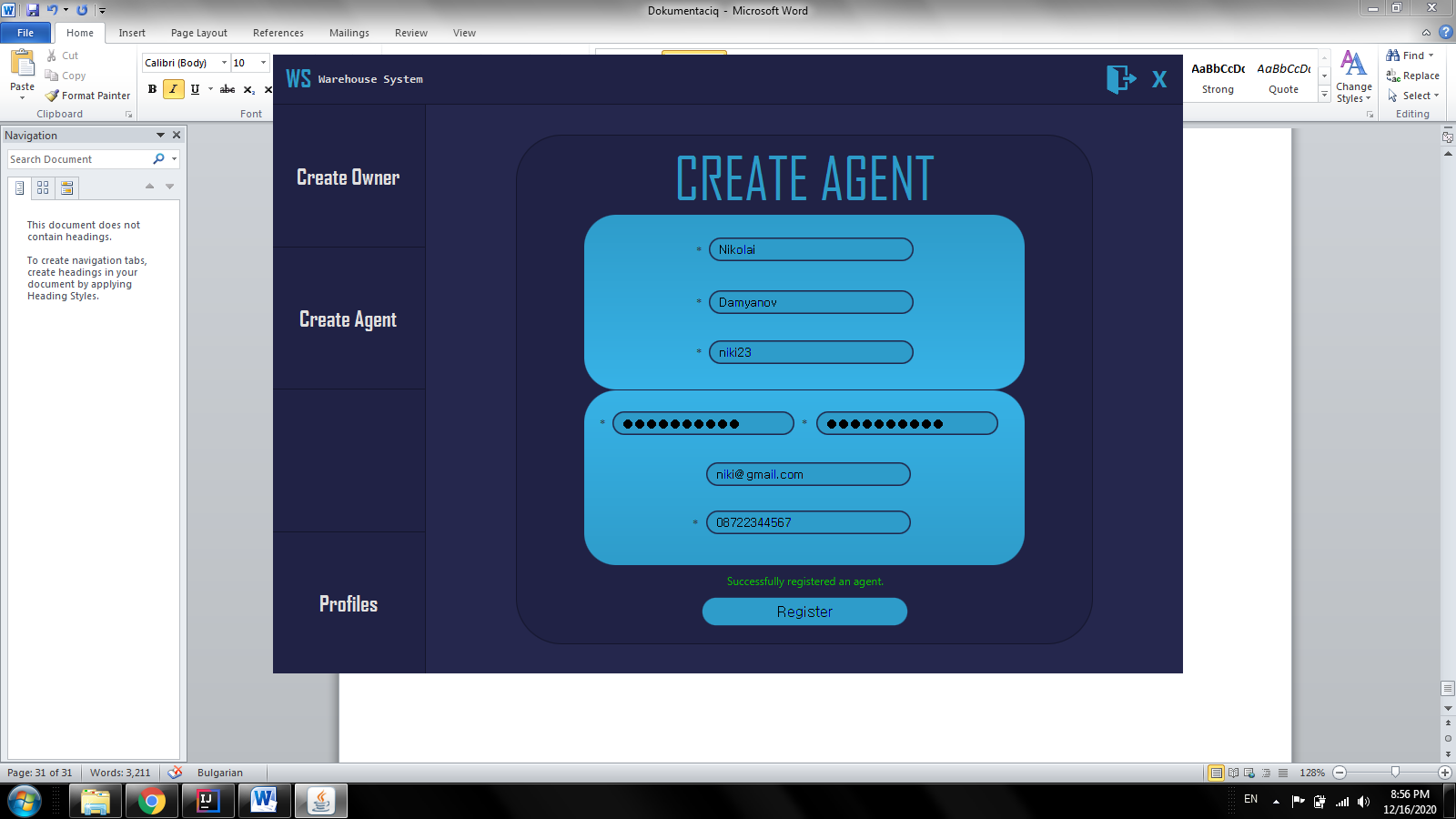
Ж) Клас RegisterAgentController



Функционалността на този контролер е почти еднакъв на RegisterOwnerController, само че тук се регистрира складов агент в складовата система.

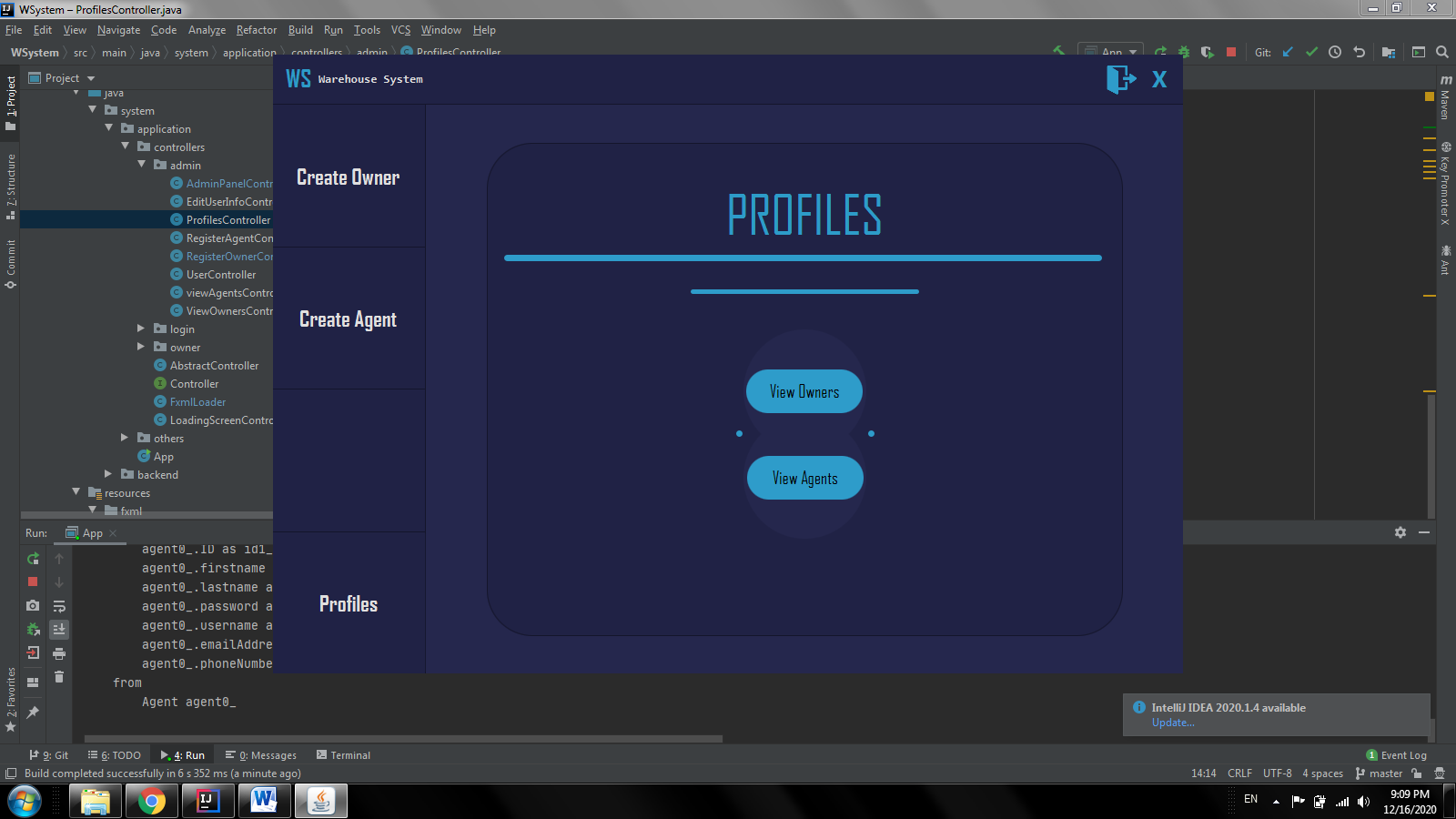


*При излизане на грешки*



*При успех*

З) Клас ProfilesController



При натискането на profiles бутон се зарежда показаната сцена. Тази сцена има два бутона, които се показват само ако съществуват профили за агент и собственик. Ако съответната група от профили не съществува бутона не се показва

При натискането на единия от двата бутона се зарежда друга сцена, която показва всички собственици или агенти в системата.

@FXML  
private Button viewOwnersButton = null;

@FXML  
private Button viewAgentsButton = null;

private BooleanProperty agentProfilesCheck = new SimpleBooleanProperty(false);  
private BooleanProperty ownerProfilesCheck = new SimpleBooleanProperty(false);

*Декларация на основните възли в този клас*

public void initialize(){  
 profilesState(); // викане на метода profilesState(), който проверява дали системата има агенти и собственици => ако има показва съответните бутони  
 viewOwnersButton.visibleProperty().bind(ownerProfilesCheck); // връзване на бутона с BooleanProperty обект, който се използва за проверка дали съществуват профили  
 viewAgentsButton.visibleProperty().bind(agentProfilesCheck); // връзване на бутона с BooleanProperty обект, който се използва за проверка дали съществуват профили  
}

*Методът, който първи се изпълнява при показването на сцената*

public void profilesState(){  
 if (wSystem.hasOwnerProfiles())  
 ownerProfilesCheck.set(true);  
 if (wSystem.hasAgentProfiles())  
 agentProfilesCheck.set(true);  
}

*Методът сетва BooleanProperty обектите на true, ако има профили в системата*

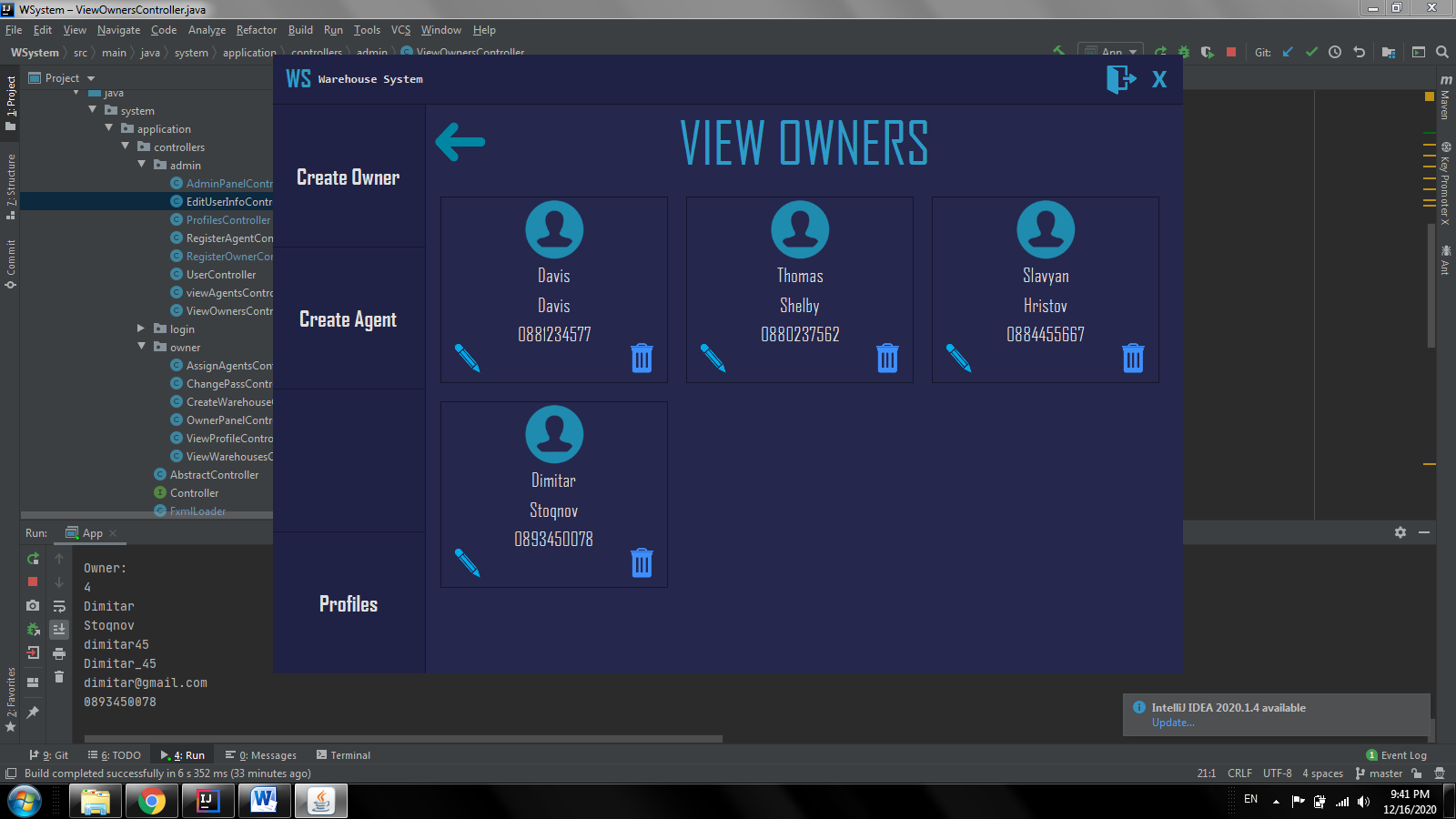
public void handleViewOwners(ActionEvent event) {  
 setLoader("viewOwnersFXML"); // викане на setLoader, който зарежда главния контейнер с нов fxml файл  
 System.*out*.println("Showing owners here:");  
  
 for (Owner owner : wSystem.getOwners()) { // принтиране на всички собственици от системата в конзолата с foreach цикъл  
 System.*out*.println("\nOwner:");  
 System.*out*.println(owner.getID());  
 System.*out*.println(owner.getFirstname());  
 System.*out*.println(owner.getLastname());  
 System.*out*.println(owner.getUsername());  
 System.*out*.println(owner.getPassword());  
 System.*out*.println(owner.getEmailAddress());  
 System.*out*.println(owner.getPhoneNumber());  
 }  
}

*Управление на събитието при натискане на View Owners бутона*

public void handleViewAgents(ActionEvent event) {  
 setLoader("viewAgentsFXML"); // викане на setLoader, който зарежда главния контейнер с нов fxml файл  
 System.*out*.println("Showing agents here:");  
  
 for (Agent agent : wSystem.getAgents()) { // принтиране на всички агенти от системата в конзолата с foreach цикъл  
 System.*out*.println("\nAgent:");  
 System.*out*.println(agent.getID());  
 System.*out*.println(agent.getFirstname());  
 System.*out*.println(agent.getLastname());  
 System.*out*.println(agent.getUsername());  
 System.*out*.println(agent.getPassword());  
 System.*out*.println(agent.getEmailAddress());  
 System.*out*.println(agent.getPhoneNumber());  
 }

*Управление на събитието при натискане на View Agents бутона*

И) Клас ViewOwnersController



View Owners сцената показва всички собственици на складове в системата. Оформлението е направено чрез GridPane(контейнер), който подрежда всяко квадратче със собственик в редове и колони. В случай GridPane-a няма ограничение на броя собственици, които могат да се добавят. Ограничението е само в колоните.

Броят на колоните е зададен на 3. Ако броят на собствениците стане по-голям от 6 се конкатинира ScrollBar отдясно, който позволява да се скролва надолу при голям брой собственици. Във всеки квадрат на собственик има бутони за редактиране на собственик и изтриване. При натискането на редактиране се отваря прозорец, който позволява на администратора да актуализира информацията за дадения собственик. А когато се натисне изтриване, определеният собственик се изтрива от базата данни.

В горната част на сцената има бутон за връщане назад. Той зарежда главния контейнер със стария fxml файл, т.е. Profiles

@Override  
public void initialize(URL url, ResourceBundle resourceBundle) {  
 wSystem = WSystem.*getInstance*();  
 int column = 0; // начален брой на колони  
 int row = 1; // начален брой на редове  
 try {  
 for (int i = 0; i < wSystem.getOwners().size(); i++) { // итерация през всички собственици  
 FXMLLoader fxmlLoader = new FXMLLoader(); // създаване на обект от тип FXMLLoader  
 fxmlLoader.setLocation(getClass().getResource("/fxml/userFXML.fxml")); // задаване на път към fxml файла, който ще се използва  
  
 AnchorPane anchorPane = fxmlLoader.load(); // зареждане на възлите от fxml файла в anchorPane контейнера  
  
 UserController userController = fxmlLoader.getController(); // създаване на обект   
 userController.setOwner(wSystem.getOwners().get(i)); // викане на метод от друг контролер, който сетва лейбълите на всеки квадрат от gridPane-a  
  
 if (column == 3){ // ограничаване на колоните  
 column = 0; // когато се достигне до колона номер 3 се преминава на нов ред  
 row++;  
 }  
 grid.add(anchorPane, column++, row); // добавяне към gridPane-a дадения квадрат на определена позиция според индекса на колоната и реда  
 GridPane.*setMargin*(anchorPane, new Insets(10)); // задаване на празно поле между всеки квадрат. 10px

// следващите редове код автоматично калкулират размерите на квадратите според размера на контейнера, в който са и размера и броя на възлите в квадратите  
 grid.setMinWidth(Region.*USE\_COMPUTED\_SIZE*);  
 grid.setPrefWidth(Region.*USE\_COMPUTED\_SIZE*);  
 grid.setMaxWidth(Region.*USE\_PREF\_SIZE*);  
  
 grid.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);  
 grid.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);  
 grid.setMaxHeight(Region.*USE\_PREF\_SIZE*);  
 }  
 }catch (IOException e) {  
 e.printStackTrace();  
 }  
}

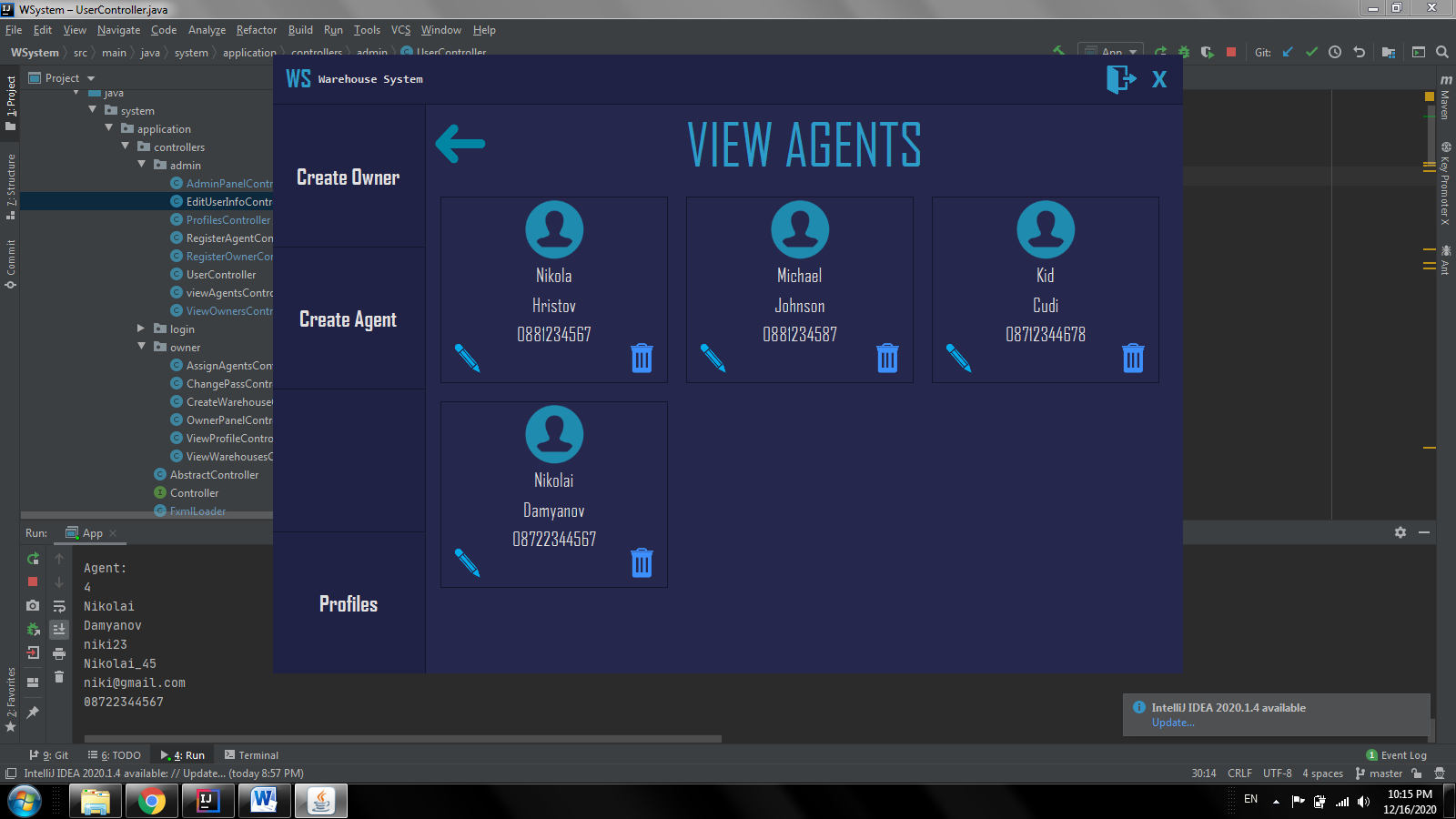
*Методът, който първи се изпълнява при показването на сцената*

public void handleBackButton(ActionEvent event) {  
 setLoader("profilesFXML"); // зарежда нов fxml файл  
}

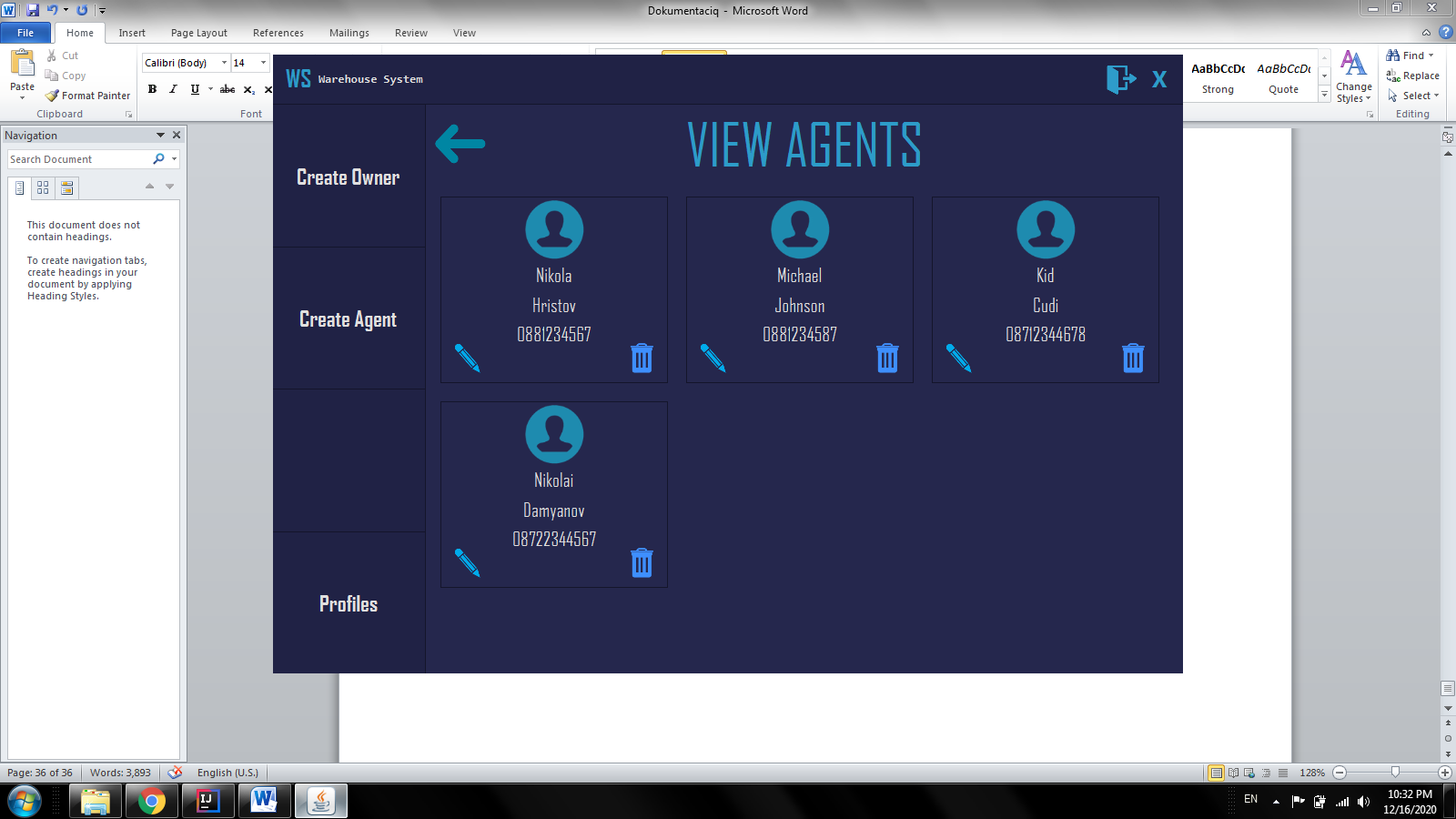
*Метод, който се изпълнява при натискането на back бутон*

Й) Клас ViewAgentsController

Функционалността на този контролер е еднакъв като ViewOwnersController, само че тук се извеждат складови агенти.



К) Клас UserController



Този клас управлява квадратите в контейнера на ViewOwners и ViewAgents

Според определения агент или собственик, данните му се вкарват в трите лейбъла за първо име, второ име, телефонен номер. В този контролер също се управляват събитията при натискане на бутоните за редактиране и изтриване на потребител.

По-късно в OwnerPanel този контролер ще послужи също за извеждането на складовете на собственика.

@FXML  
public Label firstNameLabel;  
public static Label *static\_firstName*; // за комуникация между контролери  
  
@FXML  
private Label lastNameLabel;  
public static Label *static\_lastName*; // за комуникация между контролери  
  
@FXML  
private Label phoneLabel;  
public static Label *static\_phoneNumber*; // за комуникация между контролери  
@FXML  
protected Button deleteButton = null;  
@FXML  
protected AnchorPane anchorPane = new AnchorPane();  
@FXML  
private ImageView imageView = new ImageView();  
private Image image = new Image("/styling/images/warehouseIMG.png");  
private OwnerDataHolder ownerDataHolder = OwnerDataHolder.*getInstance*();  
  
private Owner owner;  
private Agent agent;  
private Warehouse wh;

*Декларация на основните възли в този клас*

public void setOwner(Owner owner){  
 this.owner = owner;  
  
 firstNameLabel.setText(owner.getFirstname());  
 lastNameLabel.setText(owner.getLastname());  
 phoneLabel.setText(owner.getPhoneNumber());

*метод за задаване на лейбълите с текст – информация за собственика*

public void setAgent(Agent agent){  
 this.agent = agent;  
  
 firstNameLabel.setText(agent.getFirstname());  
 lastNameLabel.setText(agent.getLastname());  
 phoneLabel.setText(agent.getPhoneNumber());  
}

*метод за задаване на лейбълите с текст – информация за агента*

public void setWarehouse(Warehouse wh){ // sets image of warehouse, warehouse id and warehouse category  
 this.wh = wh;  
  
 imageView.setImage(image);  
 firstNameLabel.setText("Warehouse " + Long.*toString*(wh.getID())); // Sets id for warehouse  
 lastNameLabel.setText(wh.getCategory()); // Sets category of warehouse  
 phoneLabel.setVisible(false); // hides phoneLabel which is used only for owners and agents  
}

*метод за задаване на лейбълите с текст – информация за склад на собственик*

public void handleDeleteButton(ActionEvent event) {  
 System.*out*.println("Delete button clicked");  
 WSystem wSystem = WSystem.*getInstance*();  
  
 DAO<Agent, String> agentDAO = new MainDAO<>();  
 DAO<Owner, String> ownerDAO = new MainDAO<>();  
 DAO<Warehouse, String> warehouseDAO = new MainDAO<>();  
  
 if(this.owner != null) { // checks if owner is not null  
 ownerDAO.deleteByID(Owner.class, this.owner.getID()); // deletes from database  
 anchorPane.setVisible(false);  
 }  
 else if(this.agent != null){ // checks if agent is not null  
 agentDAO.deleteByID(Agent.class, this.agent.getID());  
 anchorPane.setVisible(false); // deletes from database  
 } else{ // deletes warehouse  
 wh.getOwner().getWarehouses().remove(wh);  
 warehouseDAO.deleteByID(Warehouse.class, this.wh.getID());  
 anchorPane.setVisible(false);  
 }  
}

*изтриване на агент, собственик или склад на собственик*

public void handleEditButton(ActionEvent event) {  
 System.*out*.println("Edit button clicked");  
  
 if (this.agent == null && this.owner == null){ // проверява дали агент и собственик са нул  
 loadEditWarehouse("/fxml/createWarehouseFXML.fxml"); // loads edit warehouse  
 }else  
 loadEditUserInfo("/fxml/editUserInfoFXML.fxml"); // loads edit user  
  
  
 *static\_firstName* = firstNameLabel; // ъпдейтва текста на лейбъла без да се рефрешва сцената  
 *static\_lastName* = lastNameLabel; // ъпдейтва текста на лейбъла без да се рефрешва сцената  
 *static\_phoneNumber* = phoneLabel; // ъпдейтва текста на лейбъла без да се рефрешва сцената  
}

*метод, който се изпълнява при натискането на редактиране*

public void loadEditWarehouse(String path){  
 try {  
 FXMLLoader loader = new FXMLLoader(getClass().getResource(path)); // създаване на обект FXMLLoader, който ще държи пътя към fxml файла, който ще се използва  
 Stage primaryStage = new Stage(); // декларация на прозореца  
 Parent root = loader.load(); // sets root node with fxml file  
 Scene scene = new Scene(root); // деклариране на сцената и задаване на кореновия възел  
 primaryStage.setScene(scene); // задаване на сцената на прозореца  
 primaryStage.initStyle(StageStyle.*UNDECORATED*);  
 setDataForWarehouse(loader); // викане на метод който ще зададе текст на всички текстови полета с данни на склада  
  
 primaryStage.show(); // показване на прозореца  
 }catch (IOException e){  
 e.printStackTrace();  
 }  
}

*метод, който отваря прозорец за редактиране на склад*

public void setDataForWarehouse(FXMLLoader loader){  
 CreateWarehouseController createWarehouseController = loader.getController(); // деклариране на обект от тип CreateWarehouseController  
 createWarehouseController.setData(wh); // викане на метод от друг контролер, който ще зададе подразбиращите се данни във всички контроли  
  
}

*метод, който ще зададе подразбиращите данни във възлите за редактиране*

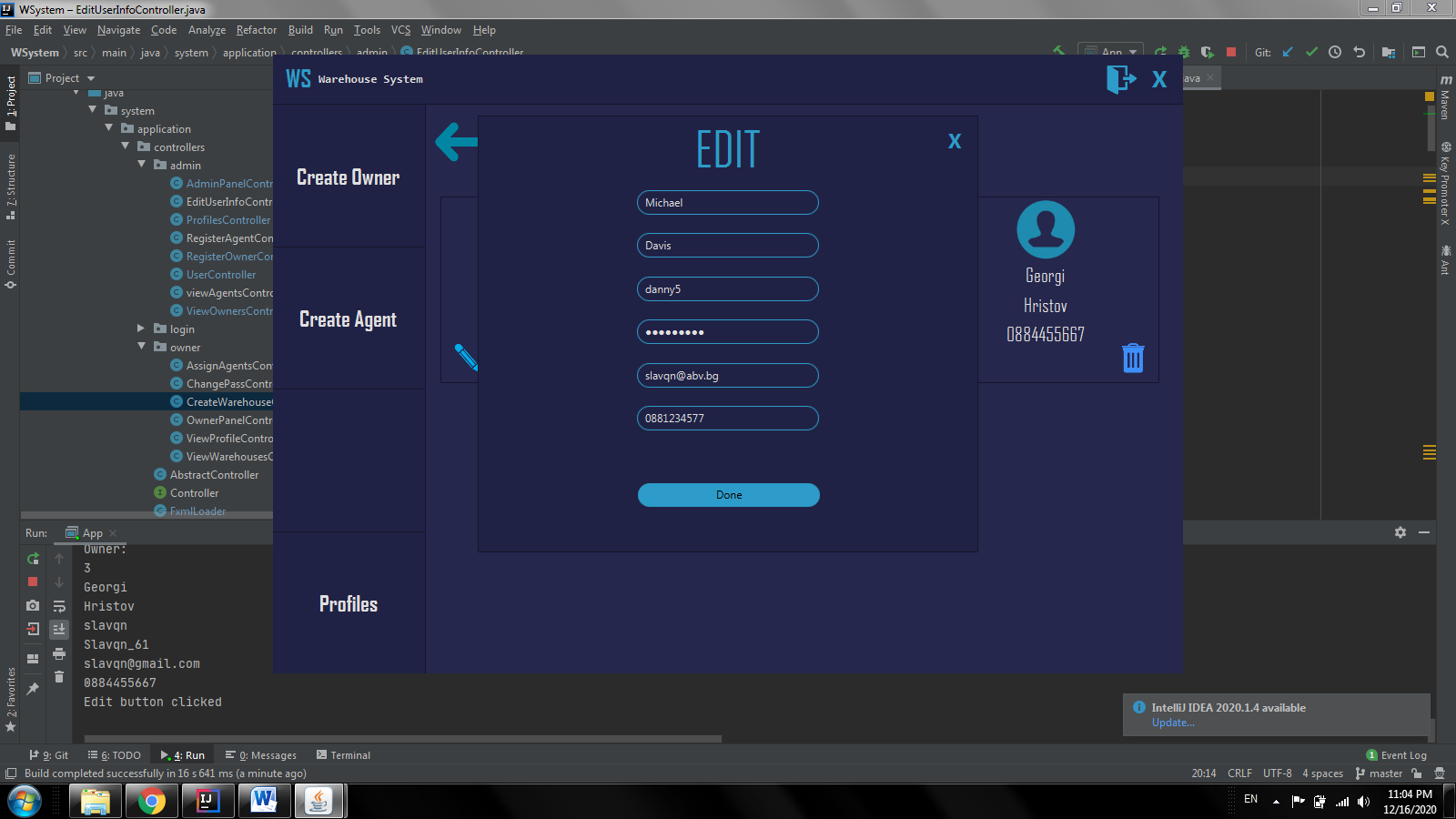
public void setDataForAgentOROwner(FXMLLoader loader){  
 EditUserInfoController editUserInfoController = loader.getController();  
 if (this.owner == null) // ако собственика е null  
 editUserInfoController.setDataForAgent(agent); // // задаване на данни в текстовите полета за агента  
  
 else { ако агента е null  
 editUserInfoController.setDataForOwner(owner); // задаване на данни в текстовите полета за собственика  
 }  
}

*метод, който ще зададе подразбиращите данни във възлите за редактиране*

public void transferMessage(String message){ // sets the warehouse category label  
 System.*out*.println("Transfer Message: " + message);  
 *static\_lastName*.setText(message); // warehouse category  
}

*метод, който ще се вика за да се зададе лейбъла за категория на склад(нужен за комуникация между контролери)*

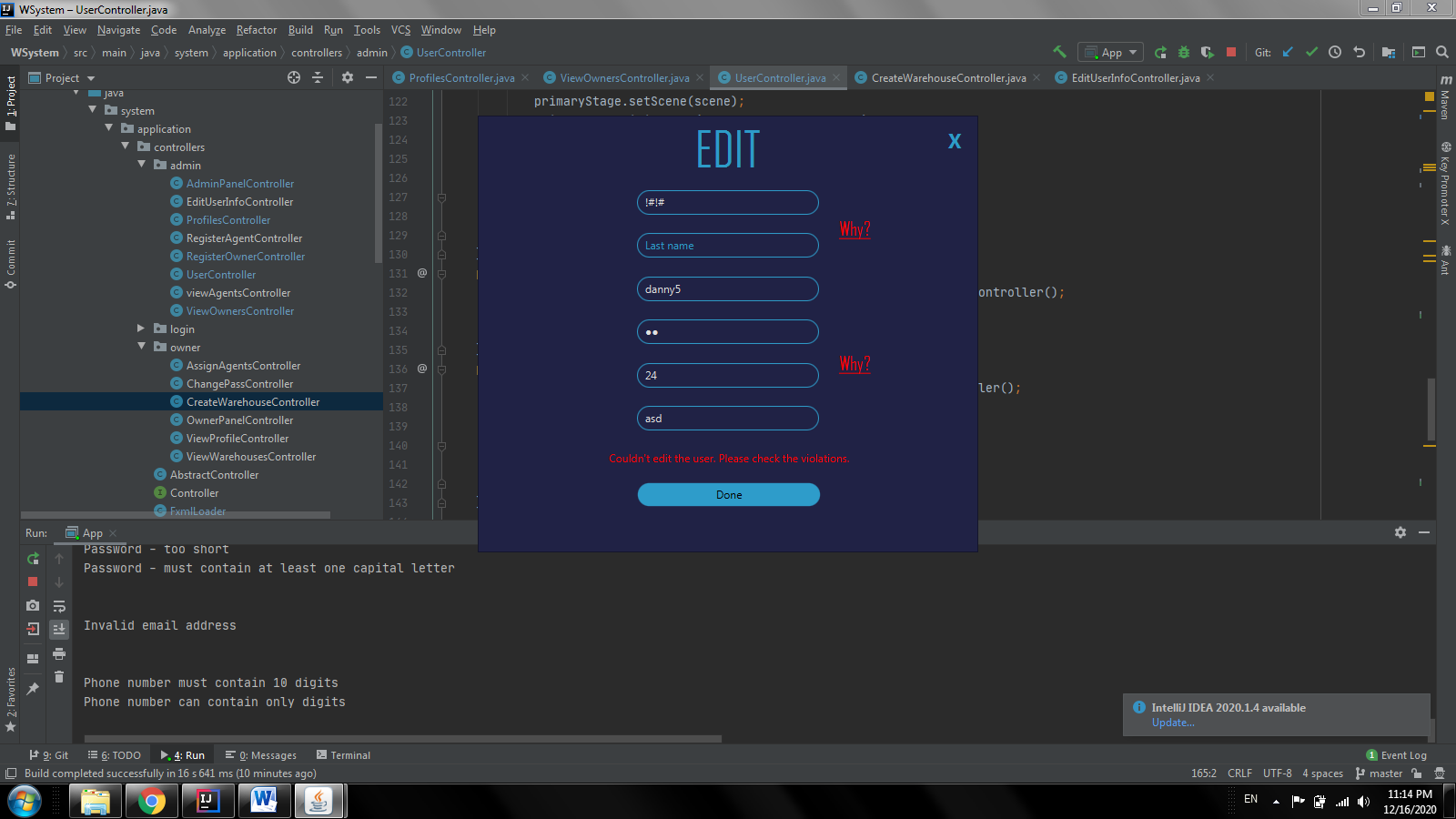
Л) Клас EditUserInfoController



Контролера EditUserInfoController служи за редактиране на агент или собственик на склад. При натискането на бутона за редактиране предварително се инициализират данните в текстовите полета. Когато се натисне бутона Done се прави проверка да променените данни, ако няма грешки при попълването се прави UPDATE в базата данни и потребителят е вече с променена информация.

След като се актуализира инфромацията в базата данни се затваря автоматично прозореца.

При неправилни попълнени данни:



Излиза лейбъл, че актуализацията не успя да се осъществи и също се показват хиперлинкове “Why?”, при които се показва мини контейнер с лейбъли, които описват грешките за всяка секция. Те се показват когато мишката влезе в тяхната област и се скрива когато излезе.

public void setWidthAndHeight(){  
 consVbox1.setMinWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox2.setMinWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setPrefWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox2.setPrefWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setMaxWidth(Region.*USE\_PREF\_SIZE*);  
 consVbox2.setMaxWidth(Region.*USE\_PREF\_SIZE*);  
  
 consVbox1.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox2.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setPrefHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox2.setPrefHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setMaxHeight(Region.*USE\_PREF\_SIZE*);  
 consVbox2.setMaxHeight(Region.*USE\_PREF\_SIZE*);  
}

*метод, който регулира размерите на мини контейнерите, които показват грешките при въведени грешни данни*

public void setDataForOwner(Owner owner){  
 this.owner = owner;  
  
 firstNameField.setText(owner.getFirstname());  
 lastNameField.setText(owner.getLastname());  
 usernameField.setText(owner.getUsername());  
 passwordField.setText(owner.getPassword());  
 emailField.setText(owner.getEmailAddress());  
 phoneNumberField.setText(owner.getPhoneNumber());  
}

*метод, който задава текста на текстовите полета по подразбиране с данните на собственика*

public void setDataForAgent(Agent agent){  
 this.agent = agent;  
  
 firstNameField.setText(agent.getFirstname());  
 lastNameField.setText(agent.getLastname());  
 usernameField.setText(agent.getUsername());  
 passwordField.setText(agent.getPassword());  
 emailField.setText(agent.getEmailAddress());  
 phoneNumberField.setText(agent.getPhoneNumber());  
  
}

*същото, но за складов агент*

public void fillConsBox1(String message) {  
  
 Label consLabel = new Label(); //декларация на лейбъл  
 consLabel.setText(message); // задаване на текста на лейбъла  
 consLabel.setStyle("-fx-text-fill: red; -fx-font-size: 11px"); // задаване на стила на лейбъла  
 consVbox1.getChildren().add(consLabel); // добавяне на лейбъла към контейнера с грешки  
 System.*out*.println(message);  
 why1.setVisible(true); // задаване на хиперлинка да се вижда  
}  
public void fillConsBox2(String message) {  
 Label consLabel = new Label();  
 consLabel.setText(message);  
 consLabel.setStyle("-fx-text-fill: red; -fx-font-size: 11px");  
 consVbox2.getChildren().add(consLabel);  
 System.*out*.println(message);  
 why2.setVisible(true);  
}

*два метода, които запълват малките контейнери с лейбъли при грешка в актуализацията*

public void showConsPane2(MouseEvent mouseEvent) { consVbox2.setVisible(true); }  
  
public void hideConsPane2(MouseEvent mouseEvent) { consVbox2.setVisible(false); }  
  
public void showConsPane1(MouseEvent mouseEvent) { consVbox1.setVisible(true); }  
  
public void hideConsPane1(MouseEvent mouseEvent) { consVbox1.setVisible(false); }

*методите, които задават видимоста, когато мишката влезе и излезе от областа на хиперлинковете*

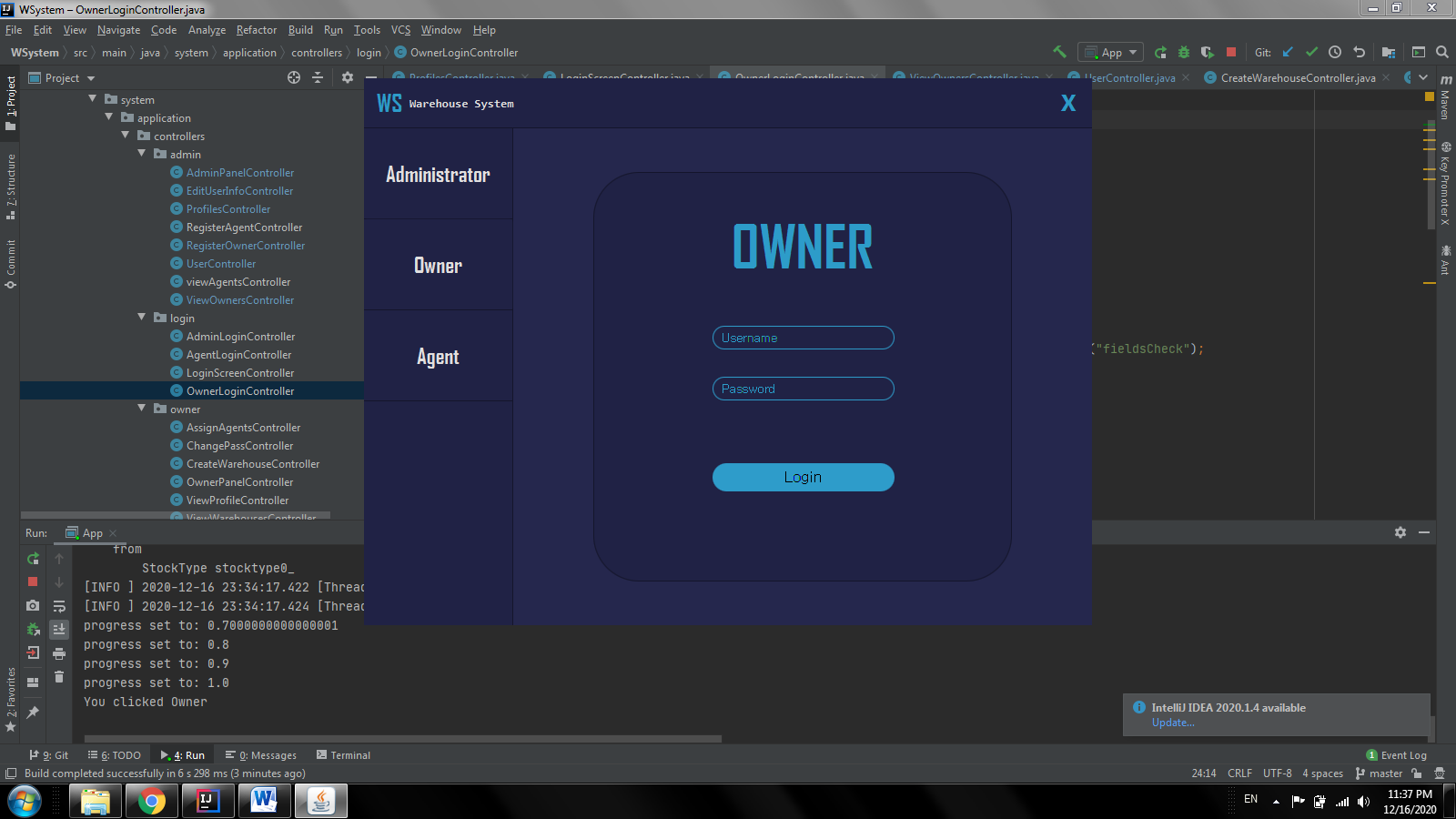
public void handleExitButton(ActionEvent event) {  
 closeStage(event);  
}

*метод, който се вика при натискането на exitButton, затваря прозореца*

public void handleDoneButton(ActionEvent event) {  
 cons.clear(); // изчистване на грешките  
 oldData.clear();  
 newData.clear();  
  
 System.*out*.println("Edit complete.");  
 violationsLabel.setVisible(false); // задаване на видимостта на лейбъла за грешки на false  
 consVbox1.getChildren().clear(); // изчистване на всички възли в контейнера за грешки  
 consVbox2.getChildren().clear(); // изчистване на всички възли в контейнера за грешки  
  
 dataRetriever.getEditData(this, newData);  
  
 if(owner != null){ // ъпдейт на собственик  
 ownerProfileManager.getProfileData(owner, oldData);  
 cons = ownerProfileManager.updateProfile(owner, Owner.class, newData);  
  
 if(cons.isEmpty()) // ако няма грешки  
 closeStage(event); // затваряне на прозореца  
 else {  
 ownerProfileManager.setProfileData(owner, oldData);  
 setWidthAndHeight();  
 violationsLabel.setVisible(true); // задаване на видимостта на true при грешка  
 System.*out*.println("Couldn't update");  
 messageService.showMessages(this, cons, keys);  
 }  
 *static\_firstName*.setText(owner.getFirstname()); // задаване на текст на лейбъл  
 *static\_lastName*.setText(owner.getLastname()); // задаване на текст на лейбъл  
 *static\_phoneNumber*.setText(owner.getPhoneNumber()); // задаване на текст на лейбъл  
 } else {  
 agentProfileManager.getProfileData(agent, oldData);  
 cons = agentProfileManager.updateProfile(agent, Agent.class, newData);  
  
 if(cons.isEmpty()) // ако няма грешки  
 closeStage(event); // затваря прозореца  
 else { // ако има грешки  
 agentProfileManager.setProfileData(agent, oldData);  
 setWidthAndHeight(); // викане на метод за задаване на размерите на контейнера за грешки  
 violationsLabel.setVisible(true); // задаване на видимостта на true при грешка  
 System.*out*.println("Couldn't update");  
 messageService.showMessages(this, cons, keys);  
 }  
 *static\_firstName*.setText(agent.getFirstname()); // задаване на текст на лейбъл  
 *static\_lastName*.setText(agent.getLastname()); // задаване на текст на лейбъл  
 *static\_phoneNumber*.setText(agent.getPhoneNumber()); // задаване на текст на лейбъл  
 }  
}

*метод, който се изпълнява при натискането на Done бутон*

М) Клас OwnerLoginController



Този контролер е същ по функционалност като AdminLoginController, но логва

собственик, а не администратор

Н) Клас OwnerPanelController



Успешното логване на собственик в системата отваря прозореца на собственика. Собственика може да изпълнява следните операции:

* Създаване на складове
* Редактиране на складове
* Назначаване на складови агенти към складове на собственика
* Преглед на собствени складове
* Редактиране на собствените си складове

Бутоните в лявата част на прозореца насочват собственика към операциите.

@FXML  
private Button exitButton = null;  
@FXML  
private Button createWarehouseButton = null;  
@FXML  
private Button assignAgentsButton = null;  
@FXML  
private Button profileButton = null;  
@FXML  
protected Button warehousesButton = new Button();  
protected static Button *warehousesButton\_static* = new Button();  
@FXML  
private Label firstNameLabel = null;  
@FXML  
private AnchorPane loaderPane = null;  
@FXML  
private Button logoutButton = null;  
private WSystem wSystem = WSystem.*getInstance*();  
private Owner owner;

*Декларация на основните възли в този клас*

public void initialize(){  
 *warehousesButton\_static* = warehousesButton; // за конкатиниране на warehousesButton, когато има складове(служи за обмен на информация между контролери)  
  
 OwnerDataHolder ownerDataHolder = OwnerDataHolder.*getInstance*(); // викане да статичен метод, който ще държи собственика, с който сме се логнали  
 owner = ownerDataHolder.getOwner(); // задаване на собственика  
 warehousesState(owner); // проверка дали собственика има складове  
 setFirstNameLabel(owner); // задава името на собственика в главния контейнер  
}

*Методът, който първи се изпълнява при показването на сцената*

public void warehousesState(Owner owner){  
 if (owner.getWarehouses().isEmpty()){  
 *warehousesButton\_static*.setVisible(false);  
 System.*out*.println("This owner hasn't got warehouses");  
 }else{  
 *warehousesButton\_static*.setVisible(true);  
 System.*out*.println("This owner has warehouses");  
 }  
}

*метод, който крие или показва бутона за складове в зависимост дали има складове*

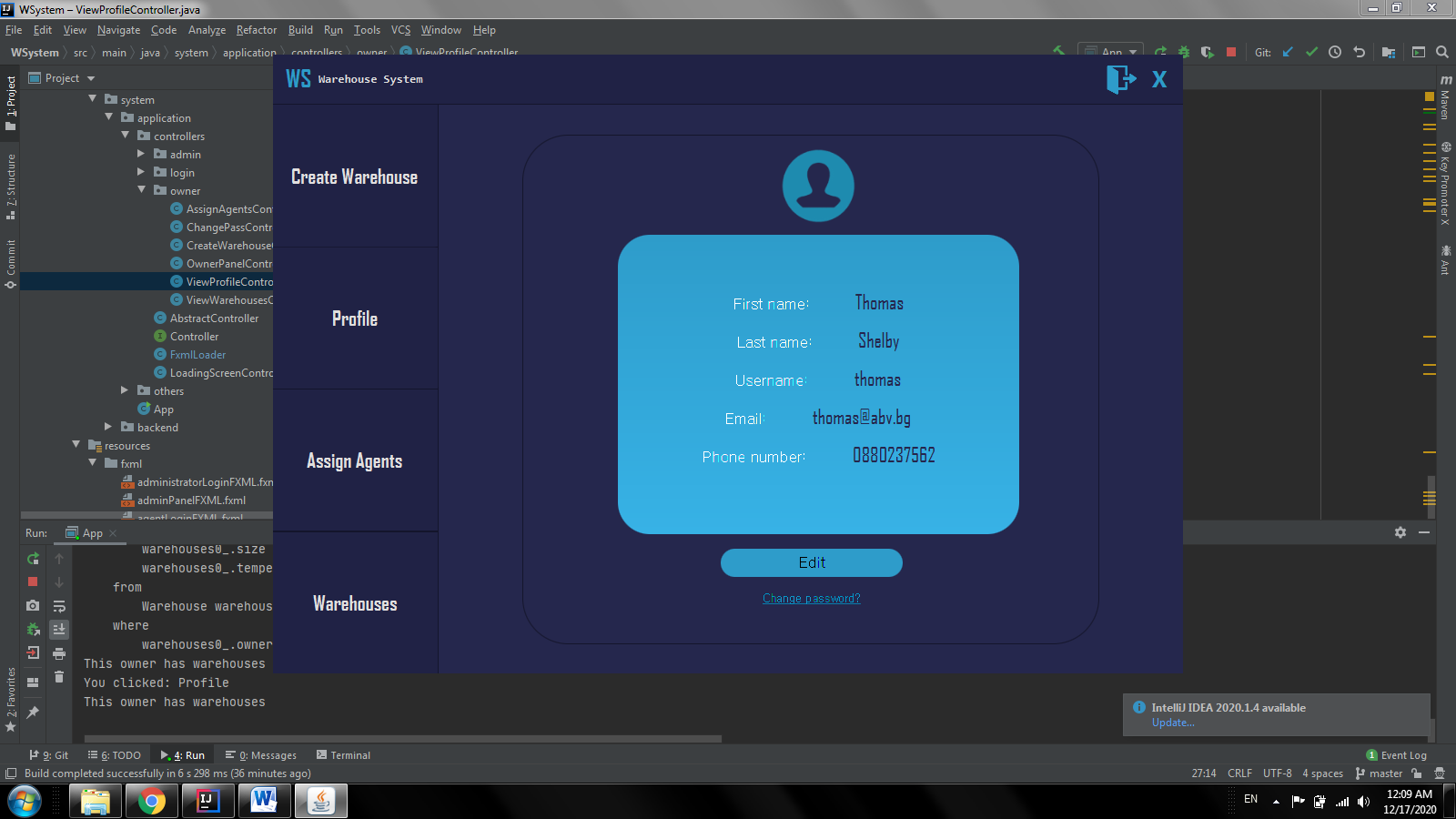
public void setFirstNameLabel(Owner owner){ // Sets the firstNameLabel to the first name of the given owner  
 firstNameLabel.setText(owner.getFirstname());  
}

public void handleLogoutButton(ActionEvent event) {  
 closeStage(event); // затваряне на прозореца  
 loadStage("/fxml/loginScreenFXML.fxml"); // отваряне на нов, който като аргумент има път към fxml файл  
}  
  
public void handleButton1Action(ActionEvent event) { // създаване на склад  
 System.*out*.println("You clicked: " + createWarehouseButton.getText());  
 setLoader("createWarehouseFXML"); // промяна на fxml файла в главния контейнер  
}  
  
public void handleButton2Action(ActionEvent event) { // назначаване на агент  
 System.*out*.println("You clicked: " + assignAgentsButton.getText());  
 setLoader("assignAgentsFXML"); // промяна на fxml файла в главния контейнер  
  
}  
  
public void handleButton3Action(ActionEvent event) { // извеждане на складовете  
 System.*out*.println("You clicked: " + warehousesButton.getText());  
 setLoader("viewWarehousesFXML"); // промяна на fxml файла в главния контейнер  
 List<Warehouse> warehouseList = owner.getWarehouses(); // складовете на собственика се задават в списък  
 if(!warehouseList.isEmpty()) { // ако списъка не е празен  
 System.*out*.println("The owner has warehouses!");  
 for(Warehouse wh : warehouseList){ // извеждане на всички складове с тяхната информация  
 System.*out*.println("Warehouse " + wh.getID());  
 System.*out*.println("Category: " + wh.getCategory());  
 System.*out*.println("Size: " + wh.getSize());  
 System.*out*.println("Temperature: " + wh.getTemperature());  
 System.*out*.println("Stock types: " + wh.getStockTypes());  
 System.*out*.println();  
 }  
 }  
 else System.*out*.println("The owner doesn't have warehouses!");  
}

public void handleButton4Action(ActionEvent event) { // извежда информация за профила на собственика  
 System.*out*.println("You clicked: " + profileButton.getText());  
 setLoader("viewProfileFXML"); // промяна на fxml файла в главния контейнер  
}

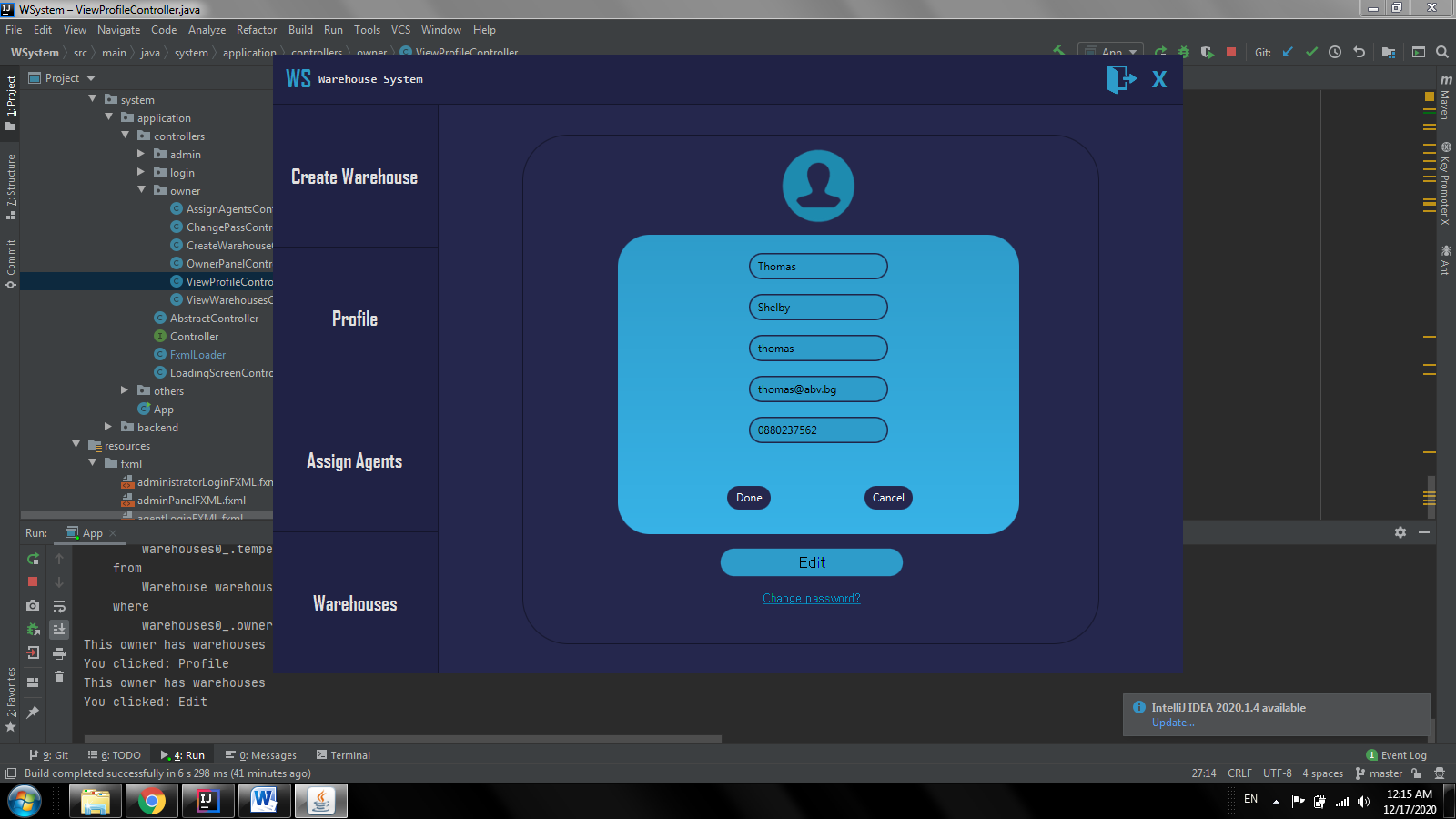
*методите, които се изпълняват при натискане на бутон*

О) Клас ViewProfileController



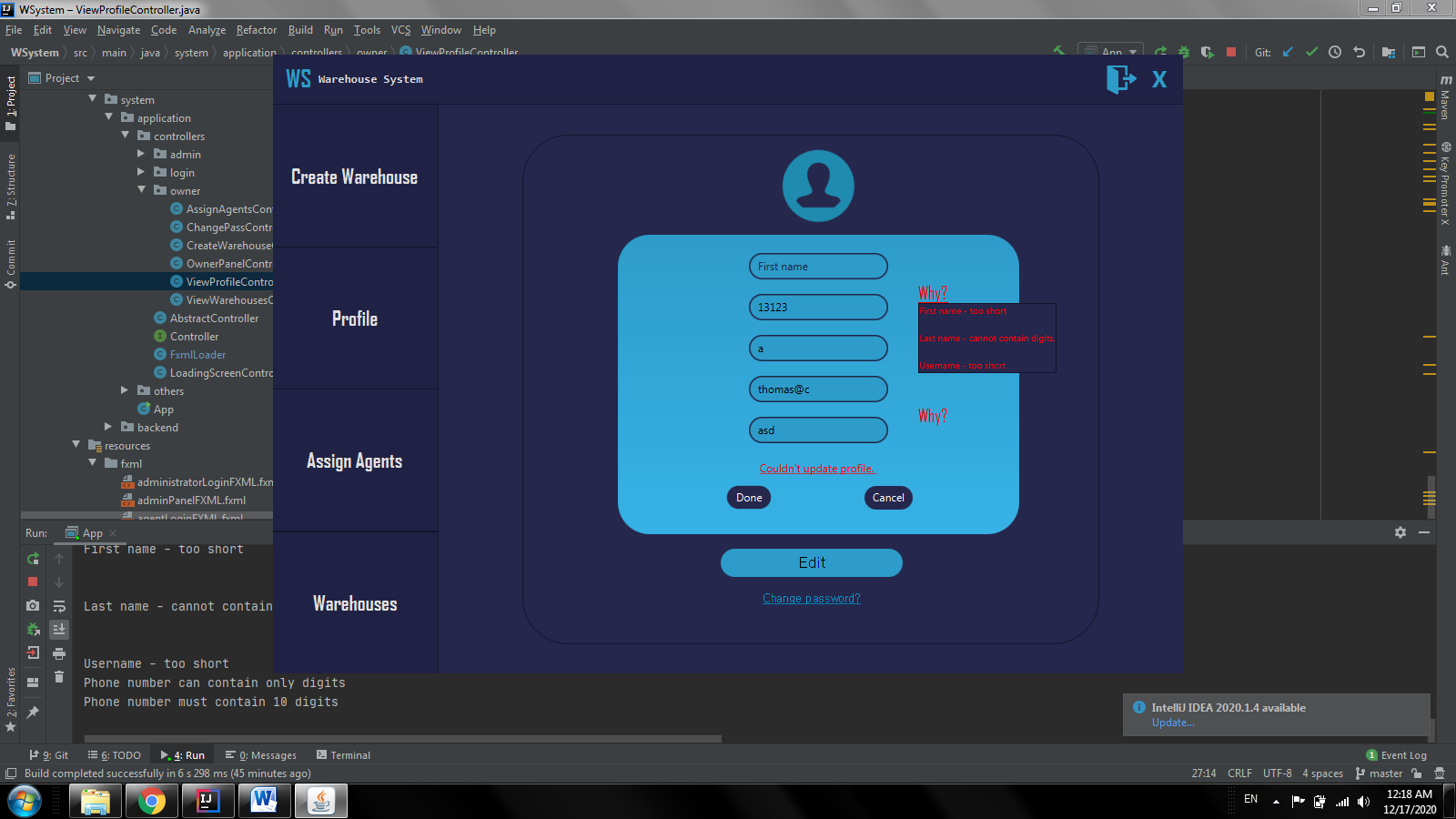
При натискането на бутона Profile се зарежда нова сцена, която съдържа информация за собственика. Собственика има право да промени собствената си инфромация чрез натискането на Edit бутона. Също той може да промени паролата си чрез натискането на хиперлинка “Change password?”.

Когато се натисне Edit бутона синият контейнер в снимката се скрива и се показва друг, който съдържа текстови полета с информацията му. Собственика може да промени личните си данни чрез този нов контейнер:



Както е вече ясно, при натискането на Done бутона се актуализират личните данни на собственика, ако не са възникнали грешки при въвеждането на нови данни.

Ако има грешки отново излизат лейбъли, хиперлинкове и контейнери, които съдържат информация за получените грешки:



При натискането на Cancel бутона, актуализацията се отнема и старите данни си остават.

@Override  
public void initialize(URL url, ResourceBundle resourceBundle) {  
 this.owner = ownerDataHolder.getOwner(); // задаване на собственика  
  
 // sets the labels with owner info  
 firstNameLabel.setText(owner.getFirstname());  
 lastNameLabel.setText(owner.getLastname());  
 usernameLabel.setText(owner.getUsername());  
 emailLabel.setText(owner.getEmailAddress());  
 phoneLabel.setText(owner.getPhoneNumber());  
  
 // Makes the size of the container dynamic  
 consVbox1.setMinWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setPrefWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setMaxWidth(Region.*USE\_PREF\_SIZE*);  
 consVbox1.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setPrefHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox1.setMaxHeight(Region.*USE\_PREF\_SIZE*);  
  
 // Makes the size of the container dynamic  
 consVbox2.setMinWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox2.setPrefWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox2.setMaxWidth(Region.*USE\_PREF\_SIZE*);  
 consVbox2.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox2.setPrefHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox2.setMaxHeight(Region.*USE\_PREF\_SIZE*);  
}

*Методът, който първи се изпълнява при показването на сцената*

public void handleEditButton(ActionEvent event) {  
  
 System.*out*.println("You clicked: " + editButton.getText());  
   
 firstNameField.setText(owner.getFirstname()); // слага текст в текстовите полета в контейнера, който ще се покаже  
 lastNameField.setText(owner.getLastname());  
 usernameField.setText(owner.getUsername());  
 emailField.setText(owner.getEmailAddress());  
 phoneField.setText(owner.getPhoneNumber());  
  
 editPane.setVisible(true); // показва контейнера, който съдържа контролите за актуализация на собственик  
 profileVbox.setVisible(false); // скрива контейнера, който просто показва личните данни за собственика  
}

*Методът, който се изпълнява при натискането на Edit бутона*

public void handleDoneButton(ActionEvent event) { // Updates user info  
 consVbox1.getChildren().clear(); // скрива контейнерите, които съдържат грешките при грешен опит за актуализация  
 consVbox2.getChildren().clear();  
 why1.setVisible(false); // скриване на хипервръзката  
 why2.setVisible(false); // скриване на хипервръзката  
 violationsLabel.setVisible(false); // скриване на лейбъла  
  
 cons.clear(); // изчистване на грешките  
 oldData.clear();  
 newData.clear();  
  
 dataRetriever.getEditDataWithoutPass(this, newData);  
 ownerProfileManager.getProfileDataWithoutPass(owner, oldData);  
 cons = ownerProfileManager.updateProfileWithoutPass(owner, Owner.class, newData);  
  
 if(cons.isEmpty()) { // ако няма грешки  
 System.*out*.println("Owner successfully updated!");  
 editPane.setVisible(false); // скриване на контейнера за актуализация  
 profileVbox.setVisible(true); // показване на контейнера за извеждане на информация за собственика  
 }  
 else { // ако има грешки  
 violationsLabel.setVisible(true); // показване на лейбъла, че актуализация не успя да се осъществи  
 ownerProfileManager.setProfileDataWithoutPass(owner, oldData);  
 messageService.showMessages(this, cons, keys);  
 }

firstNameLabel.setText(owner.getFirstname()); // задава лейбълите в другия контейнер, който показва личните данни на собственика  
 lastNameLabel.setText(owner.getLastname());  
 usernameLabel.setText(owner.getUsername());  
 emailLabel.setText(owner.getEmailAddress());  
 phoneLabel.setText(owner.getPhoneNumber());

}

*Методът, който контролира събитието при натискане на бутона Done*

public void handleCancelButton(ActionEvent event) {  
 consVbox1.getChildren().clear();  
 consVbox2.getChildren().clear();  
 violationsLabel.setVisible(false);  
 why1.setVisible(false);  
 why2.setVisible(false);  
 editPane.setVisible(false);  
 profileVbox.setVisible(true);  
}

*Методът, който контролира събитието при натискане на бутона Cancel*

public void handleChangePass(ActionEvent event) { // loads fxml file  
 System.*out*.println("You clicked: " + changePassword.getText());  
 setLoader("changePasswordFXML");  
}

*Методът, който контролира събитието при натискане на хиперлинка за смяна на парола*

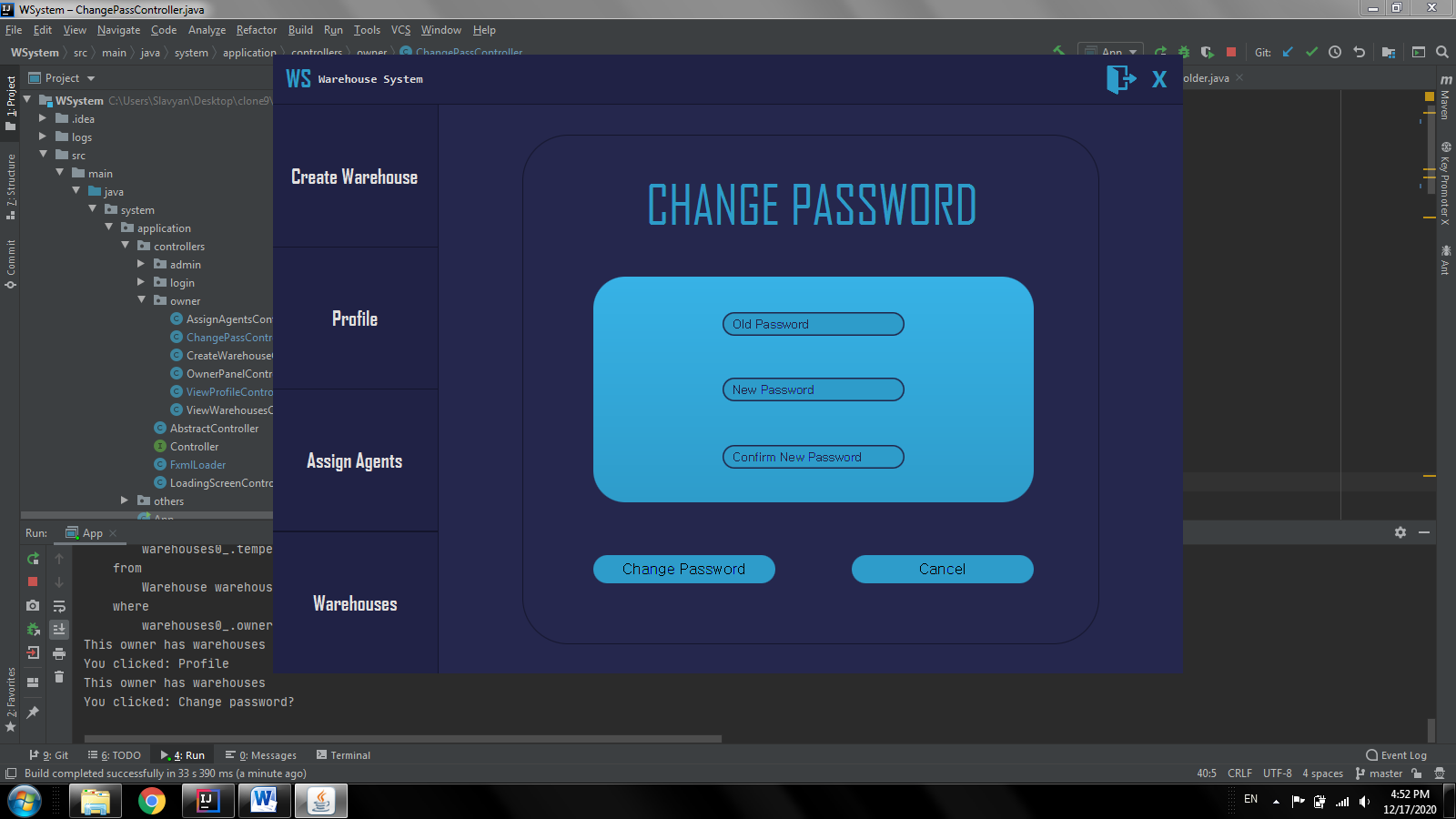
@Override  
public void fillConsBox1(String message) { // Fills the container with violation labels  
 Label consLabel = new Label();  
 consLabel.setText(message);  
 consLabel.setStyle("-fx-text-fill: red; -fx-font-size: 10px");  
 consVbox1.getChildren().add(consLabel);  
 System.*out*.println(message);  
 why1.setVisible(true);  
}  
  
@Override  
public void fillConsBox2(String message) { // Fills the container with violation labels  
 Label consLabel = new Label();  
 consLabel.setText(message);  
 consLabel.setStyle("-fx-text-fill: red; -fx-font-size: 10px");  
 consVbox2.getChildren().add(consLabel);  
 System.*out*.println(message);  
 why2.setVisible(true);  
}

*Методи, които запълват контейнерите за грешки с лейбъли за грешки*

public void showConsBox1(MouseEvent mouseEvent) {  
 consVbox1.setVisible(true);  
}  
  
public void hideConsBox1(MouseEvent mouseEvent) {  
 consVbox1.setVisible(false);  
}  
  
public void showConsBox2(MouseEvent mouseEvent) {  
 consVbox2.setVisible(true);  
}  
  
public void hideConsBox2(MouseEvent mouseEvent) {  
 consVbox2.setVisible(false);  
}

*методите, които задават видимоста, когато мишката влезе и излезе от областа на хиперлинковете*

П) Клас ChangePassController



Когато се натисне хиперлинка, който беше в миналата сцена “Change password?” се отваря сцената за смяна на парола. Собственикът за да смени собствената си парола трябва да въведе първо старата си парола и после в останалите две тесктови полета новата парола. Когато въведената парола в първото текстово поле отговаря на сегашната и данните в останалите две полета са еднакви и отговарят на правилата за парола, тогава новата парола ще може да бъде актуализирана.

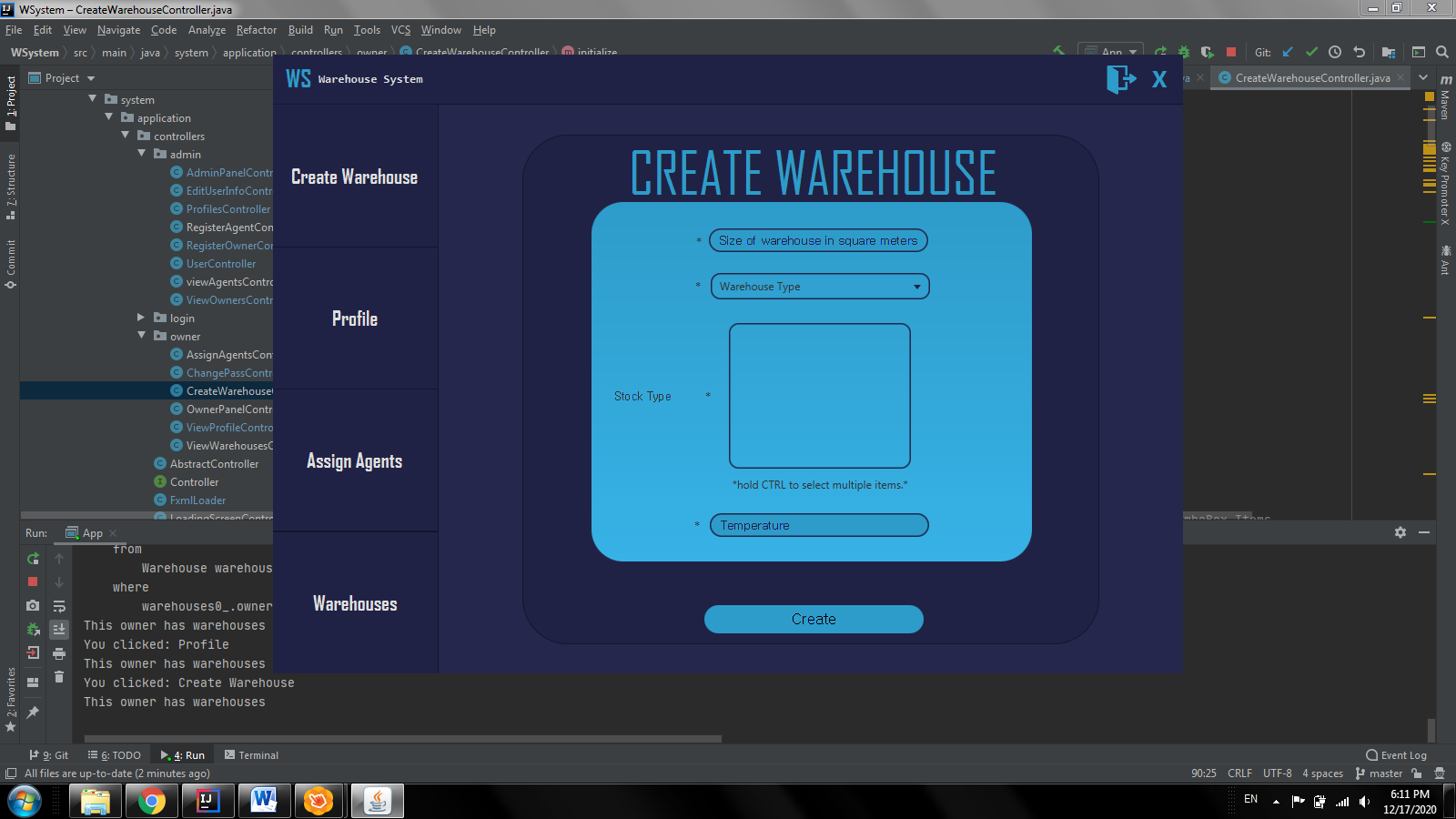
При натискането на бутона за смяна на парола се правят самите проверки.

Ако всичко е правилно ще се изведе лейбъл за успех и паролата ще е сменена и в базата данни.

Противоположно, ако има допуснати някакви грешки при въвеждане на данните за смяна на парола, собственикът ще бъде подходящо уведомен.

Другият бутон, който е до Change Password е бутона “Cancel”, при натискането, собственикът се отказва от смяна на паролата си и се връща към сцената, която показва профила му.

Р) Клас CreateWarehouseController



Тук вече собственикът на складове може да създава вече своите складове. Като въвежда големина на склада в квадратни метри, избира от comboBox категорията на склада и при избрана категория се зареждат в listView-a съответните стоки за категория. След като е избрал категория и стоки за неговия склад, въвежда температурата на склада.

При натискането на “Create” бутона се прави проверка дали всичко е въведено и дали е правилно. Вече когато е правилно тогава се създава складът.

Категории за склад:

* Хранителна индустрия(Food Industry).
* Техническа индустрия(Tech Industry).
* Военната индустрия(Military).

Стоки за категория склад:

Food Industry => Vegetables, Fruits, Seeds.

Tech Industry => Computers, Smartphones, Chargers.

Military => Explosives, Guns, Military Equipment.

Като собственикът, може да избира повече от един тип стока за склад, но само една категория.

@FXML  
private Button createButton;

@FXML  
private ComboBox<String> typeBox = null; // ComboBox for Type of Warehouse  
@FXML  
private ListView<String> stockTypeView = null; // ListView for Type of Stock  
private ObservableList<String> comboBoxItems = FXCollections.*observableArrayList*(); // ComboBox Items  
private ObservableList<String> foodsList = FXCollections.*observableArrayList*(); // ListView items  
private ObservableList<String> techList = FXCollections.*observableArrayList*(); // ListView items  
private ObservableList<String> militaryList = FXCollections.*observableArrayList*(); // ListView items  
private ObservableList<String> stockType = FXCollections.*observableArrayList*();  
@FXML  
private TextField temperatureField = null;

private Warehouse wh;  
private Owner owner;

*Декларация на основните възли в този клас*

public void initialize(){  
 comboBoxItems.addAll("Food Industry", "Tech Industry", "Military"); // Adds items to the list, which will contain different warehouse types  
 typeBox.setItems(comboBoxItems); // sets the list to the ComboBox  
 stockTypeView.getSelectionModel().setSelectionMode(SelectionMode.*MULTIPLE*); // Enables multiple choice for the ListView  
  
 OwnerDataHolder ownerDataHolder = OwnerDataHolder.*getInstance*();  
 owner = ownerDataHolder.getOwner(); // взимане на собственика, който се е логнал  
 // Задава размери на контейнера, който ще показва грешките при неуспешен опит на създаване на склад  
 consVbox.setMinWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox.setPrefWidth(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox.setMaxWidth(Region.*USE\_PREF\_SIZE*);  
 consVbox.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox.setPrefHeight(Region.*USE\_COMPUTED\_SIZE*);  
 consVbox.setMaxHeight(Region.*USE\_PREF\_SIZE*);  
 exitButton.setVisible(false);  
}

*Методът, който първи се изпълнява при показването на сцената*

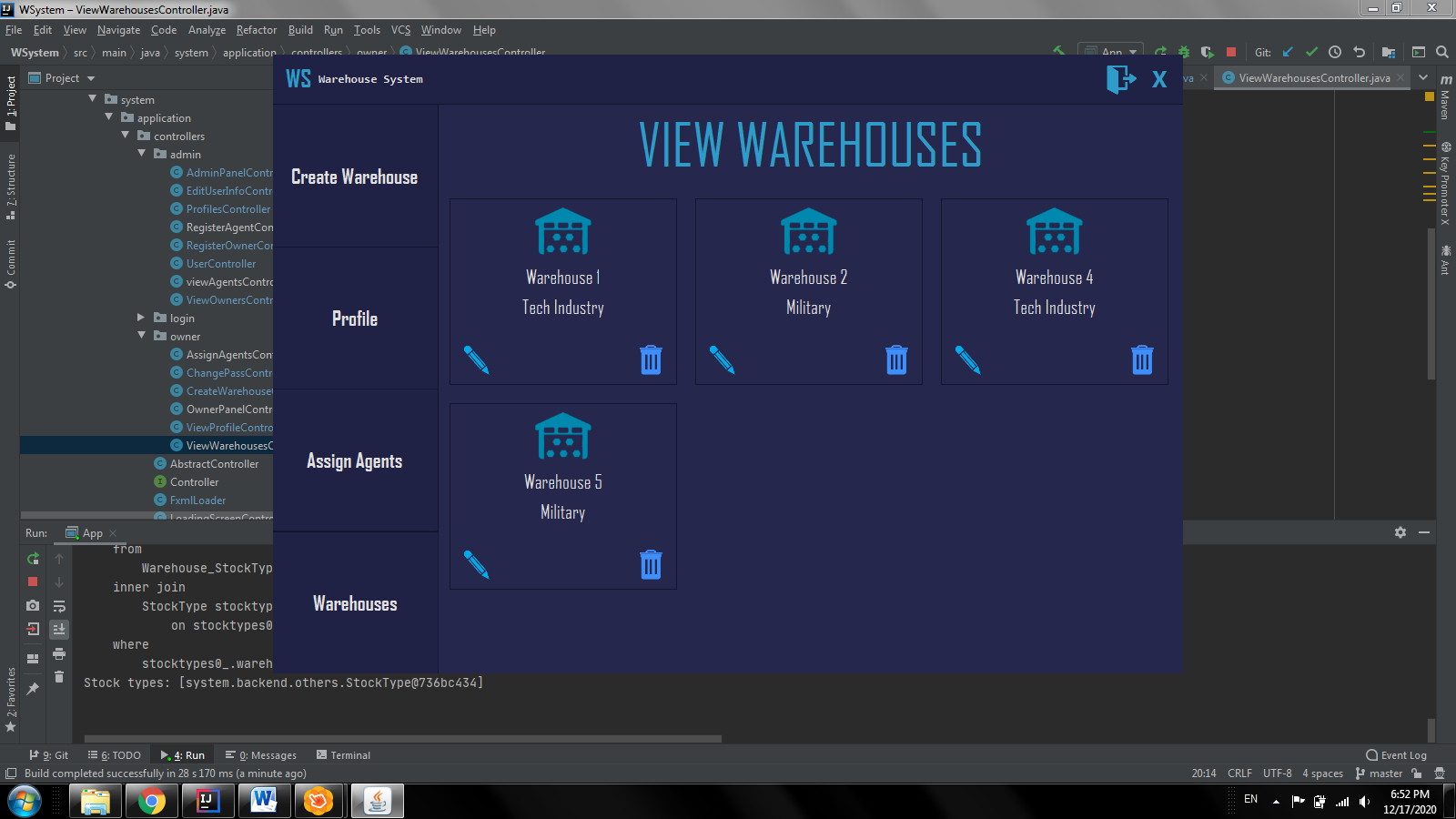
public void addToListView(){ // Method which adds items to the ListView for given index of ComboBox  
 int index = typeBox.getSelectionModel().getSelectedIndex();  
 if (index == 0) {  
 stockTypeView.getItems().clear();  
 foodsList.addAll("Vegetables", "Fruits", "Seeds");  
 stockTypeView.setItems(foodsList);  
 }  
 else if(index == 1){  
 stockTypeView.getItems().clear();  
 techList.addAll("Computers", "Smartphones", "Chargers");  
 stockTypeView.setItems(techList);  
 }  
 else{  
 stockTypeView.getItems().clear();  
 militaryList.addAll("Explosives", "Guns", "Military Equipment");  
 stockTypeView.setItems(militaryList);  
 }  
}

*Метод, който добавя елементи на списък към listView-a, който позволява да се изберат стоки*

public void createButtonAction(ActionEvent event) {  
 System.*out*.println("Create Button Clicked.");  
 consVbox.getChildren().clear(); // изчиства лейбълите в контейнера за грешки  
 successLabel.setVisible(false); // Манипулира видимостта  
 Hbox.setVisible(false); // Манипулира видимостта  
  
 data.clear();  
 cons.clear();  
  
 dataRetriever.getWarehouseDataFromController(this, data);  
  
 if (!validationService.allDataFilled(data.get("data"))) {  
 System.*out*.println("Please fill all of the required data!");  
 messageService.fillDataMessage(this, data.get("data"));  
 }  
 else {  
 if (!validationService.allDataFilled(data.get("stocktypes"))) {  
 System.*out*.println("Please select stock types!");  
 messageService.noStockTypesMessage(this);  
 }  
 else {  
 cons = owner.createWarehouse(data);  
  
 if (!cons.isEmpty()) { // ако има грешки   
 messageService.showMessages(this, cons, keys);  
 System.*out*.println("Yes cons is not emprty");  
 }else{ // Executes when cons.IsEmpty() => true  
 System.*out*.println("Warehouse created");  
 warehousesState(owner); // викане на метод, който проверява дали собственика има складове, ако има добавя бутон за преглед на складове  
 successLabel.setVisible(true); // Манипулира видимостта  
 }  
  
 }  
 }  
}

*Метод, който се изпълнява при натискането на “Create” бутон*

С) Клас ViewWarehousesController



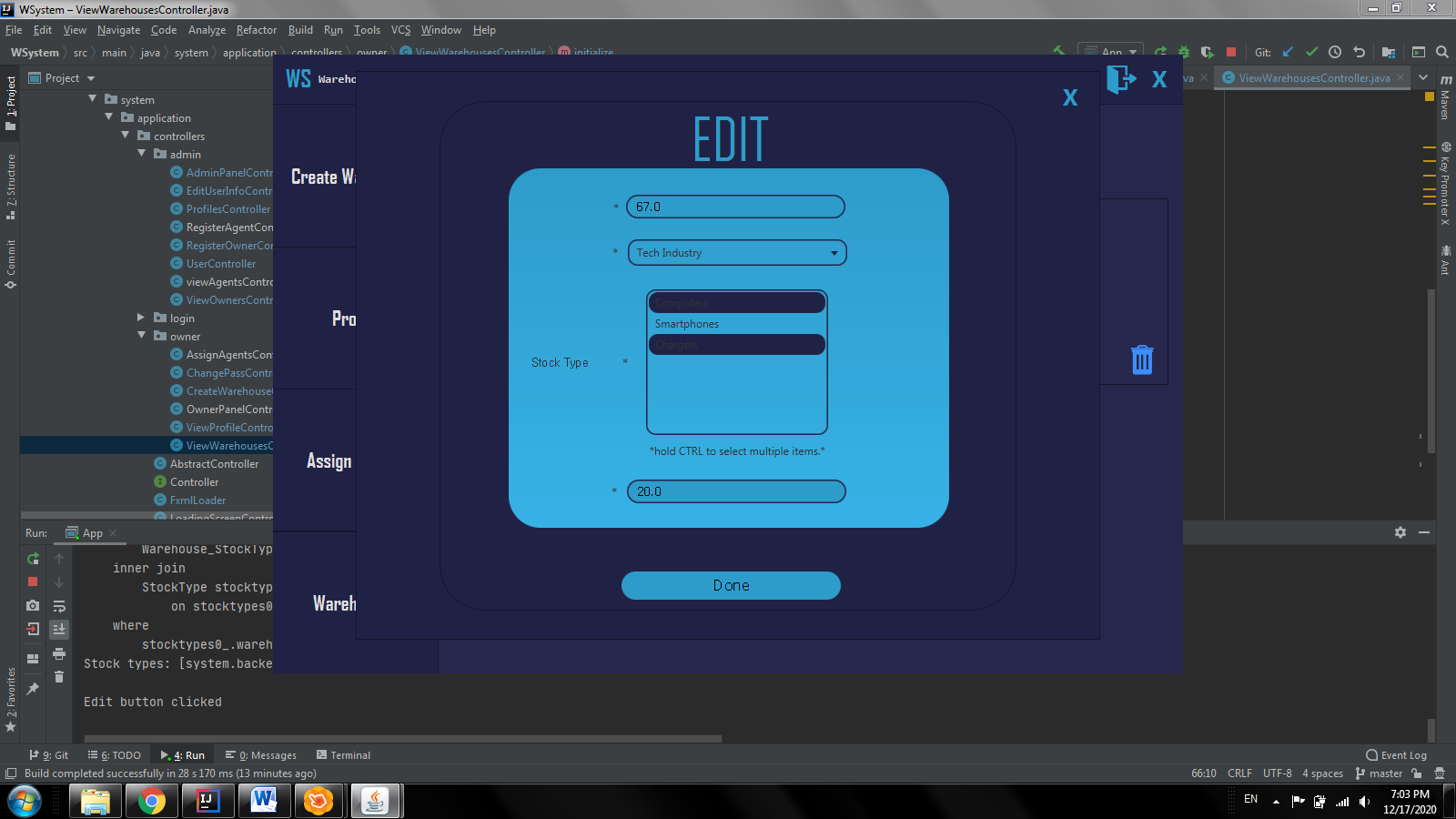
Дадената сцена показва всички складове на собственика, който се е логнал.

Този контролер има почти същата функционалност като ViewOwnersController и ViewAgentsController, но тук има само два лейбъла, който показват ID-то на склада и категорията му. Също така е променено изображението. Всеки склад има бутон за редактиране и изтриване на склад.

@Override  
public void initialize(URL url, ResourceBundle resourceBundle) {  
 OwnerDataHolder ownerDataHolder = OwnerDataHolder.*getInstance*();  
 owner = ownerDataHolder.getOwner();  
 wSystem = WSystem.*getInstance*();  
 int column = 0;  
 int row = 1;  
 try {  
 for (int i = 0; i < owner.getWarehouses().size(); i++) { // итерация през всички складове на собственика  
  
 FXMLLoader fxmlLoader = new FXMLLoader();  
 fxmlLoader.setLocation(getClass().getResource("/fxml/userFXML.fxml")); // задаване на път към fxml файла, който ще се използва  
  
  
 AnchorPane anchorPane = fxmlLoader.load();  
  
 UserController userController = fxmlLoader.getController();  
 userController.setWarehouse(owner.getWarehouses().get(i)); // викане на метод от друг контролер, който сетва лейбълите на всеки квадрат от gridPane-a  
  
  
 if (column == 3){  
 column = 0;  
 row++;  
 }  
  
 grid.add(anchorPane, column++, row); // добавяне към gridPane-a дадения квадрат на определена позиция според индекса на колоната и реда  
  
 GridPane.*setMargin*(anchorPane, new Insets(10));  
 grid.setMinWidth(Region.*USE\_COMPUTED\_SIZE*);  
 grid.setPrefWidth(Region.*USE\_COMPUTED\_SIZE*);  
 grid.setMaxWidth(Region.*USE\_PREF\_SIZE*);  
  
 grid.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);  
 grid.setMinHeight(Region.*USE\_COMPUTED\_SIZE*);  
 grid.setMaxHeight(Region.*USE\_PREF\_SIZE*);  
 }  
 }catch (IOException e) {  
 e.printStackTrace();  
 }  
}

*Методът, който първи се изпълнява при показването на сцената*

Когато се натисне бутона се редактиране се отваря следния прозорец:



Този прозорец използва контролера за създаване на склад(CreateWarehouseController). Като той замества бутона за създаване на склад с бутона “Done”, който update-ва селектирания склад. Също се добавя бутон за затваряне на прозореца и заглавния лейбъл, който беше “CREATE WAREHOUSE” се замества с “EDIT”.

При отварянето на прозореца се използва метод, който зарежда данните за склада в опеделените му контроли. Както се вижда и стоките в listView-a автоматично се селектират.

В метода за управление на „Done“ бутона има също скрипт за проверка за грешки. Ако няма грешки => актуализация на склада.

public void setData(Warehouse wh){ // (EDIT WAREHOUSE) sets fields and boxes to the values of warehouse  
 loaderPane.setStyle("-fx-background-color: #202245;\n" + "-fx-border-color: #14152b;\n" +  
 "-fx-border-width: 1px;"); // задаване на стил  
 titleLabel.setText("EDIT"); // промяна на текста на заглавния лейбъл  
 this.wh = wh; // присвояване на подаден лейбъл   
 createButton.setVisible(false); // скриване на бутона за създаване на склад  
 exitButton.setVisible(true); // показване на бутон за затваряне на прозорец  
 doneButton.setVisible(true); // показване на бутона за актуализация на склад  
 sizeField.setText(Double.*toString*(wh.getSize())); // задаване на текстовото поле с данните за размер на склада  
 typeBox.getSelectionModel().select(wh.getCategory()); // селектиране на comboBox-a   
 addToListView(); // зареждане на listView-a със стоки  
 for (StockType st: wh.getStockTypes()) {   
 stockTypeView.getSelectionModel().select(st.getType());// селектиране на стоките, които скадът притежава  
 }  
 temperatureField.setText(Double.*toString*(wh.getTemperature())); // задаване на текстовото поле  
}

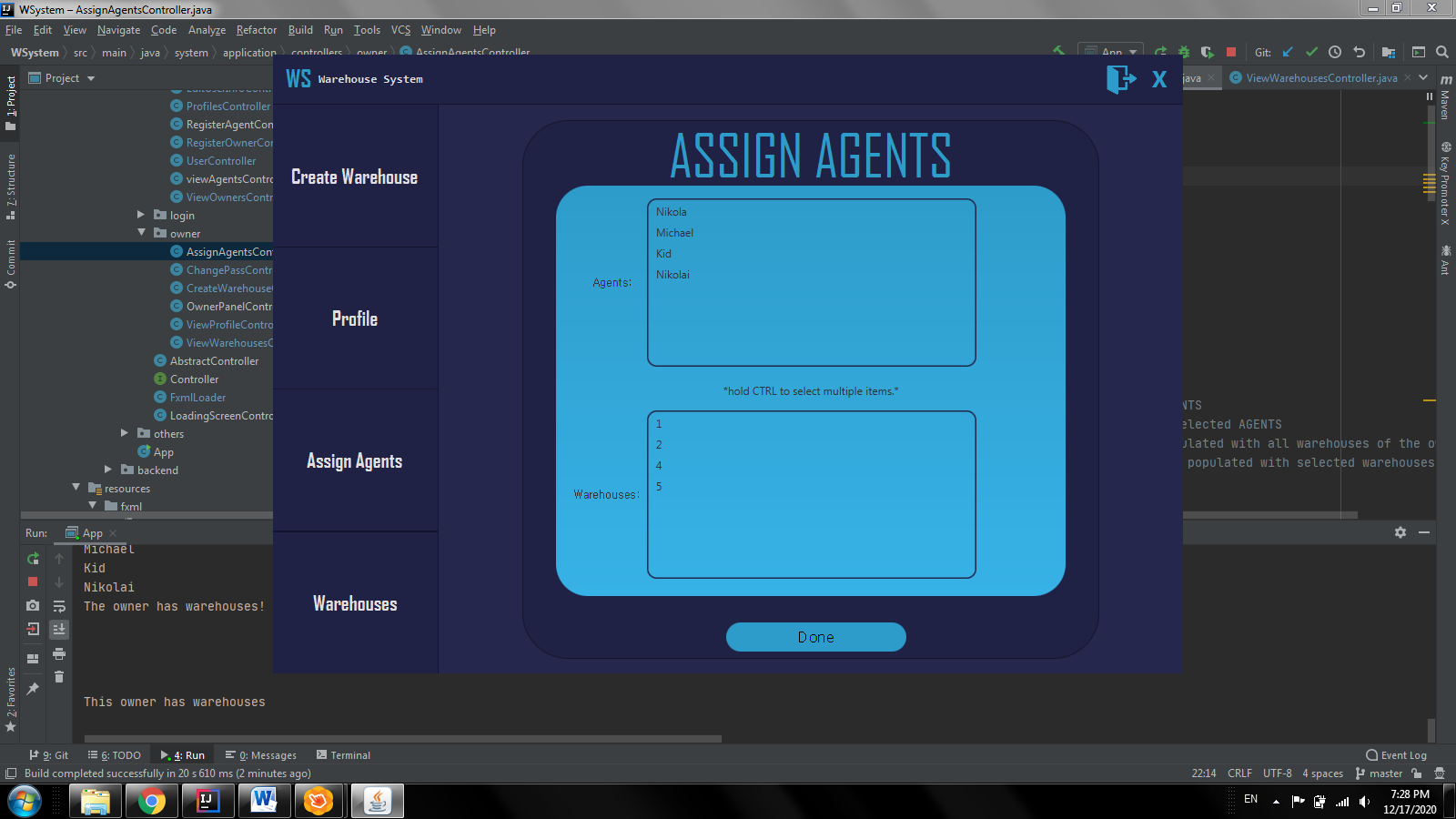
*Методът от CreateWarehouseController, който задава данните за склада в контролите*

public void doneButtonAction(ActionEvent event) { // HANDLES EDIT WAREHOUSE  
 consVbox.getChildren().clear();  
  
 oldData.clear();  
 data.clear();  
 cons.clear();  
  
 owner.getWarehouseData(wh, oldData);  
 dataRetriever.getWarehouseDataFromController(this, data);  
  
 if (!validationService.allDataFilled(data.get("data"))) {  
 System.*out*.println("Please fill all of the required data!");  
 messageService.fillDataMessage(this, data.get("data"));  
 }  
 else {  
 if (!validationService.allDataFilled(data.get("stocktypes"))) {  
 System.*out*.println("Please select stock types!");  
 messageService.noStockTypesMessage(this);  
 }  
 else{  
 cons = owner.updateWarehouse(wh, data);  
  
 if (!cons.isEmpty()) {  
 owner.setWarehouseData(wh, oldData);  
 messageService.showMessages(this, cons, keys);  
 } else {  
 System.*out*.println("Warehouse successfully updated!");  
 closeStage(event);  
 }  
 }  
 }  
 try {  
 FXMLLoader loader = new FXMLLoader(getClass().getResource("/fxml/userFXML.fxml"));  
 Parent root = loader.load();  
  
 UserController userController = loader.getController();  
 userController.transferMessage(wh.getCategory()); // transfer data between two controllers  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

*При натискане на doneButton*

Т) Клас AssignAgentsController

Когато се натисне „Assign Agent“ бутона се зарежда следната сцена:



Съдържа два listView-a и един бутон. В първия listView се извеждат всички агенти регистрирани в системата. В другия всички складове на собственика, който се е логнал. Когато се избере склад и собственик и се натисне бутона done се назначават складови агенти за избраните складове. Ако не са избрани складове и агенти от listView-овете излиза грешка.

@FXML  
private Button doneButton;  
@FXML  
private Label violationsLabel;  
@FXML  
private Label successLabel;  
@FXML  
private ListView<String> agentsView; // listView of agents  
@FXML  
private ListView<String> warehousesView; // listView of warehouses  
  
private ObservableList<String> agentsList = FXCollections.*observableArrayList*(); // ALL AGENTS  
private ObservableList<String> selectedAgentsList = FXCollections.*observableArrayList*(); // selected AGENTS  
private ObservableList<String> warehousesList = FXCollections.*observableArrayList*(); // populated with all warehouses of the owner  
private ObservableList<String> selectedWarehouses = FXCollections.*observableArrayList*(); // populated with selected warehouses  
private WSystem wSystem = WSystem.*getInstance*();  
private Owner owner;

*Декларация на основните възли в този клас*

@Override  
public void initialize(URL url, ResourceBundle resourceBundle) {  
 agentsView.getSelectionModel().setSelectionMode(SelectionMode.*MULTIPLE*); // Enables multiple choice for the ListView  
 warehousesView.getSelectionModel().setSelectionMode(SelectionMode.*MULTIPLE*); // Enables multiple choice for the ListView  
  
 OwnerDataHolder ownerDataHolder = OwnerDataHolder.*getInstance*();  
 owner = ownerDataHolder.getOwner();  
  
 if(wSystem.hasAgentProfiles()) { // Fills the listViews with their respective values if there are AGENTS  
 addAgentsToListView(); // добавя агенти към listView 1  
 outputWarehouses(); // добавя складове към listView 2  
 }  
 else  
 System.*out*.println("The system doesn't have agent profiles!");  
}

*Методът, който първи се изпълнява при показването на сцената*

public void addAgentsToListView(){ // adds agents to agentsView  
 System.*out*.println("The system has agent profiles!");  
 System.*out*.println("Showing agents here:");  
  
 for (Agent agent : wSystem.getAgents()) {  
 System.*out*.println(agent.getFirstname());  
 agentsList.add(agent.getFirstname()); // добавяне на всички агенти в списък  
 }  
 agentsView.setItems(agentsList); // добавя всички елементи от списък към listView  
  
}

*Методът, който се използва за добавяне на елементи към първия listView*

public void outputWarehouses(){ // add warehouses to warehousesView  
 List<Warehouse> warehouseList = owner.getWarehouses();  
 if(!warehouseList.isEmpty()) {  
 System.*out*.println("The owner has warehouses!");  
 for(Warehouse wh : warehouseList){  
 warehousesList.add(Long.*toString*(wh.getID())); // добавяне на всички складове в списък  
  
 System.*out*.println();  
 }  
 warehousesView.setItems(warehousesList); // добавя всички елементи от списък към listView  
 }  
 else System.*out*.println("The owner doesn't have warehouses!");  
}

*Методът, който се използва за добавяне на елементи към втория listView*

public void handleDoneButton(ActionEvent event) { // When done button is pressed  
 violationsLabel.setVisible(false);  
 successLabel.setVisible(false);  
  
 System.*out*.println("You clicked: " + doneButton.getText());  
 selectedAgentsList = agentsView.getSelectionModel().getSelectedItems(); // assigns selected elements to a list  
 selectedWarehouses = warehousesView.getSelectionModel().getSelectedItems(); // assigns selected elements to a list  
  
 if (!selectedAgentsList.isEmpty() && !selectedWarehouses.isEmpty()){ // checks if there are selected items in both listViews  
 successLabel.setVisible(true);  
 System.*out*.println(selectedAgentsList.toString());  
 System.*out*.println(selectedWarehouses.toString());  
 for (String o : selectedAgentsList)  
 System.*out*.println("Agent = " + o);  
 }else  
 violationsLabel.setVisible(true);  
}

*При натискане на doneButton*

1. **Работа с база данни и сървърни операции(Back-end)**
2. *Конфигурация*

Първата задача, която трябваше да изпълним беше да създадем “Maven” проект, който позволява сравнително лесно добавяне на библиотеки чрез деклариране на „dependency” във файла “pom.xml”. Библиотеките се достъпват от дистанционно хранилище. В нашия проект са добавените всички нужни библиотеки за работата с различните експлоатирани технологии.

*hibernate.cfg.xml:*

<!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>

<!-- JDBC Database connection settings -->

<property name="connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property name="connection.url">jdbc:mysql://localhost:3306/wsystem\_db?createDatabaseIfNotExist=true</property>

<property name="connection.username">root</property>

<property name="connection.password">root</property>

</session-factory>

</hibernate-configuration>

В този файл се задават потребителско име и парола на сървъра на базата данни, за да може да се прави връзка с нея. Задава се също името на сървъра и името на базата данни, като е добавен параметър за направа на базата данни автоматично, ако не съществува.

*persistence.xml:*

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

<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.2"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_2.xsd">

<!-- Define persistence unit -->

<persistence-unit name="my-persistence-unit" transaction-type="RESOURCE\_LOCAL">

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

<properties>

<property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>

<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost/wsystem\_db"/>

<property name="javax.persistence.jdbc.user" value="root"/>

<property name="javax.persistence.jdbc.password" value="root"/>

<property name="dialect" value="org.hibernate.dialect.MySQLDialect"/>

<property name="javax.persistence.validation.mode" value="none"/>

<property name="hibernate.show\_sql" value="true" />

<property name="hibernate.format\_sql" value="true" />

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

</properties>

</persistence-unit>

</persistence>

Тъй като в нашия проект използване спефицикацията JPA и като “provider” използваме Hibernate, сме добавили и “persistance.xml” файл. В него отново трябва да се зададат свойства, някои от които са същите като в по-горния файл. Едно от предимствата при използването на JPA е това, че може лесно да се премине от “Hibernate” към друг „provider”(например EclipseLink). Не е нужно почти никаква промяна на код, за да се случи това.

<persistence-unit name="my-persistence-unit" transaction-type="RESOURCE\_LOCAL">

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider> - Тук се задава като стойност “provider” за JPA спецификацията. Зададен е Hibernate.

Важни свойства са:

А) <property name="javax.persistence.validation.mode" value="none"/> - Това свойство служи за контрол на валидацията SQL операции. В случая стойността му е “none”, тъй като валидацията се извършва експлицитно, където е нужно чрез Hibernate Validator

Б) <property name="hibernate.show\_sql" value="true" /> - Това свойство служи за показване на SQL заявките, които Hibernate изпълнява.

В) <property name="hibernate.format\_sql" value="true" /> - Това свойство служи за форматиране на SQL заявките.

Г) <property name="hibernate.hbm2ddl.auto" value="update"/> - Това свойство служи за изпълняване на някаква операция, която се извършва при направата на “SessionFactory”. В случая използваме стойността „update”. С това се постига само и единствено актуализиране на базата данни, без изтриване и повторна направа.

1. *Детайлно обяснение на компонентите и тяхната функционалност*
2. *Configuration.class* – Функционалността на този клас се извиква при стартиране на проекта. Това е първото нещо, което се стартира. Под конфигурация се има предвид направа на базата данни, направа на “EntityManagerFactory” и “EntityManager”, инстанцииране на системата и инициализиране на базата данни(добавяне на стойности в таблиците). Цялото това нещо се постига с този фрагмент от код:

*public void configure(){*

*new Thread(() -> {*

*try {*

*// Creating the database*

*createDB();*

*LOGGER.info("Successfully finished creating the database!");*

*// Creating factory and manager*

*createFactory();*

*createManager();*

*LOGGER.info("Successfully finished creating factory and manager!");*

*// Creating the system*

*WSystem wSystem = WSystem.getInstance();*

*wSystem.initializeDB();*

*LOGGER.info("Successfully finished creating the system!");*

*LOGGER.info("Successfully finished the configuration of the application!");*

*} catch (Exception e) {*

*LOGGER.error("An error occurred during the configuration: " + e.getMessage());*

*e.printStackTrace();*

*}*

*}).start();*

*}*

За да се постигне изпълнението на тази функция заедно със зареждането на системата чрез “LoadingScreenController” и неговия прозорец, се използва „Multithreading”. Дефинирана е нова нишка, с помоща на която се постига конфигурация по време на зареждането на програмата.

В този клас е използван “Singleton Pattern”, тъй като класът е нужно да се инстанциира само веднъж и само ако е нужно да се извиква инстанциирания обект.

public static WSystem getInstance(){

if(wsystem == null)

wsystem = new WSystem();

return wsystem;

}

За да се постигне отделяне на слоя “Persistence“ от останалите слоеве е използван “DAO(Data Access Object) шаблон“.

1. *DAO интерфейс:*

*public interface DAO<T, V> {*

*void save(T object);*

*void delete(T object);*

*T findByID(Class<T> c, Long id);*

*void deleteByID(Class<T> c, Long id);*

*void update(T object);*

*List<T> selectAll(Class<T> c);*

*T findBy2Values(Class<T> c, String column1, String column2, V value1, V value2);*

*T findBy1Value(Class<T> c, String column1, V value);*

*T findBy1ValueExcept(Class<T> c, String column1, V value, long ignoreThisID);*

*void updateColumn(Class<T> c, String column, V newValue, V oldValue);*

*}*

В този интерфейс са вградени всички операции свързани с базата данни. За да се постигне драстично намаляване на обема на кода, интерфейсът е “Generic” с два типови параметъра. Функционалността на интерфейса се съсредоточава върху само и единствено върху подадените като параметър типове, с които трябва да работи. Единият от тях “T” приема типове на различни обекти от домейн модела на програмата. С помощта на този параметър изпълненето на заявките се насочва към конкретна таблица. Вторият от тях е “V”. Този параметър служи за типизиране на стойностите, с които ще се изврършва дадена операция. Тези стойности могат да са от тип “String”, „Double”, “Integer” и т.н.

1. *class MainDAO<T, V> implements DAO<T, V>:*

Това е класът, който имплементира функционалността на интерфейса. Тук са дефинирани всички функции, които служат за нужните операции с базата данни. Примерна заявка, която служи за вкарване на обект в базата данни е:

*public void save(T object){*

*try {*

*manager.getTransaction().begin();*

*manager.persist(object);*

*manager.getTransaction().commit();*

*} catch(Exception e){*

*e.printStackTrace();*

*System.out.println("Exception occurred during saving.”);*

*}*

*}*

Друга примерна заявка, която служи за селектиране на всички обекти в дадена таблица и тяхното връщнане като списък е:

public List<T> selectAll(Class<T> c){

CriteriaBuilder cb = manager.getCriteriaBuilder();

CriteriaQuery<T> q = cb.createQuery(c);

Root<T> r = q.from(c);

q.select(r);

TypedQuery<T> query = manager.createQuery(q);

List<T> results = query.getResultList();

return results;

}

В нашият проект използваме “CriteriaQuery API“ вместо JPQL или HQL, за да се избегне допускането на грешки при писане на заявките. Счетохме за по-удобно.

1. *class AuthorizationService<T>* - Този клас служи за автентикация там, където е нужно. В нашия проект имаме само една единствена функция “authorizeLogin(…)”, която служи за извършване на автентикация при влизане в профил.
2. *Class ValidationService* – Този клас служи за валидация там, където е нужно.

Ключов момент е нуждата от валидация при някои операции с базата данни. За да се извърши някаква операция, която изисква валидация на данните(регистрация или актуализация на профил), въведените данни трябва да се валидират експлицитно. Ако въведените данни са валидни, операцията продължава своето изпълнение. Ако, обаче, въведините данни са невалидни, операцията не продължава. В такава ситуация трябва потребителят да бъде уведомен за грешките и че данните, които е въвел трябва да се поправят, за да завърши операцията успешно. За целта се извиква функцията “validate(Object object)”. Тази функция връща колекция от тип “Map”, която съдържа като ключ имената на грешните елементи и като стойност - грешките допуснати по време на валидацията. Тези грешки се изобразяват на екрана на потребителя.

1. *class CryptoService* – Този клас служи за криптиране и декриптиране на сензитивни данни(парола). Имплементираният метод е „AES”. Избрахме този метод, защото е един от широко използваните. Криптирането, обаче, е изключено в нашата програма, защото за извършване на криптиране и декриптиране трябва да се използва един и същ ключ, а нов ключ се генерира наново след всяко стартиране на програмата, което следователно означава, че декриптиране на стари данни не може да се осъществи.
2. *class OwnerDataHolder –* Този клас се използва за прехвърляне на данни за собственик от един контролер в друг.
3. *interface MainProfile –* Този интерфейс съдържа гетъри и сетъри за идентификационен номер, първо име, фамилно име, потребителско име и парола. Имплементират го класовете за администратор, агент и собственик.
4. *interface SecondaryProfile –* Този интерфейс съдържа гетъри и сетъри за имейл адрес и телефон. Имплементират го класовете за агент и собственик.
5. *abstract class AbstractMainProfile implements MainProfile –* Този абстрактен клас ще бъде наследен от актьорите в системата – Администратор, Агент и Собственик. Анотациите @Pattern и @Size служат за задаване на ограничения на свойствата.
6. *class Owner extends AbstractMainProfile implements SecondaryProfile –* Този клас е за собственика на склад. Ключов момент в този клас е:

*@OneToMany(mappedBy = "owner", cascade = CascadeType.ALL)*

*private List<Warehouse> warehouses = new ArrayList<>();*

Тъй като собственикът притежава складове и един собственик може да притежава много складове едновременно, връзката между класа за собственик и класа за склад е едно към много. Следователно индентификаторът на всеки собственик, притежаващ склад, ще влезе като чужд ключ в таблицата за складове. За стойността на каскадност е избрана “CascadeType.ALL”. Следователно, когато се извърши някаква операция(зависи каква) със собственик в базата данни, резултатите може да засегнат таблицата за склад.

1. *class Agent extends AbstractMainProfile implements SecondaryProfile –* Този клас е за агента.
2. *class Admin extends AbstractMainProfile –* Този клас е за администратора.
3. *class ProfileManager<T> -* Този клас служи за извършване на операции с профили. Класът е „Generic”, защото може да служи за операции с профил, без значение дали е агент или собственик. Това намалява значително кода. Пример за функция, която служи за направата на профил е:

public <T extends MainProfile & SecondaryProfile> Map<String, Set<String>> createProfile(Class<T> c, Map<String, String> data) {

try {

T object = c.getDeclaredConstructor().newInstance();

setProfileData(object, data);

ValidationService validationService = ValidationService.getInstance();

Indicator indicator = new Indicator();

indicator.setValidationIndicator(c);

validationService.setIndicator(indicator);

Map<String, Set<String>> constraints = validationService.validate(object);

if(constraints.isEmpty()) {

if (object.getClass() == Owner.class) {

DAO<Owner, String> ownerDAO = new MainDAO<>();

ownerDAO.save((Owner) object);

}

else if (object.getClass() == Agent.class) {

DAO<Agent, String> agentDAO = new MainDAO<>();

agentDAO.save((Agent) object);

}

}

return constraints;

} catch (Exception e){

System.out.println("Something happened during creating the profile.");

e.printStackTrace();

return null;

}

Декларацията на функцията съдържа <T extends MainProfile & SecondaryProfile>, което означава, че само обекти от класове, които имплементират MainProfile и SecondaryProfile, могат да бъдат използвани от функцията. Ако това отсъства в декларацията, ще доведе до проблеми и грешки, защото не могат да се използват сетърите, нужни във извиканата второстепенна функция:

public <T extends MainProfile & SecondaryProfile> void setProfileData(T object, Map<String, String> data){

object.setFirstname(data.get("firstname"));

object.setLastname(data.get("lastname"));

object.setUsername(data.get("username"));

object.setPassword(data.get("password"));

object.setEmailAddress(data.get("email"));

object.setPhoneNumber(data.get("phone"));

}

Тук вече компилаторът ще знае, че обектът е от тип, който наследява MainProfile и SecondaryProfile. Следователно сетърите ще могат да бъдат използвани.

1. *@interface MyUnique –* Дефинира „custom“ анотация, която в този случай представлява ограничение. Тъй като е нужно при регистрация или актуализация на профил да се извърши валидация, която включва проверка на някои данни(имей адрес, телефон, потребителско име) дали вече съществуват, сме дефинирали ограничение за тази цел и собствен валидатор, който извършва тази проверка.

*@Constraint(validatedBy = UniqueValidator.class) –* Показва, че валидаторът който ще извършва проверката се нарича “UniqueValidator.class”.

*String message() default " is already taken"* – Това служи за дефиниране на съобщението, което ще се връща, ако проверката завърши с негативен резултат. В някои части на програмата, където се извършва споменатата по-горе тип валидация и тя върне негативен резултат, към това съобщение в началото се конкатенира и нарушения елемент(имейл адрес, телефон, потребителско име).

1. *class UniqueValidator implements ConstraintValidator<MyUnique, String> -* Това е собственоръчно-създаденият валидатор за извършването на „unique” валидацията. При направа на нов профил или актуализиране на такъв се проверява дали въведените съвършени данни съществуват някъде в профилните таблици. За тази цел се изпълняват заявки за търсене към тези талбици. Ако такива данни съществуват, проверката завършва с негативен резултат.
2. *class Indicator –* Това е помощен клас за класът „UniqueValidator”. Преди всяко извършване на валидация се слага индикатор, който показва от какъв тип е обектът, който се валидара в момента. В противен случай валидаторът няма да проработи.
3. *class Warehouse –* Клас за складовете. Ключови моменти тук са:

*@ManyToMany(cascade = CascadeType.PERSIST)*

*@JoinTable(name = "Warehouse\_StockType",*

*joinColumns = {@JoinColumn(name = "warehouse\_id")},*

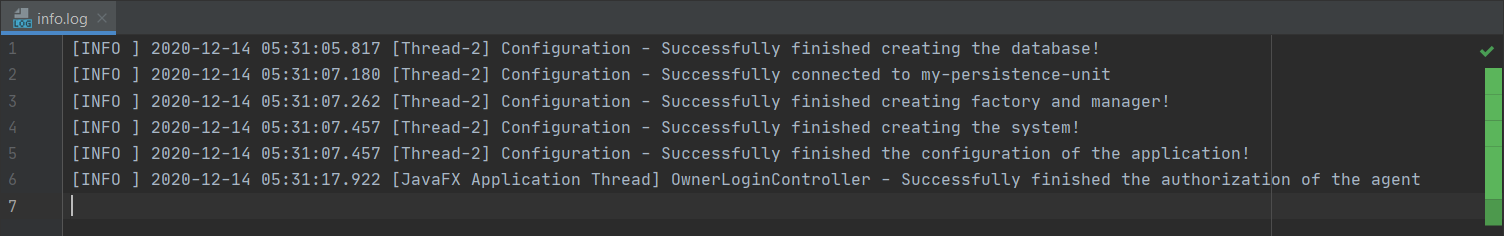
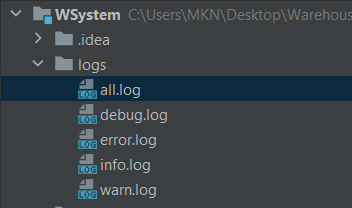
*inverseJoinColumns = {@JoinColumn(name = "stockType\_id")})*

*private Set<StockType> stockTypes;*

Един склад може да съдържа много видове стоки и обратно – един вид стока може да бъде складирана в много складове. Затова връзката между склад и тип стока е много към много.

1. *class StockType –* Това е класът за тип стока
2. *class Wsystem –* Това е класът за самата система
3. *class DataRetriever* – Това е клас, служещ за взимане на данните от контролер, когато е нужно. Във функциите като параметър влиза контролер.
4. *class MessageService –* Това е клас, служещ за изобразяването на грешките след извършване на валидация, която връща негативен резултат. За тази цел, в съответните функции трябва да влиза като параметър контролерът. За да се избегне повторяемостта на код, тъй като контролерите са от различни типове, сме дефинирали интерфейс “Controller”, в който са добавени нужните за използване функции. Тъй като няколко контролера ще имплементират този интерфейс и трябва да се избегне задължителната имплементация на ненужни функции във всеки един от тези контролери, сме дефинирали един абстрактен клас „class AbstractController implements Controller“. В този клас е добавена просто някаква функционалност на тези функции, която не е от никакво значение. Контролерите, които ще бъдат използвани от “MessageService” наследяват този абстрактен клас. По този начин се постига експлоатиране на функциите на даден контролер, влязъл като параметър в някоя функция на класа “MessageService”.
5. Реализация на модул за регистриране на събития(Log4J2)

Събитията се регистрират на няколко нива, както  
е показано на снимката в дясно(all, debug, error, info  
warn).



Примерен файл(info.log – регистрират се само събития от тип „info”)

1. *Тестови резултати*

*А)* Тест за регистрация на потребител от админ:

@Test

@Ignore

public void registerUserTest(){

Configuration configuration = Configuration.getInstance();

configuration.configure();

Map<String, String> data = new HashMap<>();

data.put("firstname", "Ivan");

data.put("lastname", "Tomov");

data.put("username", "vankata");

data.put("password", "Asd123\_123");

data.put("email", "vankata@abv.bg");

data.put("phone", "44444444444");

ProfileManager<Owner> profileManager = new ProfileManager<>();

Map<String, Set<String>> cons = new LinkedHashMap<>();

cons = profileManager.createProfile(Owner.class, data);

if(!cons.isEmpty()){

Set<Map.Entry<String, Set<String>>> entries = cons.entrySet();

for(Map.Entry<String, Set<String>> entry : entries){

System.out.println("\n" + entry.getKey() + ": ");

for(String message : entry.getValue())

System.out.println(message);

}

fail();

}

}

Ако данните при входа са валидни, тестът ще върне положителен резултат. Ако, обаче данните са невалидни, тестът връща отрицателен резултат и ще принтира грешките в конзолата. В случая по-горе, тестът ще върне положителен резултат.

Ако, обаче въведем данните по този начин:

data.put("firstname", "321AA\_@");

data.put("lastname", "Tomov");

data.put("username", "van—kata");

data.put("password", "zzz");

data.put("email", "vankata@abv.bg");

data.put("phone", "44444444444");

Тук, тестът ще върне отрицателен резултат, защото първото име, потребителското име и паролата са невалидни. Тестът също ще принтира грешките, които са допуснати при входа.

Б) Тест за актуализация на профил от админ:

@Test

@Ignore

public void adminUpdateProfilesTest(){

Configuration configuration = Configuration.getInstance();

configuration.configure();

DAO<Owner, String> dao = new MainDAO<>();

Owner owner = dao.findByID(Owner.class, 1L);

Map<String, String> newData = new HashMap<>();

newData.put("firstname", "!#@AA33");

newData.put("lastname", "Stefanov");

newData.put("username", "\_\_--");

newData.put("password", "asa");

newData.put("email", "kiro@abv.bg");

newData.put("phone", "222222222222");

ProfileManager<Owner> profileManager = new ProfileManager<>();

Map<String, Set<String>> cons = new LinkedHashMap<>();

cons = profileManager.updateProfile(owner, Owner.class, newData);

if(!cons.isEmpty()){

Set<Map.Entry<String, Set<String>>> entries = cons.entrySet();

for(Map.Entry<String, Set<String>> entry : entries){

System.out.println("\n" + entry.getKey() + ": ");

for(String message : entry.getValue())

System.out.println(message);

}

fail();

}

}

Тук, в този случай тестът ще фалира, защото данните, които се въвеждат не са коректни. Те отново трябва да са съобразени с ограниченията.

В) Тест за направа на склад от собственик:

@Test

@Ignore

public void updateWarehouseTest(){

Configuration configuration = Configuration.getInstance();

configuration.configure();

DAO<Owner, String> ownerDAO = new MainDAO<>();

Owner owner = ownerDAO.findByID(Owner.class, 1L);

Warehouse warehouse = owner.getWarehouses().get(0);

// DAO<Warehouse, String> warehousedao = new MainDAO<>();

// Warehouse warehouse = warehousedao.findByID(Warehouse.class, 1L);

Map<String, Map<String, String>> newData = new HashMap<>();

newData.put("data", new HashMap<>());

newData.get("data").put("size", "23432532453245234");

newData.get("data").put("temperature", "324523452345234");

newData.get("data").put("category", "Tech Industry");

newData.put("stocktypes", new HashMap<>());

newData.get("stocktypes").put("stocktype 1", "Computers");

newData.get("stocktypes").put("stocktype 2", "Smartphones");

Map<String, Set<String>> cons = new LinkedHashMap<>();

cons = owner.updateWarehouse(warehouse, newData);

if(!cons.isEmpty()){

Set<Map.Entry<String, Set<String>>> entries = cons.entrySet();

for(Map.Entry<String, Set<String>> entry : entries){

System.out.println("\n" + entry.getKey() + ": ");

for(String message : entry.getValue())

System.out.println(message);

}

fail();

}

}

По аналогичен начин са нещата и с този тест. Отново данните за склад трябва да са валидни. В противен случай, тестът фалира.