**14-01-2024**

**Politechnika Krakowska im. Tadeusza Kościuszki**

**Wydział Inżynierii Elektrycznej i Komputerowej**

**Autorzy:**

**Sylwia Jaworska**

**Grzegorz Listwan**

**Krzysztof Pacura**

**Przedmiot:**

**Programowanie w Języku Java**

**Projekt**

**System do zarządzania zasobami(ludzkimi, parkiem maszyn, zleceniami, magazynem) w zakładzie przemysłowym.**

1. **Cel i zakres projektu**

Celem projektu jest stworzenie aplikacji wspierającej zarządzanie pracą produkcyjną. Aplikacja ta może być rozszerzona o dodatkowe moduły takie jak magazyn, utrzymanie ruchu, zakupy, logistyka czy HR tworząc razem kompleksowy system do zarządzania wszystkimi zasobami w zakładzie przemysłowym.

Zakres projektu obejmuje moduł produkcyjny. Aplikacja ta umożliwi efektywne zarządzanie zasobami materialnymi: maszynami, sprzętem, komponentami i półproduktami a także na optymalizację czasu procesu produkcji dzięki dynamicznemu przydzielaniu zadań. Pozwoli na generowanie danych potrzebnych do analiz, planowania czy korekt planów zarówno w obrębie działu produkcyjnego jak i działów współpracujących. Ponadto pozwoli na lepszy przepływ informacji.

Aplikacja będzie wsparciem i narzędziem dla pracowników na każdym stanowisku: pracownik produkcji, lider, kierownik i administrator.

Pracownik po zalogowaniu będzie automatycznie otrzymywać dostępne zlecenia, będzie mieć możliwość wybrania oraz zakończenia zlecenia podając ilość wyprodukowanego towaru. System będzie automatycznie przypisywał dostępne zlecenia pracownikom z uwzględnieniem ich uprawnień, maszyn przypisanych do danego etapu oraz dostępności sprzętu. Lider będzie miał możliwość zarządzania zleceniami, pracownikami i maszynami na danej części zakładu. Dyrektor oprócz funkcji lidera, będzie uprawniony do zarządzania pracownikami, zleceniami i maszynami na poziomie całego zakładu. Administrator pełni rolę nadzorczą nad systemem, zapewniając jego stabilność, bezpieczeństwo i zgodność z wymaganiami zakładu przemysłowego.

Wszystkie funkcje systemu będą zaimplementowane w oparciu o paradygmat programowania obiektowego, co ułatwi rozwój, zrozumienie i utrzymanie kodu.

1. **Główne funkcje systemu**

1. **Zarządzanie Zleceniami:**

- Przyjmowanie zleceń

- Wystawianie zleceń do produkcji

- Zakończenie zleceń

2. **Zarządzanie zasobami ludzkimi:**

- Przydzielanie pracowników do kategorii zleceń

- Hierarchia ról i uprawnienia

- Zarządzanie dostępnością pracowników

3**. Zarządzanie maszynami:**

- Przypisywanie maszyn do etapów produkcji

- Dodawanie, usuwanie i wyłączanie Maszyn

4. **Monitorowanie postępu i statystyki**

- Monitorowanie bieżącego postępu

- Generowanie statystyk wydajności

- Śledzenie czasu realizacji zleceń

5. **Konfiguracja priorytetów zleceń:**

- Dynamiczna zmiana priorytetów zleceń

- Dostosowywanie planu produkcji do priorytetów

6. **Bezpieczeństwo i uprawnienia:**

- Klarowne definiowanie uprawnień dla ról użytkowników

- Zarządzanie kontami użytkowników

1. **Diagram Klas**

Obraz zawierający tekst, zrzut ekranu, Czcionka, Równolegle

Opis wygenerowany automatycznie

1. **Diagram ERD**

Obraz zawierający tekst, zrzut ekranu, Czcionka, pismo odręczne

Opis wygenerowany automatycznie

1. Projekt który został zaimplementowany

Zaimplementowano projekt z wykorzystaniem wielowątkowości w oparciu o wątki Runnable, wykorzystano połączenie TCP między serwerem a klientami oraz zastosowano bazę danych MariaDB a do utworzenia interfejsu użytkownika wykorzystano javaFX. Poniżej zostanie opisany sposób implementacji i oraz wyniki działania aplikacji.

1. Implementacja bazy danych z wykorzystaniem XAMPP i MariaDB zgonie z przedstawionym wyżej diagramem ERD.

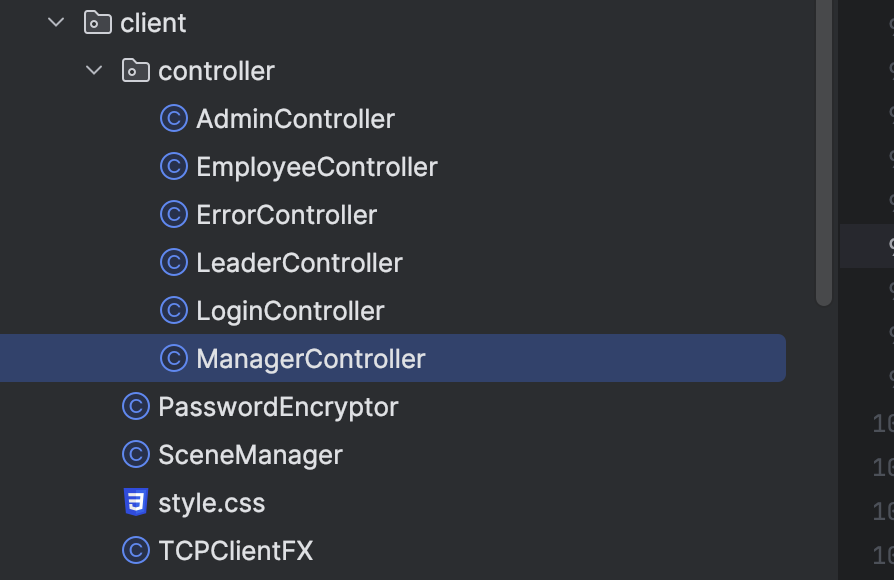


Baza danych została zaimplementowana tak aby możliwa była realizacja wszystkich niezbędnych elementów projektu. W bazie tej zastosowano klucze główne do unikatowej identyfikacji rekordów, klucze obce do powiązania relacji w bazie danych między tabelami. Wykorzystano również różne typy relacji: jeden-do-jeden, jeden-do-wielu oraz wiele-do-wielu co spowodowało konieczność stworzenia tabel łączących tabele w tej ostatniej relacji. Następnie w bazie danych wprowadzono niezbędne dane początkowe bez których aplikacja by nie działała.

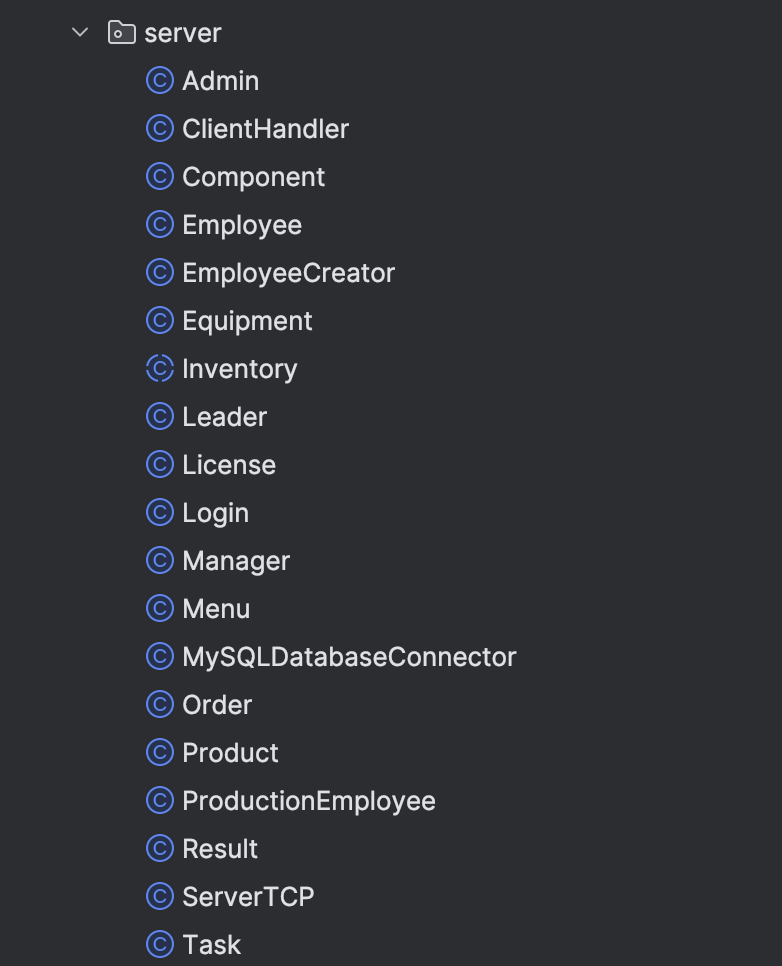
1. Implementacja aplikacji.

Aplikacja została podzielona na 3 główne pakiety.

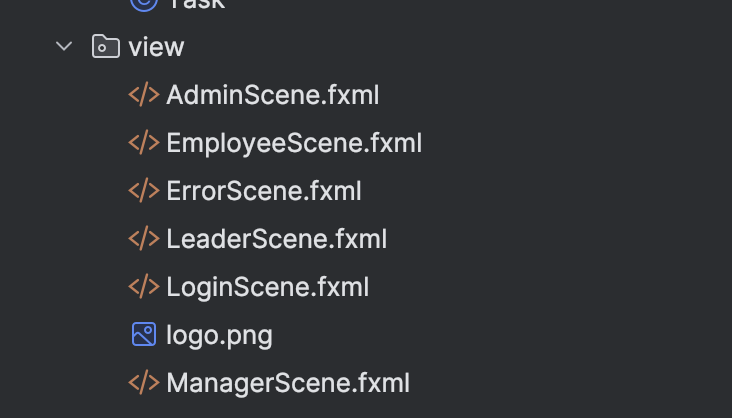
- Client – pakiet w którym znajduje się implementacja części klienckiej w której możemy wyróżnić dodatkowy pakiet ***controler***  do przechowywania klas odpowiedzialnych za kontrolery widoków. Struktura tego pakietu przedstawiona jest poniżej:



- Server – pakiet w którym znajduje się implementacja części serwerowej aplikacji. Struktura poniżej:



- View – pakiet w którym znajdują się pliki fxml odpowiedzialne za widoki klienckie aplikacji. Struktura i zawartość poniżej:



Opis implementacji rozpoczniemy od strony klienckiej

Klasa PasswordEncryptor

package client;  
  
import java.nio.charset.StandardCharsets;  
import java.security.MessageDigest;  
import java.security.NoSuchAlgorithmException;  
  
public class PasswordEncryptor {  
  
 public static String encryptPassword(String password) {  
 try {  
 // Tworzymy instancję obiektu MessageDigest z algorytmem SHA-256  
 MessageDigest digest = MessageDigest.*getInstance*("SHA-256");  
  
 // Przekazujemy hasło do obiektu MessageDigest z użyciem UTF-8  
 byte[] hash = digest.digest(password.getBytes(StandardCharsets.*UTF\_8*));  
  
 // Konwertujemy wynikowy bajtowy hash do postaci szesnastkowej (hex)  
 StringBuilder hexString = new StringBuilder();  
 for (byte b : hash) {  
 String hex = Integer.*toHexString*(0xff & b);  
 if (hex.length() == 1) {  
 hexString.append('0');  
 }  
 hexString.append(hex);  
 }  
  
 // Zwracamy zaszyfrowane hasło  
 return hexString.toString();  
  
 } catch (NoSuchAlgorithmException e) {  
 e.printStackTrace();  
 // Tutaj możesz obsłużyć wyjątek odpowiednio do twojej aplikacji  
 return null;  
 }  
 }  
}

Klasa ta zawiera metodę ***encryptPassword,*** która służy do szyfrowania haseł za pomocą algorytmu SHA-256. Oto opis tej metody:

1. MessageDigest: Wykorzystuje klasę `MessageDigest`, aby utworzyć instancję obiektu z algorytmem SHA-256, co jest bezpiecznym algorytmem haszowania.

2. Szyfrowanie hasła: Przekazuje hasło do obiektu `MessageDigest` przy użyciu kodowania UTF-8, a następnie uzyskuje z tego wynikowy bajtowy hash.

3. Konwersja do postaci szesnastkowej: Konwertuje bajtowy hash na postać szesnastkową (hex). Każdy bajt jest zamieniany na dwucyfrowy zapis szesnastkowy.

4. StringBuilder: Używa `StringBuilder` do budowania szesnastkowej reprezentacji zaszyfrowanego hasła, z uwzględnieniem poprawnej długości dla każdego bajtu.

5. Zwracanie wyniku: Zwraca zaszyfrowane hasło jako string w postaci szesnastkowej.

6. Obsługa wyjątku: W przypadku braku obsługi algorytmu SHA-256 (co jest mało prawdopodobne), metoda wyświetla błąd (StackTrace) i zwraca null.

Klasa SceneManager

package client;  
  
import client.controller.\*;  
import javafx.fxml.FXMLLoader;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.stage.Stage;  
import server.Employee;  
import server.Equipment;  
import server.Order;  
import server.Task;  
  
  
import java.io.IOException;  
import java.util.List;  
  
public class SceneManager {  
 private Stage primaryStage;  
 public SceneManager(Stage primaryStage){  
 this.primaryStage = primaryStage;  
 }  
 public void showLoginScene(TCPClientFX tcpClientFX) {  
 try {  
 FXMLLoader loader = new FXMLLoader(getClass().getResource("/view/LoginScene.fxml"));  
 LoginController loginController = new LoginController(tcpClientFX);  
 loader.setController(loginController);  
 Parent root = loader.load();  
 primaryStage.setScene(new Scene(root));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void showAdminScene(TCPClientFX tcpClientFX, Employee employee, List<Employee> employees) {  
 try {  
 FXMLLoader loader = new FXMLLoader(getClass().getResource("/view/AdminScene.fxml"));  
 AdminController adminController = new AdminController(tcpClientFX,employee, employees);  
 loader.setController(adminController);  
 Parent root = loader.load();  
 primaryStage.setScene(new Scene(root));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void showEmployeeScene(TCPClientFX tcpClientFX, Employee employee, List<Task> tasks) {  
 try {  
 FXMLLoader loader = new FXMLLoader(getClass().getResource("/view/EmployeeScene.fxml"));  
 EmployeeController employeeController = new EmployeeController(tcpClientFX,employee, tasks);  
 loader.setController(employeeController);  
 Parent root = loader.load();  
 primaryStage.setScene(new Scene(root));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void showLeaderScene(TCPClientFX tcpClientFX, Employee employee) {  
 try {  
 FXMLLoader loader = new FXMLLoader(getClass().getResource("/view/LeaderScene.fxml"));  
 LeaderController leaderController = new LeaderController(tcpClientFX,employee);  
 loader.setController(leaderController);  
 Parent root = loader.load();  
 primaryStage.setScene(new Scene(root));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void showManagerScene(TCPClientFX tcpClientFX, Employee employee, List<Order> orders, List<Task> tasks, List<Employee> employees, List<Equipment> equipmentList) {  
 try {  
 FXMLLoader loader = new FXMLLoader(getClass().getResource("/view/ManagerScene.fxml"));  
 ManagerController managerController = new ManagerController(tcpClientFX,employee,orders, tasks,employees,equipmentList);  
 loader.setController(managerController);  
 Parent root = loader.load();  
 primaryStage.setScene(new Scene(root));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 public void showErrorScene(TCPClientFX tcpClientFX) {  
 try {  
 FXMLLoader loader = new FXMLLoader(getClass().getResource("/view/ErrorScene.fxml"));  
 ErrorController errorController = new ErrorController(tcpClientFX);  
 loader.setController(errorController);  
 Parent root = loader.load();  
 primaryStage.setScene(new Scene(root));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 public void closeApp(){  
 primaryStage.close();  
 }  
}

SceneManager to klasa odpowiedzialna za zarządzanie widokami w interfejsie użytkownika. Przyjmuje obiekt ***Stage*** w konstruktorze, reprezentujący główne okno aplikacji. Metody tej klasy, takie jak showLoginScene, showAdminScene czy showManagerScene, pozwalają na przejście między różnymi widokami aplikacji w zależności od uprawnień użytkownika. Każda z tych metod tworzy obiekt FXMLLoader do ładowania plików FXML reprezentujących poszczególne sceny, przypisuje do nich odpowiednich kontrolerów, a następnie ustawia nowy widok w głównym oknie. Ta struktura umożliwia dynamiczne przełączanie się między różnymi interfejsami w trakcie działania aplikacji.

Klasa TCPClientFX

package client;  
  
import javafx.application.Application;  
import javafx.fxml.FXMLLoader;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.stage.Stage;  
import client.controller.LoginController;  
import server.Employee;  
import server.Equipment;  
import server.Order;  
import server.Task;  
  
import java.io.IOException;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.io.PrintWriter;  
import java.net.Socket;  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Scanner;  
  
public class TCPClientFX extends Application {  
 private String username;  
 private String password;  
 private SceneManager sceneManager;  
 private Socket socket;  
 private Scanner in;  
 private PrintWriter out;  
 private ObjectInputStream objectInputStream;  
 private ObjectOutputStream objectOutputStream;  
  
  
  
 @Override  
 public void start(Stage primaryStage) throws InterruptedException {  
 try {  
  
  
 FXMLLoader loader = new FXMLLoader(getClass().getResource("/view/LoginScene.fxml"));  
 LoginController loginController = new LoginController(this);  
 loader.setController(loginController);  
  
 Parent root = loader.load();  
 Scene scene = new Scene(root);  
 scene.getStylesheets().add(getClass().getResource("style.css").toExternalForm());  
 primaryStage.setTitle("Welcome");  
 primaryStage.setScene(scene);  
 primaryStage.show();  
  
 sceneManager = new SceneManager(primaryStage);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void setLoginData(String username, String password) {  
  
 this.username = username;  
 this.password = password;  
 try {  
 socket = new Socket("localhost", 12345);  
 in = new Scanner(socket.getInputStream());  
 out = new PrintWriter(socket.getOutputStream(), true);  
 objectOutputStream = new ObjectOutputStream(socket.getOutputStream());  
 objectInputStream =new ObjectInputStream(socket.getInputStream());  
 objectOutputStream.writeObject(username);  
 objectOutputStream.writeObject(password);  
  
  
 Employee employee = (Employee) objectInputStream.readObject();  
 if (employee != null) {  
 switch (employee.getRole()) {  
 case "Production Employee":  
 List<Task> employeeTasks = (List<Task>) objectInputStream.readObject();  
 employeeTasks.forEach(task -> System.*out*.println(task.getName()));  
 sceneManager.showEmployeeScene(this, employee,employeeTasks);  
 break;  
 case "Admin": {  
 List<Employee> employees = (List<Employee>) objectInputStream.readObject();  
 sceneManager.showAdminScene(this, employee, employees);  
 break;  
 }  
 case "Leader":  
 sceneManager.showLeaderScene(this, employee);  
 break;  
 case "Manager": {  
 List<Order> orders = (List<Order>) objectInputStream.readObject();  
 List<Task> tasks = (List<Task>) objectInputStream.readObject();  
 List<Employee> employees = (List<Employee>) objectInputStream.readObject();  
 List<Equipment> equipmentList = (List<Equipment>) objectInputStream.readObject();  
 sceneManager.showManagerScene(this, employee, orders, tasks,employees,equipmentList);  
 break;  
 }  
 }  
 } else {  
 sceneManager.showErrorScene(this);  
 }  
  
 } catch (IOException | ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void logOut() {  
  
 try {  
 objectOutputStream.writeObject("Close");  
 // Zamknij gniazdo i strumienie wejścia/wyjścia  
 if (socket != null && !socket.isClosed()) {  
 socket.close();  
 }  
 if (in != null) {  
 in.close();  
 }  
 if (out != null) {  
 out.close();  
 }  
 if (objectInputStream != null) {  
 objectInputStream.close();  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 sceneManager.showLoginScene(this);  
 }  
  
 public static void main(String[] args) {  
 *launch*(args);  
 }  
  
 public List<String> getRoles() {  
 try {  
 return (List<String>) objectInputStream.readObject();  
 }catch (IOException | ClassNotFoundException e){  
 e.printStackTrace();  
 }  
 return null;  
 }  
 public void updateEmployee(Employee employee) {  
  
 try {  
 objectOutputStream.writeObject("Update");  
 objectOutputStream.writeObject(employee);  
 }catch (IOException e){  
 e.printStackTrace();  
 }  
 }  
  
 public void showLogin() {  
 sceneManager.showLoginScene(this);  
 }  
 public List<String> getZones() {  
 try {  
 return (List<String>) objectInputStream.readObject();  
 }catch (IOException | ClassNotFoundException e){  
 e.printStackTrace();  
 }  
 return null;  
 }  
  
 public void cancel() {  
 try {  
  
  
 objectOutputStream.writeObject("Cancel");  
 }catch (IOException e){  
 e.printStackTrace();  
 }  
 }  
  
 public List<Object> getOrderInfo(Order order) {  
 List<Object> objects=new ArrayList<>();  
 try {  
 objectOutputStream.writeObject("orders");  
 objectOutputStream.writeObject(order);  
 objects = (List<Object>) objectInputStream.readObject();  
  
 }catch (IOException | ClassNotFoundException e){  
 e.printStackTrace();  
 }  
  
 return objects ;  
 }  
  
 public void addTask(Task task) {  
 try {  
 objectOutputStream.writeObject("addTask");  
 objectOutputStream.writeObject(task);  
 }catch (IOException e){  
 e.printStackTrace();  
 }  
 }  
  
 public List<List<String>> getUseEquipment(int id) {  
 try {  
 objectOutputStream.writeObject("getUseEquipment");  
 objectOutputStream.writeObject(id);  
 List<List<String>> useEquipment = (List<List<String>>) objectInputStream.readObject();  
 return useEquipment;  
 }catch (IOException | ClassNotFoundException e){  
 e.printStackTrace();  
 }  
 return null;  
 }  
  
 public void updateEquipment(Equipment equipment) {  
 try{  
 objectOutputStream.writeObject("updateEquipment");  
 objectOutputStream.writeObject(equipment);  
 }catch (IOException e){  
 e.printStackTrace();  
 }  
 }  
}

TCPClientFX to klasa, która pełni rolę klienta w architekturze klient-serwer w systemie opartym na technologii JavaFX. Aplikacja kliencka ta jest odpowiedzialna za nawiązywanie połączenia z serwerem, wymianę danych oraz zarządzanie interfejsem użytkownika.

Główne aspekty klasy:

1. Inicjalizacja Interfejsu Użytkownika:

- Klasa rozszerza klasę ***Application*** z JavaFX, co oznacza, że jej startową metodą jest ***start(Stage primaryStage).***

- W metodzie ***start***, ładowane są widoki za pomocą FXML i ustawiane kontrolery.

- Tworzony jest obiekt ***SceneManager*** do zarządzania widokami.

2. Nawiązywanie Połączenia:

- Po wprowadzeniu danych logowania, klient nawiązuje połączenie z serwerem, tworząc gniazdo i strumienie wejścia/wyjścia.

- Dane logowania są przesyłane na serwer za pomocą obiektów ***ObjectOutputStream*** i oczekuje na odpowiedź.

3. Logowanie i Obsługa Ról:

- W zależności od roli użytkownika, klient po odebraniu danych od serwera przechodzi do odpowiedniego widoku, który jest zarządzany przez ***SceneManager.***

4. Zamykanie Połączenia:

- Przy wylogowaniu następuje zamykanie gniazda i strumieni.

5. Komunikacja z Serwerem:

- Klient posiada metody do odbierania różnych rodzajów danych od serwera, takich jak role użytkowników, informacje o zamówieniach, listy zadań itp.

- Posiada także metody do aktualizacji danych pracownika oraz dodawania nowych zadań i aktualizacji sprzętu.

W sumie, TCPClientFX jest kluczowym elementem interfejsu użytkownika w systemie zarządzania zasobami, umożliwiającym użytkownikowi komunikację z serwerem i interakcję z systemem w zależności od przypisanej mu roli.

Teraz klasy kontrolerów widoków.

Klasa AdminController

package client.controller;  
  
import client.TCPClientFX;  
import javafx.beans.property.SimpleIntegerProperty;  
import javafx.beans.property.SimpleStringProperty;  
import javafx.collections.FXCollections;  
import javafx.fxml.FXML;  
import javafx.scene.Scene;  
import javafx.scene.control.\*;  
import javafx.scene.layout.HBox;  
import javafx.scene.layout.VBox;  
import javafx.stage.Stage;  
import server.\*;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class AdminController {  
 private TCPClientFX tcpClientFX;  
 private Employee employee;  
 private List<Employee> employees;  
 private List<Task> tasks;  
 private List<Order> orders;  
 private List<Equipment> equipment;  
 private List<Component> components;  
 List<String> roles = new ArrayList<>();  
 List<String> zones = new ArrayList<>();  
  
 @FXML  
 private Label name;  
 @FXML  
 private Label lastName;  
 @FXML  
 private Label role;  
 @FXML  
 private Label zone;  
 @FXML  
 private TabPane tabPane;  
  
 public AdminController(TCPClientFX tcpClientFX, Employee employee, List<Employee> employees, List<Task> tasks, List<Order> orders, List<Equipment> equipment, List<Component> components) {  
 this.tcpClientFX = tcpClientFX;  
 this.employee = employee;  
 this.employees = employees;  
 this.tasks = tasks;  
 this.orders = orders;  
 this.equipment = equipment;  
 this.components = components;  
 }  
  
 public AdminController(TCPClientFX tcpClientFX, Employee employee, List<Employee> employees) {  
 this.tcpClientFX = tcpClientFX;  
 this.employee = employee;  
 this.employees = employees;  
  
 }  
  
 @FXML  
 private void initialize() {  
 name.setText("Name: " + employee.getName());  
 lastName.setText("Last name: " + employee.getLastName());  
 role.setText("Role: " + employee.getRole());  
 zone.setText("Zone: " + employee.getZone());  
 roles = tcpClientFX.getRoles();  
 zones = tcpClientFX.getZones();  
 // Dodajemy zakładki i ich zawartość  
 addEmployeesTab();  
   
 }  
  
 private void addEmployeesTab() {  
 Tab employeesTab = new Tab("Employees");  
  
 // Tworzymy TableView dla pracowników  
 TableView<Employee> employeesTable = new TableView<>();  
 employeesTab.setContent(employeesTable);  
  
 // Tworzymy kolumny tabeli  
 TableColumn<Employee, Integer> idColumn = new TableColumn<>("ID");  
 TableColumn<Employee, String> nameColumn = new TableColumn<>("Name");  
 TableColumn<Employee, String> lastNameColumn = new TableColumn<>("Last Name");  
 TableColumn<Employee, String> roleColumn = new TableColumn<>("Role");  
 TableColumn<Employee, String> zoneColumn = new TableColumn<>("Zone");  
 TableColumn<Employee, Void> editColumn = new TableColumn<>("Edit");  
  
 // Ustawiamy, jakie wartości mają być wyświetlane w kolumnach  
 idColumn.setCellValueFactory(cellData -> new SimpleIntegerProperty(cellData.getValue().getId()).asObject());  
 nameColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getName()));  
 lastNameColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getLastName()));  
 roleColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getRole()));  
 zoneColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getZone()));  
  
 // Ustawiamy przyciski w kolumnie "Edit"  
 editColumn.setCellFactory(col -> {  
 TableCell<Employee, Void> cell = new TableCell<>() {  
 private final Button editButton = new Button("Edit");  
  
 {  
 editButton.setOnAction(event -> {  
 Employee employee = getTableView().getItems().get(getIndex());  
 int employeeId = employee.getId();  
  
 // Tworzymy nowe okno  
 Stage editStage = new Stage();  
 editStage.setTitle("Edit Employee");  
  
 // Tworzymy VBox, aby umieścić ComboBoxy, etykiety z danymi pracownika oraz przyciski  
 VBox vbox = new VBox();  
 vbox.setSpacing(10);  
  
 // ComboBox dla roli  
 ComboBox<String> roleComboBox = new ComboBox<>(FXCollections.*observableArrayList*(roles));  
 roleComboBox.setValue(employee.getRole()); // Ustawiamy początkową wartość na rolę pracownika  
  
 // ComboBox dla strefy  
 ComboBox<String> zoneComboBox = new ComboBox<>(FXCollections.*observableArrayList*(zones));  
 zoneComboBox.setValue(employee.getZone()); // Ustawiamy początkową wartość na strefę pracownika  
  
 // Etykiety z danymi pracownika  
 vbox.getChildren().add(new Label("Name: " + employee.getName()));  
 vbox.getChildren().add(new Label("Last name: " + employee.getLastName()));  
  
 // Etykieta i ComboBox dla roli  
 HBox roleBox = new HBox(new Label("Role: "), roleComboBox);  
 vbox.getChildren().add(roleBox);  
  
 // Etykieta i ComboBox dla strefy  
 HBox zoneBox = new HBox(new Label("Zone: "), zoneComboBox);  
 vbox.getChildren().add(zoneBox);  
  
 // Przycisk "Apply"  
 Button applyButton = new Button("Apply");  
 applyButton.setOnAction(applyEvent -> {  
 // Pobierz wybrane wartości z ComboBoxów  
 String newRole = roleComboBox.getValue();  
 String newZone = zoneComboBox.getValue();  
  
 // Zaktualizuj dane pracownika w TableView  
 employee.setRole(newRole);  
 employee.setZone(newZone);  
 tcpClientFX.updateEmployee(employee);  
 TableView<Employee> tableView = getTableView();  
 tableView.refresh();  
  
 // Zamknij okno po zastosowaniu zmian  
 editStage.close();  
 });  
  
 // Przycisk "Cancel"  
 Button cancelButton = new Button("Cancel");  
 cancelButton.setOnAction(cancelEvent -> {  
 tcpClientFX.cancel();  
 editStage.close(); // Zamknij okno bez zapisywania zmian  
 });  
  
 // Dodajemy przyciski do VBox  
 vbox.getChildren().addAll(applyButton, cancelButton);  
  
 // Dodajemy VBox do sceny  
 Scene editScene = new Scene(vbox, 300, 200);  
  
 // Ustawiamy scenę w nowym oknie  
 editStage.setScene(editScene);  
  
 // Pokazujemy nowe okno  
 editStage.show();  
  
 roles.forEach(System.*out*::println);  
 zones.forEach(System.*out*::println);  
 System.*out*.println("Edit button clicked for employee with ID: " + employeeId);  
 });  
  
  
 }  
  
 @Override  
 protected void updateItem(Void item, boolean empty) {  
 super.updateItem(item, empty);  
 setGraphic(empty ? null : editButton);  
 }  
 };  
 return cell;  
 });  
  
 // Dodajemy kolumny do tabeli  
 employeesTable.getColumns().addAll(idColumn, nameColumn, lastNameColumn, roleColumn, zoneColumn, editColumn);  
  
 // Dodajemy dane do tabeli  
 employeesTable.getItems().addAll(employees);  
  
 // Dodajemy zakładkę do TabPane  
 tabPane.getTabs().add(employeesTab);  
 }  
  
  
 @FXML  
 private void adminButtonAction() {  
 tcpClientFX.logOut();  
 }  
}

AdminController to kontroler obsługujący interfejs graficzny dla administratora w systemie zarządzania zasobami. Kontroler ten działa w środowisku JavaFX i współpracuje z obiektem TCPClientFX do komunikacji z serwerem.

Główne elementy kontrolera:

1. Inicjalizacja Interfejsu:

- Dane o zalogowanym administratorze oraz listy pracowników, zadań, zamówień, sprzętu i komponentów są przekazywane do kontrolera przy jego tworzeniu.

- Metoda ***initialize()*** ustawia etykiety informacyjne o administratorze i pobiera listy dostępnych ról i stref.

2. Zakładka "Employees":

- Metoda ***addEmployeesTab()*** tworzy zakładkę w interfejsie dla listy pracowników.

- Używając kontrolek tabeli, kolumn i przycisków, prezentuje informacje o pracownikach oraz umożliwia edycję ich roli i strefy.

- Przycisk "Edit" otwiera nowe okno z możliwością modyfikacji danych pracownika.

3. Edycja Danych Pracownika:

- W oknie edycji, administrator może wybrać nowe role i strefy z dostępnych list roli i stref.

- Zmiany są zatwierdzane przyciskiem "Apply" i przekazywane do serwera za pomocą ***TCPClientFX.***

4. Zamykanie Sesji:

- Przycisk "Logout" wywołuje metodę ***logOut()*** z obiektu ***TCPClientFX,*** co prowadzi do wylogowania administratora.

W rezultacie, ***AdminController*** zapewnia intuicyjny interfejs graficzny dla administratora, umożliwiając mu zarządzanie danymi pracowników i korzystanie z funkcji systemu zarządzania zasobami.

Klasa EmployeeController

package client.controller;  
  
import client.TCPClientFX;  
import javafx.fxml.FXML;  
import javafx.scene.control.Label;  
import javafx.scene.control.Tab;  
import javafx.scene.control.TabPane;  
import javafx.scene.control.TableColumn;  
import javafx.scene.control.TableView;  
import javafx.scene.control.cell.PropertyValueFactory;  
import server.Employee;  
import server.Task;  
  
import java.util.List;  
  
public class EmployeeController {  
  
 private TCPClientFX tcpClientFX;  
 private Employee employee;  
 private List<Task> tasks;  
  
 @FXML  
 private Label name;  
 @FXML  
 private Label lastName;  
 @FXML  
 private Label role;  
 @FXML  
 private Label zone;  
  
 @FXML  
 private TabPane tabPane;  
  
 public EmployeeController(TCPClientFX tcpClientFX, Employee employee, List<Task> tasks) {  
 this.tcpClientFX = tcpClientFX;  
 this.employee = employee;  
 this.tasks = tasks;  
 }  
  
 @FXML  
 private void initialize() {  
 name.setText("Name: " + employee.getName());  
 lastName.setText("Last name: " + employee.getLastName());  
 role.setText("Role: " + employee.getRole());  
 zone.setText("Zone: " + employee.getZone());  
  
  
 }  
 @FXML  
 private void employeeButtonAction() {  
 tcpClientFX.logOut();  
 }  
}

EmployeeController jest kontrolerem interfejsu graficznego przeznaczonym dla pracownika. Skupia się na prezentacji informacji o pracowniku oraz jego przypisanych zadaniach. W rezultacie ***EmployeeController*** zapewnia intuicyjny interfejs graficzny dla pracownika, wyświetlając podstawowe informacje o nim i umożliwiając łatwe wylogowanie się z systemu. Obecnie interfejs ten jest przygotowany do ewentualnego rozszerzenia o dodatkowe zakładki lub funkcje specyficzne dla roli pracownika.

ErrorController

package client.controller;  
import client.SceneManager;  
import client.TCPClientFX;  
import javafx.fxml.FXML;  
  
public class ErrorController {  
 private TCPClientFX tcpClientFX;  
  
 public ErrorController(TCPClientFX tcpClientFX) {  
 this.tcpClientFX = tcpClientFX;  
 }  
@FXML  
 private void errorButtonAction(){  
 tcpClientFX.showLogin();  
 }  
}

ErrorController obsługuje błąd logowania w interfejsie graficznym klienta, umożliwiając powrót do ekranu logowania po wystąpieniu problemu.

LeaderController

package client.controller;  
  
import client.TCPClientFX;  
import javafx.fxml.FXML;  
import javafx.scene.control.Label;  
import server.Employee;  
  
  
public class LeaderController {  
private TCPClientFX tcpClientFX;  
private Employee employee;  
 @FXML  
 private Label name;  
 @FXML  
 private Label lastName;  
 @FXML  
 private Label role;  
 @FXML  
 private Label zone;  
  
 public LeaderController(TCPClientFX tcpClientFX, Employee employee) {  
 this.tcpClientFX = tcpClientFX;  
 this.employee = employee;  
 }  
  
 @FXML  
 private void initialize() {  
 name.setText("Name: "+employee.getName());  
 lastName.setText("Last name: "+employee.getLastName());  
 role.setText("Role: "+employee.getRole());  
 zone.setText("Zone: "+employee.getZone());  
 }  
  
 @FXML  
 private void leaderButtonAction() {  
 tcpClientFX.logOut();  
 }  
}

LeaderController jest kontrolerem dla interfejsu lidera w systemie. Po pierwsze, inicjalizuje dane lidera, takie jak imię, nazwisko, rola i strefa, aby zostały wyświetlone na ekranie. Po drugie, obsługuje akcję przycisku, umożliwiając liderowi wylogowanie się z systemu. Kontroler ten jest przygotowany do dalszego rozwoju, w czasie aktualizacji aplikacji w kolejnych jej wersjach aby uzyskać w pełni funkcjonalny system.

Login Controller

package client.controller;  
  
import client.PasswordEncryptor;  
import client.TCPClientFX;  
import javafx.fxml.FXML;  
import javafx.scene.control.Button;  
import javafx.scene.control.PasswordField;  
import javafx.scene.control.TextField;  
  
public class LoginController {  
 private String username;  
 private String password;  
 @FXML  
 private TextField nameField;  
 @FXML  
 private PasswordField passField;  
 private Button loginButton;  
 private TCPClientFX tcpClientFX;  
  
 public LoginController(TCPClientFX tcpClientFX){  
 this.tcpClientFX = tcpClientFX;  
 }  
  
 @FXML  
 private void loginButtonAction(){  
 this.username = nameField.getText();  
 this.password = PasswordEncryptor.*encryptPassword*(passField.getText());  
 tcpClientFX.setLoginData(username,password);  
 }  
}

LoginController jest kontrolerem dla interfejsu logowania w systemie. Jego główne zadania to:

1. Obsługa Przycisku Logowania: Kontroler zawiera metodę ***loginButtonAction(),*** która jest wywoływana po naciśnięciu przycisku logowania. W tej metodzie pobierane są dane wprowadzone przez użytkownika (nazwa użytkownika i hasło), a następnie hasło jest szyfrowane przy użyciu ***PasswordEncryptor.*** Następnie te zaszyfrowane dane są przesyłane do klienta TCP aby przeprowadzić proces uwierzytelniania.

Klasa ManagerController

package client.controller;  
  
import client.TCPClientFX;  
import javafx.beans.property.SimpleIntegerProperty;  
import javafx.beans.property.SimpleStringProperty;  
import javafx.fxml.FXML;  
import javafx.geometry.Pos;  
import javafx.scene.Scene;  
import javafx.scene.control.\*;  
import javafx.scene.layout.GridPane;  
import javafx.scene.layout.HBox;  
import javafx.scene.layout.VBox;  
import javafx.stage.Stage;  
import server.\*;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Optional;  
import java.util.StringJoiner;  
  
public class ManagerController {  
 private TCPClientFX tcpClientFX;  
 private Employee employee;  
 private List<Order> orders;  
 private List<Task> tasks;  
 private List<Employee> employees;  
private List<Equipment> equipmentList;  
 @FXML  
 private Label name;  
 @FXML  
 private Label lastName;  
 @FXML  
 private Label role;  
 @FXML  
 private Label zone;  
  
 @FXML  
 private TabPane tabPane;  
  
 public ManagerController(TCPClientFX tcpClientFX, Employee employee, List<Order> orders, List<Task> tasks, List<Employee> employees, List<Equipment> equipmentList) {  
 this.tcpClientFX = tcpClientFX;  
 this.employee = employee;  
 this.orders = orders;  
 this.tasks = tasks;  
 this.employees = employees;  
 this.equipmentList=equipmentList;  
 }  
  
 @FXML  
 private void initialize() {  
 name.setText("Name: " + employee.getName());  
 lastName.setText("Last name: " + employee.getLastName());  
 role.setText("Role: " + employee.getRole());  
 zone.setText("Zone: " + employee.getZone());  
  
 // Dodajemy zakładki i ich zawartość  
 addOrdersTab();  
 addTasksTab(tasks);  
 addEmployeesTab(employees);  
 addEquipmentsTab(equipmentList);  
  
 }  
 private void addOrdersTab() {  
 Tab ordersTab = new Tab("Orders");  
  
 // Tworzymy TableView dla zamówień  
 TableView<Order> ordersTable = new TableView<>();  
 ordersTab.setContent(ordersTable);  
  
 // Tworzymy kolumny tabeli  
 TableColumn<Order, Integer> idColumn = new TableColumn<>("ID");  
 TableColumn<Order, String> productColumn = new TableColumn<>("Product");  
 TableColumn<Order, Integer> quantityOrderedColumn = new TableColumn<>("Quantity Ordered");  
 TableColumn<Order, Integer> quantityInProductionColumn = new TableColumn<>("Quantity In Production");  
 TableColumn<Order, Integer> quantityFinishedColumn = new TableColumn<>("Quantity Finished");  
 TableColumn<Order, String> statusColumn = new TableColumn<>("Status");  
 TableColumn<Order, Void> startColumn = new TableColumn<>("Start");  
  
 // Ustawiamy, jakie wartości mają być wyświetlane w kolumnach  
 idColumn.setCellValueFactory(cellData -> new SimpleIntegerProperty(cellData.getValue().getId()).asObject());  
 productColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getProduct().getName()));  
 quantityOrderedColumn.setCellValueFactory(cellData -> new SimpleIntegerProperty(cellData.getValue().getProduct().getQuantityOrdered()).asObject());  
 quantityInProductionColumn.setCellValueFactory(cellData -> new SimpleIntegerProperty(cellData.getValue().getProduct().getQuantityInProduction()).asObject());  
 quantityFinishedColumn.setCellValueFactory(cellData -> new SimpleIntegerProperty(cellData.getValue().getProduct().getQuantityFinished()).asObject());  
 statusColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getStatus()));  
  
 // Ustawiamy przyciski w kolumnie "Start"  
 startColumn.setCellFactory(col -> {  
 TableCell<Order, Void> cell = new TableCell<>() {  
 private final Button startButton = new Button("Start");  
  
 {  
 startButton.setOnAction(event -> {  
 Order order = getTableView().getItems().get(getIndex());  
 int orderId = order.getId();  
 // Tutaj dodaj logikę do obsługi przycisku Start dla danego zamówienia (orderId)  
 List<Object> objects = tcpClientFX.getOrderInfo(order);  
 List<Equipment> equipment = (List<Equipment>) objects.get(0);  
 List<Component> components = (List<Component>) objects.get(1);  
 // Otwórz nowe okno "New Task"  
 openNewTaskWindow(order,equipment,components, ordersTable);  
  
 });  
 }  
  
 @Override  
 protected void updateItem(Void item, boolean empty) {  
 super.updateItem(item, empty);  
  
 if (empty || getTableRow().getItem() == null) {  
 setGraphic(null);  
 } else {  
 Order order = (Order) getTableRow().getItem();  
 String status = order.getStatus();  
  
 // Wyświetl przycisk tylko dla zamówień z statusami "accepted" lub "inprogress"  
 if ("accepted".equalsIgnoreCase(status) || "progress".equalsIgnoreCase(status)) {  
 setGraphic(startButton);  
 } else {  
 setGraphic(null);  
 }  
 }  
 }  
 };  
 return cell;  
 });  
  
 // Dodajemy kolumny do tabeli  
 ordersTable.getColumns().addAll(idColumn, productColumn, quantityOrderedColumn, quantityInProductionColumn, quantityFinishedColumn, statusColumn, startColumn);  
  
 // Dodajemy wszystkie zamówienia do tabeli  
 ordersTable.getItems().addAll(orders);  
  
 // Dodajemy zakładkę do TabPane  
 tabPane.getTabs().add(ordersTab);  
 }  
  
 private void openNewTaskWindow(Order selectedOrder, List<Equipment> equipmentList, List<Component> components, TableView<Order> ordersTable) {  
 Stage newTaskStage = new Stage();  
 newTaskStage.setTitle("New Task");  
  
 // Tworzymy elementy interfejsu użytkownika  
 Label nameLabel = new Label("Name:");  
 TextField nameTextField = new TextField();  
  
 Label priorityLabel = new Label("Priority:");  
 ComboBox<String> priorityComboBox = new ComboBox<>();  
 priorityComboBox.getItems().addAll("High", "Normal", "Low");  
  
 Label descriptionLabel = new Label("Description:");  
 TextArea descriptionTextArea = new TextArea();  
  
 Label normLabel = new Label("Norm:");  
 TextField normTextField = new TextField();  
  
 Label componentLabel = new Label("Components:");  
 VBox componentsVBox = new VBox();  
  
 List<CheckBox> componentCheckboxes = new ArrayList<>();  
 components.forEach(component -> componentCheckboxes.add(new CheckBox(component.getName())));  
 componentsVBox.getChildren().addAll(componentCheckboxes);  
  
 Label equipmentLabel = new Label("Equipment:");  
 ComboBox<String> equipmentComboBox = new ComboBox<>();  
 equipmentList.forEach(equipment -> equipmentComboBox.getItems().add(equipment.getName()));  
  
  
 // Dodajemy pole Spinner dla ilości (Quantity)  
 Label quantityLabel = new Label("Quantity:");  
 Spinner<Integer> quantitySpinner = new Spinner<>(0, calculateMaxQuantity(selectedOrder), 0, 1);  
  
 // Tworzymy przyciski "Apply" i "Cancel"  
 Button applyButton = new Button("Apply");  
 Button cancelButton = new Button("Cancel");  
  
 // Ustawiamy akcję dla przycisku "Apply"  
 applyButton.setOnAction(event -> {  
 // Tutaj dodaj logikę do zastosowania wprowadzonych danych  
 List<Component> selectedComponents = new ArrayList<>();  
  
 for (CheckBox checkBox : componentCheckboxes) {  
 if (checkBox.isSelected()) {  
 String componentName = checkBox.getText();  
  
 // Szukaj obiektu Component po nazwie w liście components  
 Optional<Component> foundComponent = components.stream()  
 .filter(component -> component.getName().equals(componentName))  
 .findFirst();  
  
 foundComponent.ifPresent(selectedComponents::add);  
 }  
 }  
  
 // Pobierz nazwę wybranego sprzętu  
 String selectedEquipmentName = equipmentComboBox.getValue();  
  
 // Szukaj obiektu Equipment o danej nazwie w equipmentList  
 Equipment selectedEquipment = equipmentList.stream()  
 .filter(equipment -> equipment.getName().equals(selectedEquipmentName))  
 .findFirst()  
 .orElse(null);  
  
 assert selectedEquipment != null;  
 Task task = new Task(1, nameTextField.getText(), priorityComboBox.getValue(), descriptionTextArea.getText(), Integer.*parseInt*(normTextField.getText()), selectedComponents, selectedEquipment, selectedEquipment.getZone(), quantitySpinner.getValue(),selectedOrder.getProduct().getId(), selectedOrder.getId());  
 tcpClientFX.addTask(task);  
 selectedOrder.getProduct().setQuantityInProduction(selectedOrder.getProduct().getQuantityInProduction()+quantitySpinner.getValue());  
 selectedOrder.setStatus("progress");  
 // Aktualizuj dane w tabeli  
 ordersTable.getItems().clear(); // Wyczyść aktualne dane  
 ordersTable.getItems().addAll(orders); // Dodaj nowe dane  
  
 newTaskStage.close();  
 });  
  
  
 // Ustawiamy akcję dla przycisku "Cancel"  
 cancelButton.setOnAction(event -> {  
 // Tutaj dodaj logikę do anulowania wprowadzonych zmian  
 newTaskStage.close();  
 });  
  
 // Ustawiamy layout za pomocą GridPane  
 GridPane gridPane = new GridPane();  
 gridPane.setVgap(10);  
 gridPane.setHgap(10);  
 gridPane.setAlignment(Pos.*CENTER*);  
  
 // Dodajemy etykiety i pola do wprowadzania danych do GridPane  
 gridPane.add(nameLabel, 0, 0);  
 gridPane.add(nameTextField, 1, 0);  
  
 gridPane.add(priorityLabel, 0, 1);  
 gridPane.add(priorityComboBox, 1, 1);  
  
 gridPane.add(descriptionLabel, 0, 2);  
 gridPane.add(descriptionTextArea, 1, 2);  
  
 gridPane.add(normLabel, 0, 3);  
 gridPane.add(normTextField, 1, 3);  
  
 gridPane.add(componentLabel, 0, 4);  
 gridPane.add(componentsVBox, 1, 4);  
  
 gridPane.add(equipmentLabel, 0, 5);  
 gridPane.add(equipmentComboBox, 1, 5);  
  
  
 gridPane.add(quantityLabel, 0, 7);  
 gridPane.add(quantitySpinner, 1, 7);  
  
 gridPane.add(new HBox(10, applyButton, cancelButton), 1, 8);  
  
 // Ustawiamy scenę  
 Scene scene = new Scene(gridPane, 400, 600); // Zwiększam wysokość okna  
 newTaskStage.setScene(scene);  
  
 // Pokazujemy nowe okno  
 newTaskStage.show();  
 }  
  
 private void addTasksTab(List<Task> tasks) {  
 Tab tasksTab = new Tab("Tasks");  
  
 // Tworzymy TableView dla zadań  
 TableView<Task> tasksTable = new TableView<>();  
 tasksTab.setContent(tasksTable);  
  
 // Tworzymy kolumny tabeli  
 TableColumn<Task, Integer> idColumn = new TableColumn<>("ID");  
 TableColumn<Task, String> nameColumn = new TableColumn<>("Name");  
 TableColumn<Task, String> priorityColumn = new TableColumn<>("Priority");  
 TableColumn<Task, String> descriptionColumn = new TableColumn<>("Description");  
 TableColumn<Task, Integer> normColumn = new TableColumn<>("Norm");  
 TableColumn<Task, String> zoneColumn = new TableColumn<>("Zone");  
 TableColumn<Task, Integer> quantityColumn = new TableColumn<>("Quantity");  
 TableColumn<Task, Void> viewColumn = new TableColumn<>("View");  
  
 // Ustawiamy, jakie wartości mają być wyświetlane w kolumnach  
 idColumn.setCellValueFactory(cellData -> new SimpleIntegerProperty(cellData.getValue().getId()).asObject());  
 nameColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getName()));  
 priorityColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getPriority()));  
 descriptionColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getDescription()));  
 normColumn.setCellValueFactory(cellData -> new SimpleIntegerProperty(cellData.getValue().getNorm()).asObject());  
 zoneColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getZone()));  
 quantityColumn.setCellValueFactory(cellData -> new SimpleIntegerProperty(cellData.getValue().getQuantity()).asObject());  
  
 // Ustawiamy przyciski w kolumnie "View"  
 viewColumn.setCellFactory(col -> {  
 TableCell<Task, Void> cell = new TableCell<>() {  
 private final Button viewButton = new Button("View");  
  
 {  
 viewButton.setOnAction(event -> {  
 Task task = getTableView().getItems().get(getIndex());  
 int taskId = task.getId();  
 // Tutaj dodaj logikę do obsługi przycisku View dla danego zadania (taskId)  
 System.*out*.println("View button clicked for task with ID: " + taskId);  
  
 // Otwórz nowe okno "View Task"  
 //openViewTaskWindow(task);  
 });  
 }  
  
 @Override  
 protected void updateItem(Void item, boolean empty) {  
 super.updateItem(item, empty);  
  
 if (empty || getTableRow().getItem() == null) {  
 setGraphic(null);  
 } else {  
 setGraphic(viewButton);  
 }  
 }  
 };  
 return cell;  
 });  
  
 // Dodajemy kolumny do tabeli  
 tasksTable.getColumns().addAll(idColumn, nameColumn, priorityColumn, descriptionColumn, normColumn, zoneColumn, quantityColumn, viewColumn);  
  
 // Dodajemy wszystkie zadania do tabeli  
 tasksTable.getItems().addAll(tasks);  
  
 // Dodajemy zakładkę do TabPane  
 tabPane.getTabs().add(tasksTab);  
 }  
  
  
  
 private int calculateMaxQuantity(Order selectedOrder) {  
 if (selectedOrder != null) {  
 int maxQuantity = selectedOrder.getProduct().getQuantityOrdered()  
 - selectedOrder.getProduct().getQuantityInProduction()  
 - selectedOrder.getProduct().getQuantityFinished();  
  
 return Math.*max*(maxQuantity, 0); // Nie pozwalamy na ujemne wartości  
 }  
 return 0;  
 }  
  
 private void addEmployeesTab(List<Employee> employees) {  
 Tab employeesTab = new Tab("Employees");  
  
 // Tworzymy TableView dla pracowników  
 TableView<Employee> employeesTable = new TableView<>();  
 employeesTab.setContent(employeesTable);  
  
 // Tworzymy kolumny tabeli  
 TableColumn<Employee, Integer> idColumn = new TableColumn<>("ID");  
 TableColumn<Employee, String> nameColumn = new TableColumn<>("Name");  
 TableColumn<Employee, String> lastNameColumn = new TableColumn<>("Last Name");  
 TableColumn<Employee, String> roleColumn = new TableColumn<>("Role");  
 TableColumn<Employee, String> zoneColumn = new TableColumn<>("Zone");  
 TableColumn<Employee, Void> viewColumn = new TableColumn<>("View");  
  
 // Ustawiamy, jakie wartości mają być wyświetlane w kolumnach  
 idColumn.setCellValueFactory(cellData -> new SimpleIntegerProperty(cellData.getValue().getId()).asObject());  
 nameColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getName()));  
 lastNameColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getLastName()));  
 roleColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getRole()));  
 zoneColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getZone()));  
  
 // Ustawiamy przyciski w kolumnie "View"  
 viewColumn.setCellFactory(col -> {  
 TableCell<Employee, Void> cell = new TableCell<>() {  
 private final Button viewButton = new Button("View");  
  
 {  
 viewButton.setOnAction(event -> {  
 Employee employee = getTableView().getItems().get(getIndex());  
 int employeeId = employee.getId();  
 // Tutaj dodaj logikę do obsługi przycisku View dla danego pracownika (employeeId)  
 System.*out*.println("View button clicked for employee with ID: " + employeeId);  
  
 // Otwórz nowe okno "View Employee"  
 // openViewEmployeeWindow(employee);  
 });  
 }  
  
 @Override  
 protected void updateItem(Void item, boolean empty) {  
 super.updateItem(item, empty);  
  
 if (empty || getTableRow().getItem() == null) {  
 setGraphic(null);  
 } else {  
 setGraphic(viewButton);  
 }  
 }  
 };  
 return cell;  
 });  
  
 // Dodajemy kolumny do tabeli  
 employeesTable.getColumns().addAll(idColumn, nameColumn, lastNameColumn, roleColumn, zoneColumn, viewColumn);  
  
 // Dodajemy wszystkich pracowników do tabeli  
 employeesTable.getItems().addAll(employees);  
  
 // Dodajemy zakładkę do TabPane  
 tabPane.getTabs().add(employeesTab);  
 }  
 private void addEquipmentsTab(List<Equipment> equipments) {  
 Tab equipmentsTab = new Tab("Equipments");  
  
 // Tworzymy TableView dla sprzętu  
 TableView<Equipment> equipmentsTable = new TableView<>();  
 equipmentsTab.setContent(equipmentsTable);  
  
 // Tworzymy kolumny tabeli  
 TableColumn<Equipment, Integer> idColumn = new TableColumn<>("ID");  
 TableColumn<Equipment, String> nameColumn = new TableColumn<>("Name");  
 TableColumn<Equipment, String> statusColumn = new TableColumn<>("Status");  
 TableColumn<Equipment, String> zoneColumn = new TableColumn<>("Zone");  
 TableColumn<Equipment, Void> viewColumn = new TableColumn<>("View");  
  
 // Ustawiamy, jakie wartości mają być wyświetlane w kolumnach  
 idColumn.setCellValueFactory(cellData -> new SimpleIntegerProperty(cellData.getValue().getId()).asObject());  
 nameColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getName()));  
 statusColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getStatus()));  
 zoneColumn.setCellValueFactory(cellData -> new SimpleStringProperty(cellData.getValue().getZone()));  
  
 // Ustawiamy przyciski w kolumnie "View"  
 viewColumn.setCellFactory(col -> {  
 TableCell<Equipment, Void> cell = new TableCell<>() {  
 private final Button viewButton = new Button("View");  
  
 {  
 viewButton.setOnAction(event -> {  
 Equipment equipment = getTableView().getItems().get(getIndex());  
 List<List<String>> equipmentUse = tcpClientFX.getUseEquipment(equipment.getId());  
 openViewEquipmentWindow(equipment,equipmentUse);  
 });  
 }  
  
 @Override  
 protected void updateItem(Void item, boolean empty) {  
 super.updateItem(item, empty);  
  
 if (empty || getTableRow().getItem() == null) {  
 setGraphic(null);  
 } else {  
 setGraphic(viewButton);  
 }  
 }  
 };  
 return cell;  
 });  
  
 // Dodajemy kolumny do tabeli  
 equipmentsTable.getColumns().addAll(idColumn, nameColumn, statusColumn, zoneColumn, viewColumn);  
  
 // Dodajemy wszystkie sprzęty do tabeli  
 equipmentsTable.getItems().addAll(equipments);  
  
 // Dodajemy zakładkę do TabPane  
 tabPane.getTabs().add(equipmentsTab);  
 }  
 private void openViewEquipmentWindow(Equipment equipment, List<List<String>> equipmentUse) {  
 // Tworzymy nowe okno "View Equipment"  
 Stage viewEquipmentStage = new Stage();  
 viewEquipmentStage.setTitle("View Equipment");  
  
 // Tworzymy kontener VBox dla układu okna  
 VBox vbox = new VBox(10);  
  
 // Dodajemy etykiety i pola tekstowe do VBox  
 Label nameLabel = new Label("Name: " + equipment.getName());  
  
 // Lista rozwijana (ComboBox) dla statusu  
 Label statusLable = new Label("Status: ");  
 ComboBox<String> statusComboBox = new ComboBox<>();  
 statusComboBox.getItems().addAll("available", "in use", "out of use");  
 statusComboBox.setValue(equipment.getStatus()); // Ustawienie domyślnego statusu  
  
 Label zoneLabel = new Label("Zone: " + equipment.getZone());  
  
 // Dodajemy elementy do VBox  
 vbox.getChildren().addAll(nameLabel, statusLable, statusComboBox, zoneLabel);  
  
 // Dodaj informacje z equipmentUse  
 for (List<String> innerList : equipmentUse) {  
 StringJoiner stringJoiner = new StringJoiner(" ");  
 for (String value : innerList) {  
 stringJoiner.add(value);  
  
 }  
 Label label = new Label(stringJoiner.toString());  
 vbox.getChildren().add(label);  
  
 }  
  
 // Tworzymy HBox dla przycisków Apply i Cancel  
 HBox buttonBox = new HBox(10);  
 Button applyButton = new Button("Apply");  
 Button cancelButton = new Button("Cancel");  
  
 // Obsługa przycisku Apply  
 applyButton.setOnAction(event -> {  
 equipment.setStatus(statusComboBox.getValue());  
 tcpClientFX.updateEquipment(equipment);  
  
 // Zamknij okno po zastosowaniu zmian  
 viewEquipmentStage.close();  
 });  
  
 // Obsługa przycisku Cancel  
 cancelButton.setOnAction(event -> {  
 // Zamknij okno bez zastosowywania zmian  
 viewEquipmentStage.close();  
 });  
  
 // Dodaj przyciski do HBox  
 buttonBox.getChildren().addAll(applyButton, cancelButton);  
  
 // Dodaj HBox do VBox  
 vbox.getChildren().add(buttonBox);  
  
 // Ustawiamy VBox jako scenę  
 Scene scene = new Scene(vbox, 400, 300); // Zwiększyłem szerokość okna, aby pomieścić więcej informacji  
 viewEquipmentStage.setScene(scene);  
  
 // Pokazujemy nowe okno  
 viewEquipmentStage.show();  
 }  
  
 @FXML  
 private void managerButtonAction() {  
 tcpClientFX.logOut();  
 }  
}

Kod w klasie ManagerController odpowiada za obsługę interfejsu graficznego dla managera w systemie. W kodzie tym można wyróżnić:

1. Inicjalizacja Interfejsu: Metoda ***initialize()*** jest wywoływana podczas inicjalizacji kontrolera. W tej metodzie ustawiane są początkowe dane dla etykiet, a także dodawane są zakładki dla zamówień, zadań, pracowników i sprzętów.

2. Zakładka Zamówień: Tworzenie zakładki "Orders" zawierającej tabelę z zamówieniami. Dla każdego zamówienia dodawane są kolumny, a także przycisk "Start" w kolumnie, który uruchamia nowe zadanie związanego z zamówieniem.

3. Zakładka Zadań: Tworzenie zakładki "Tasks" zawierającej tabelę z zadaniami. Dla każdego zadania dodawane są kolumny, a także przycisk "View" w kolumnie, który jest przygotowywany do wyświetlania informacji o danym tasku.

4. Zakładka Pracowników: Tworzenie zakładki "Employees" zawierającej tabelę z informacjami o pracownikach. Dla każdego pracownika dodawane są kolumny, a także przycisk "View" w kolumnie, który będzie odpowiadał za wyświetlanie danych o pracownikach oraz będzie pozwalał na zmianę pewnych danych.

5. Zakładka Sprzętu: Tworzenie zakładki "Equipments" zawierającej tabelę z informacjami o sprzęcie. Dla każdego sprzętu dodawane są kolumny, a także przycisk "View" w kolumnie, który otwiera szczegóły sprzętu, w tym informacje o jego użyciu.

6. Obsługa Nowego Zadania: Otwarcie nowego okna umożliwiającego managerowi przypisanie nowego zadania dla konkretnego zamówienia. Manager może określić priorytet, opis zadania, normę, komponenty, sprzęt i ilość do produkcji.

7. Obsługa Nowego Sprzętu: Otwarcie nowego okna umożliwiającego managerowi przeglądanie informacji o konkretnym sprzęcie. Możliwość zmiany statusu sprzętu i wyświetlenia informacji o jego użyciu.

8. Akcja Przycisku Managera: Obsługa przycisku "Manager", który umożliwia wylogowanie się z systemu.

W pakiecie tym znajduje się też plik z stylami css.

. root{  
 -fx-min-width: 800px;  
 -fx-min-height: 600px;  
 -fx-background-color: #fffaaa;  
}  
  
/\* Styl dla GridPane \*/  
#gridPane {  
 -fx-min-width: 800;  
 -fx-min-height: 600;  
 -fx-padding: 0 80 5 80;  
  
}  
  
  
/\* Styl dla Label i TextField \*/  
.label, .text-field {  
 -fx-padding: 5;  
}  
  
/\* Styl dla Button \*/  
#loginButton {  
 -fx-padding: 5 0 5 0;  
 -fx-background-color: #3f919a;  
 -fx-font-size: 12;  
 -fx-font-family: Arial;  
 -fx-background-radius: 12;  
  
  
  
}  
  
/\* Styl dla kolumny z logo \*/  
#logoColumn {  
 -fx-pref-width: 400; /\* Szerokość kolumny z logo \*/  
}  
  
/\* Styl dla kolumny z polami logowania \*/  
#loginColumn {  
 -fx-alignment: CENTER;  
  
 -fx-pref-width: 400; /\* Szerokość kolumny z polami logowania \*/  
}

Plik do odpowiedzialny za wygląd aplikacji. Jest to wstęp do stworzenia interfejsu graficznego według pierwotnych założeń. Plik ten może być dalej aktualizowany dostosowując grafikę aplikacji.

Teraz pakiet View i pierwsze widoki aplikacji:

Plik AdminScene.fxml

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.geometry.Insets?>  
<?import javafx.scene.control.Button?>  
<?import javafx.scene.control.Label?>  
<?import javafx.scene.control.Tab?>  
<?import javafx.scene.control.TabPane?>  
<?import javafx.scene.layout.ColumnConstraints?>  
<?import javafx.scene.layout.GridPane?>  
<?import javafx.scene.layout.RowConstraints?>  
  
<GridPane alignment="center" hgap="5" maxHeight="600.0" maxWidth="600.0" minHeight="400.0" minWidth="400.0" vgap="5" xmlns="http://javafx.com/javafx/21" xmlns:fx="http://javafx.com/fxml/1">  
 <padding>  
 <Insets bottom="10" left="10" right="10" top="10" />  
 </padding>  
  
 <Button onAction="#adminButtonAction" text="Wyloguj" GridPane.columnIndex="4" />  
  
 <!-- Dodane Label'e -->  
 <Label fx:id="name" />  
 <Label fx:id="lastName" GridPane.columnIndex="1" />  
 <Label fx:id="role" GridPane.columnIndex="2" />  
 <Label fx:id="zone" GridPane.columnIndex="3" />  
  
 <TabPane fx:id="tabPane" GridPane.columnIndex="0" GridPane.columnSpan="4" GridPane.rowIndex="1">  
 <tabs>  
  
 </tabs>  
 </TabPane>  
  
 <columnConstraints>  
 <ColumnConstraints maxWidth="108.5" minWidth="13.5" prefWidth="78.5" />  
 <ColumnConstraints maxWidth="314.0" minWidth="0.0" prefWidth="117.0" />  
 <ColumnConstraints maxWidth="342.0" minWidth="0.0" prefWidth="163.0" />  
 <ColumnConstraints hgrow="ALWAYS" maxWidth="415.0" minWidth="56.0" prefWidth="138.0" />  
 <ColumnConstraints hgrow="ALWAYS" maxWidth="366.0" minWidth="56.0" prefWidth="75.0" />  
 </columnConstraints>  
 <rowConstraints>  
 <RowConstraints maxHeight="191.0" minHeight="10.0" prefHeight="33.0" vgrow="ALWAYS" />  
 <RowConstraints maxHeight="327.0" minHeight="169.0" prefHeight="327.0" vgrow="ALWAYS" />  
 <RowConstraints />  
 <RowConstraints />  
 <RowConstraints />  
 </rowConstraints>  
  
</GridPane>

1. Dodanie Przycisku Wyloguj:

Dodano przycisk "Wyloguj" na pozycji (4, 0) w siatce GridPane, a jego akcja zostanie obsłużona w metodzie o nazwie adminButtonAction.

2. Etykiety Informacji o Pracowniku:

Dodane są etykiety Label reprezentujące informacje o pracowniku (Imię, Nazwisko, Rola, Strefa) umieszczone w kolumnach 0-3 w wierszu 0.

Pozostałe pliki FXML są podobne do tego pierwszego w nich również znajduje się siatka GridPane oraz Lable i Buttony są to podstawowe widoki aplikacji pozostałą część widoków realizują już kontrolery opisane wyżej.

Plik EmployeeScene.fxml

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.geometry.Insets?>  
<?import javafx.scene.control.Button?>  
<?import javafx.scene.control.Label?>  
<?import javafx.scene.layout.ColumnConstraints?>  
<?import javafx.scene.layout.GridPane?>  
<?import javafx.scene.layout.RowConstraints?>  
  
<GridPane hgap="5" maxHeight="600.0" maxWidth="600.0" minHeight="400.0" minWidth="400.0" vgap="5" xmlns="http://javafx.com/javafx/21" xmlns:fx="http://javafx.com/fxml/1">  
  
 <padding>  
 <Insets bottom="10" left="10" right="10" top="10" />  
 </padding>  
  
 <Button onAction="#employeeButtonAction" text="Wyloguj" GridPane.columnIndex="4" />  
  
 <!-- Dodane Label'e -->  
 <Label fx:id="name"/>  
 <Label fx:id="lastName" GridPane.columnIndex="1" />  
 <Label fx:id="role" GridPane.columnIndex="2" />  
 <Label fx:id="zone" GridPane.columnIndex="3" />  
  
 <columnConstraints>  
 <ColumnConstraints maxWidth="108.5" minWidth="13.5" prefWidth="78.5" />  
 <ColumnConstraints maxWidth="314.0" minWidth="0.0" prefWidth="117.0" />  
 <ColumnConstraints maxWidth="342.0" minWidth="0.0" prefWidth="163.0" />  
 <ColumnConstraints hgrow="ALWAYS" maxWidth="415.0" minWidth="56.0" prefWidth="138.0" />  
 <ColumnConstraints hgrow="ALWAYS" maxWidth="366.0" minWidth="56.0" prefWidth="75.0" />  
 </columnConstraints>  
 <rowConstraints>  
 <RowConstraints maxHeight="191.0" minHeight="10.0" prefHeight="33.0" vgrow="ALWAYS" />  
 <RowConstraints maxHeight="327.0" minHeight="169.0" prefHeight="327.0" vgrow="ALWAYS" />  
 <RowConstraints />  
 <RowConstraints />  
 <RowConstraints />  
 </rowConstraints>  
  
</GridPane>

Plik ErrorScene.fxml

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.scene.control.Button?>  
<?import javafx.scene.control.Label?>  
<?import javafx.scene.layout.AnchorPane?>  
<?import javafx.scene.text.Font?>  
  
<AnchorPane prefHeight="145.0" prefWidth="155.0" xmlns="http://javafx.com/javafx/21" xmlns:fx="http://javafx.com/fxml/1">  
 <children>  
 <Button layoutX="59.0" layoutY="118.0" mnemonicParsing="false" onAction="#errorButtonAction" text="Close" />  
 <Label layoutX="26.0" layoutY="73.0" prefHeight="44.0" prefWidth="141.0" text="Login error !!!" textFill="#e10000">  
 <font>  
 <Font size="24.0" />  
 </font>  
 </Label>  
 </children>  
</AnchorPane>

Plik LeaderScene.fxml

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.scene.control.\*?>  
<?import javafx.scene.layout.\*?>  
  
<?import javafx.geometry.Insets?>  
<GridPane xmlns="http://javafx.com/javafx"  
 xmlns:fx="http://javafx.com/fxml"  
 maxHeight="600.0" maxWidth="600.0" minHeight="400.0" minWidth="400.0"  
 alignment="center"  
 hgap="5"  
 vgap="5">  
 <padding>  
 <Insets bottom="10" left="10" right="10" top="10" />  
 </padding>  
  
 <Button onAction="#leaderButtonAction" text="Wyloguj" GridPane.columnIndex="4" />  
  
 <!-- Dodane Label'e -->  
 <Label fx:id="name"/>  
 <Label fx:id="lastName" GridPane.columnIndex="1" />  
 <Label fx:id="role" GridPane.columnIndex="2" />  
 <Label fx:id="zone" GridPane.columnIndex="3" />  
  
 <columnConstraints>  
 <ColumnConstraints maxWidth="108.5" minWidth="13.5" prefWidth="78.5" />  
 <ColumnConstraints maxWidth="314.0" minWidth="0.0" prefWidth="117.0" />  
 <ColumnConstraints maxWidth="342.0" minWidth="0.0" prefWidth="163.0" />  
 <ColumnConstraints hgrow="ALWAYS" maxWidth="415.0" minWidth="56.0" prefWidth="138.0" />  
 <ColumnConstraints hgrow="ALWAYS" maxWidth="366.0" minWidth="56.0" prefWidth="75.0" />  
 </columnConstraints>  
 <rowConstraints>  
 <RowConstraints maxHeight="191.0" minHeight="10.0" prefHeight="33.0" vgrow="ALWAYS" />  
 <RowConstraints maxHeight="327.0" minHeight="169.0" prefHeight="327.0" vgrow="ALWAYS" />  
 <RowConstraints />  
 <RowConstraints />  
 <RowConstraints />  
 </rowConstraints>  
</GridPane>

Plik LoginScene.fxml

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.geometry.Insets?>  
<?import javafx.scene.control.Button?>  
<?import javafx.scene.control.Label?>  
<?import javafx.scene.control.PasswordField?>  
<?import javafx.scene.control.TextField?>  
<?import javafx.scene.layout.ColumnConstraints?>  
<?import javafx.scene.layout.GridPane?>  
<?import javafx.scene.layout.RowConstraints?>  
  
<?import javafx.scene.image.ImageView?>  
<?import javafx.scene.layout.AnchorPane?>  
<?import javafx.scene.image.Image?>  
<AnchorPane style="-fx-background-color: #fffaaa" xmlns="http://javafx.com/javafx"  
 xmlns:fx="http://javafx.com/fxml"  
 >  
<GridPane id="gridPane" xmlns="http://javafx.com/javafx/21" xmlns:fx="http://javafx.com/fxml/1" >  
  
 <padding>  
 <Insets bottom="10" left="10" right="10" top="10" />  
 </padding>  
  
 <columnConstraints>  
 <ColumnConstraints fx:id="logoColumn"/>  
 <ColumnConstraints fx:id="loginColumn"/>  
 </columnConstraints>  
  
  
 <ImageView id="logoImage" GridPane.columnIndex="0" GridPane.rowIndex="0" GridPane.columnSpan="6" >  
 <image >  
 <Image url="@logo.png" />  
 </image>  
 </ImageView>  
  
 <Label id="loginLabel" text="Login:" GridPane.columnIndex="1" GridPane.rowIndex="1" minWidth="100" alignment="CENTER\_RIGHT"/>  
 <TextField fx:id="nameField" id="loginField" GridPane.columnIndex="2" GridPane.rowIndex="1" minWidth="100"/>  
  
 <Label id="passwordLabel" text="Password:" GridPane.columnIndex="3" GridPane.rowIndex="1" minWidth="100" alignment="CENTER\_RIGHT"/>  
 <PasswordField fx:id="passField" id="passwordField" GridPane.columnIndex="4" GridPane.rowIndex="1" minWidth="100"/>  
  
 <Button id="loginButton" onAction="#loginButtonAction" text="Login" GridPane.columnIndex="5" GridPane.rowIndex="1" minWidth="100" />  
  
</GridPane>  
</AnchorPane>

Plik ManagerScene.fxml

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.geometry.Insets?>  
<?import javafx.scene.control.Button?>  
<?import javafx.scene.control.Label?>  
<?import javafx.scene.control.Tab?>  
<?import javafx.scene.control.TabPane?>  
<?import javafx.scene.layout.ColumnConstraints?>  
<?import javafx.scene.layout.GridPane?>  
<?import javafx.scene.layout.RowConstraints?>  
  
<?import javafx.scene.layout.VBox?>  
<GridPane style="-fx-background-color: #fffaaa; -fx-min-width: 800;-fx-min-height: 600" alignment="center" hgap="5" maxHeight="600.0" maxWidth="600.0" minHeight="400.0" minWidth="400.0" vgap="5" xmlns="http://javafx.com/javafx/21" xmlns:fx="http://javafx.com/fxml/1">  
  
 <padding>  
 <Insets bottom="10" left="10" right="10" top="10" />  
 </padding>  
  
 <Button onAction="#managerButtonAction" text="Wyloguj" GridPane.columnIndex="4" />  
  
 <!-- Dodane Label'e -->  
 <Label fx:id="name" />  
 <Label fx:id="lastName" GridPane.columnIndex="1" />  
 <Label fx:id="role" GridPane.columnIndex="2" />  
 <Label fx:id="zone" GridPane.columnIndex="3" />  
  
 <TabPane fx:id="tabPane" GridPane.columnIndex="0" GridPane.columnSpan="5" GridPane.rowIndex="1">  
 <tabs>  
  
 </tabs>  
 </TabPane>  
  
 <columnConstraints>  
 <ColumnConstraints maxWidth="108.5" minWidth="13.5" prefWidth="78.5" />  
 <ColumnConstraints maxWidth="314.0" minWidth="0.0" prefWidth="117.0" />  
 <ColumnConstraints maxWidth="342.0" minWidth="0.0" prefWidth="163.0" />  
 <ColumnConstraints hgrow="ALWAYS" maxWidth="415.0" minWidth="56.0" prefWidth="138.0" />  
 <ColumnConstraints hgrow="ALWAYS" maxWidth="366.0" minWidth="56.0" prefWidth="75.0" />  
 </columnConstraints>  
 <rowConstraints>  
 <RowConstraints maxHeight="191.0" minHeight="10.0" prefHeight="33.0" vgrow="ALWAYS" />  
 <RowConstraints maxHeight="327.0" minHeight="169.0" prefHeight="327.0" vgrow="ALWAYS" />  
 <RowConstraints />  
 <RowConstraints />  
 <RowConstraints />  
 </rowConstraints>  
  
</GridPane>

Na koniec pozostaje pakiet server który tak naprawdę jest „mózgiem” całej aplikacji i to on odpowiada za dane wyświetlane klientowi, oraz za przetwarzanie danych od niego odebranych oraz za łączenie i wykonywanie zapytań na bazie danych.

Klasa Admin

package server;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class Admin extends Employee{  
 public Admin(int id, String name, String lastName, String role, String zone, String login, String password) {  
 super(id, name, lastName, role, zone, login, password);  
 }  
  
 public Admin(int id, String name, String lastName, String role, String zone, String login, String password, Task task) {  
 super(id, name, lastName, role, zone, login, password, task);  
 }  
 public void addRole(Employee employee, String role){  
  
 }  
 public void changeRole(Employee updateEmployee){  
 MySQLDatabaseConnector mySQLDatabaseConnector = new MySQLDatabaseConnector();  
 mySQLDatabaseConnector.updateEmployee(updateEmployee.getId(), updateEmployee.getRole(), updateEmployee.getZone());  
 }  
  
  
  
}

W klasie tej zaimplementowano konstruktory dla admina, uwzględniające różne scenariusze przekazywania danych. Dodatkowo, w klasie znajdują się metody ***addRole*** oraz ***changeRole***, z których pierwsza ma na celu dodanie nowej roli dla pracownika, a druga aktualizację roli pracownika w bazie danych poprzez wykorzystanie klasy ***MySQLDatabaseConnector.***

Klasa ClientHandler

package server;  
  
import java.io.IOException;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.net.Socket;  
import java.util.ArrayList;  
import java.util.List;  
  
public class ClientHandler implements Runnable {  
 private Socket clientSocket;  
 private Employee employee;  
  
 public ClientHandler(Socket socket) {  
 this.clientSocket = socket;  
 }  
  
 @Override  
 public void run() {  
 try {  
 ObjectOutputStream objectOutputStream = new ObjectOutputStream(clientSocket.getOutputStream());  
 ObjectInputStream objectInputStream = new ObjectInputStream(clientSocket.getInputStream());  
  
 String login = (String) objectInputStream.readObject();  
 String password = (String) objectInputStream.readObject();  
 Login log = new Login();  
 MySQLDatabaseConnector connector = new MySQLDatabaseConnector();  
  
 if (log.check(login, password)) {  
 employee = connector.getUserInfo(login, password);  
 log.startLogin(employee);  
 objectOutputStream.writeObject(employee);  
  
 if (employee instanceof Manager) {  
 List<Order> orders = employee.getListOfOrder();  
 List<Task> tasks = employee.getListOfTask();  
 List<Employee> employees = employee.getListOfEmployees();  
 List<Equipment> equipments = employee.getListOfEquipment();  
 objectOutputStream.writeObject(orders);  
 objectOutputStream.writeObject(tasks);  
 objectOutputStream.writeObject(employees);  
 objectOutputStream.writeObject(equipments);  
  
 String answer;  
 do {  
 answer = (String) objectInputStream.readObject();  
 if (answer.equals("orders")) {  
 Order order = (Order) objectInputStream.readObject();  
 String nameProduct = order.getProduct().getName();  
 List<Equipment> equipmentListOfTask = employee.getListOfEquipmentToTask(nameProduct);  
 List<Component> componentListOfTask = employee.getListOfComponentToTask(nameProduct);  
 List<Object> objects = new ArrayList<>();  
 objects.add(equipmentListOfTask);  
 objects.add(componentListOfTask);  
 objectOutputStream.writeObject(objects);  
 }  
 if (answer.equals("addTask")) {  
 Task task = (Task) objectInputStream.readObject();  
 ((Manager) employee).addNewTask(task, employee);  
 }  
 if (answer.equals("getUseEquipment")) {  
 int equipmentID = (int) objectInputStream.readObject();  
 List<List<String>> useEquipment = ((Manager) employee).getEquipmentTimeOfUseReport(equipmentID);  
 objectOutputStream.writeObject(useEquipment);  
 }  
 if (answer.equals("updateEquipment")) {  
 Equipment equipment = (Equipment) objectInputStream.readObject();  
 ((Manager) employee).changeEquipmentStatus(equipment);  
 }  
 if(answer.equals("Close")){  
 log.endLogin(employee);  
 }  
 } while (!answer.equals("Close"));  
 }  
  
 if (employee instanceof Admin) {  
 List<Employee> employees = ((Admin) employee).getListOfEmployees();  
 objectOutputStream.writeObject(employees);  
 objectOutputStream.writeObject(connector.getRolesList());  
 objectOutputStream.writeObject(connector.getZonesList());  
  
 String answer;  
 do {  
 answer = (String) objectInputStream.readObject();  
 if (answer.equals("Update")) {  
 Employee updateEmployee = (Employee) objectInputStream.readObject();  
 ((Admin) employee).changeRole(updateEmployee);  
 }  
 if(answer.equals("Close")){  
 log.endLogin(employee);  
 }  
 } while (!answer.equals("Close"));  
 }  
  
 if (employee instanceof ProductionEmployee) {  
 Task myTask = ((ProductionEmployee) employee).getMyTask(employee);  
 if (myTask == null) {  
 List<Task> tasksOfEmployee = ((ProductionEmployee) employee).getListOfTask(employee.getId());  
 objectOutputStream.writeObject(tasksOfEmployee);  
 } else {  
 List<Task> tasks = new ArrayList<>();  
 tasks.add(myTask);  
 objectOutputStream.writeObject(tasks);  
 }  
  
 }  
 } else {  
 objectOutputStream.writeObject(null);  
 }  
  
 clientSocket.close();  
 connector.closeConnection();  
 } catch (IOException | ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
}

Klasa ClientHandler pełni rolę obsługi klienta po stronie serwera w systemie opartym na gniazdach w języku Java. Głównym zadaniem tej klasy jest przyjęcie połączenia od klienta, obsługa przesyłanych danych oraz koordynacja interakcji między klientem a serwerem. Oto kilka kluczowych punktów dotyczących tej klasy:

1. Konstruktor: Klasa przyjmuje gniazdo klienta w konstruktorze, co umożliwia utworzenie instancji ClientHandler do obsługi konkretnej sesji komunikacyjnej.

2. Metoda run(implementacja interfejsu Runnable): Metoda ta definiuje logikę obsługi klienta w oddzielnym wątku. Zadania obejmują odbiór i wysyłanie danych między klientem a serwerem.

3. Strumienie danych: Klasa używa ObjectInputStream i ObjectOutputStream do przesyłania obiektów między klientem a serwerem. Wysyłane są informacje o logowaniu, obiekty pracownika, listy zadań, zamówień, itp.

4. Logowanie i interakcje w zależności od roli pracownika: W zależności od roli pracownika (Manager, Admin, ProductionEmployee), klasa obsługuje różne typy interakcji. Na przykład, dla Managera są przesyłane informacje o zamówieniach, zadaniach, a także obsługuje dodawanie nowych zadań czy aktualizację statusu sprzętu. Dla Admina udostępnia informacje o pracownikach oraz umożliwia aktualizację ich ról. Dla ProductionEmployee dostarcza informacje o zadaniach przypisanych do pracownika.

5. Zamknięcie połączenia: Po zakończeniu obsługi klienta, gniazdo jest zamykane, a także ewentualne połączenie z bazą danych przez obiekt MySQLDatabaseConnector.

Klasa ClientHandler odgrywa kluczową rolę w implementacji wieloklientowego systemu serwerowego, umożliwiając interakcję z różnymi rodzajami pracowników w zależności od ich ról w systemie.

Klasa Component

package server;  
  
public class Component extends Inventory{  
 public int quantity;  
 public Component(int id, String name, int quantity) {  
 super(id, name);  
 this.quantity = quantity;  
 }  
  
 @Override  
 public void reserve(Task task) {  
  
 }  
  
 @Override  
 public void cancelReservation(Task task) {  
  
 }  
 public void updateQuantity(int quantity){  
  
 }  
}

Klasa ta to „przepis” na to jak ma wyglądać każdy komponent. Znajdują się tu też metody przygotowane do wykorzystania w dalszym rozwoju aplikacji.

Klasa Employee

package server;  
  
import java.io.Serializable;  
import java.sql.\*;  
import java.util.ArrayList;  
import java.util.List;  
  
public class Employee implements Serializable {  
 private int id;  
 private String name;  
 private String lastName;  
 private String role;  
 private String zone;  
 private String login;  
 private String password;  
 private Task task;  
 private static final String *JDBC\_URL* = "jdbc:mysql://localhost:3306/system";  
 private static final String *USER* = "root";  
 private static final String *PASSWORD* = "";  
 private Connection connection;  
 private String startTime;  
  
 public String getStartTime() {  
 return startTime;  
 }  
  
 public void setStartTime(String startTime) {  
 this.startTime = startTime;  
 }  
  
 public Employee(int id, String name, String lastName, String role, String zone, String login, String password) {  
 this.id = id;  
 this.name = name;  
 this.lastName = lastName;  
 this.role = role;  
 this.zone = zone;  
 this.login = login;  
 this.password = password;  
  
 }  
 public Employee(int id, String name, String lastName, String role, String zone) {  
 this.id = id;  
 this.name = name;  
 this.lastName = lastName;  
 this.role = role;  
 this.zone = zone;  
 }  
 public Employee(int id, String name, String lastName, String role, String zone, String login, String password, Task task) {  
 this.id = id;  
 this.name = name;  
 this.lastName = lastName;  
 this.role = role;  
 this.zone = zone;  
 this.login = login;  
 this.password = password;  
 this.task = task;  
 }  
 public void getTask(Task task){  
  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public String getRole() {  
 return role;  
 }  
  
 public String getZone() {  
 return zone;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void acceptTask(Task task){  
  
 }  
 public void endTask(Task task){  
  
 }  
 public void addTaskResult(Task task, Result result){  
  
 }  
 public List<Order> getListOfOrder(){  
 List<Order> orders = new ArrayList<>();  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 String query = "SELECT \n" +  
 " oq.orderQuantityID,\n" +  
 " p.productID,\n" +  
 " p.name AS productName,\n" +  
 " o.Status AS orderStatus,\n" +  
 " oq.quantityOrdered,\n" +  
 " oq.QuantityInProduction,\n" +  
 " oq.OuantityFinished\n" +  
 "FROM \n" +  
 " orderQuantity oq\n" +  
 "JOIN \n" +  
 " product p ON oq.productID = p.productID\n" +  
 "JOIN \n" +  
 " orders o ON oq.orderID = o.OrderID;";  
 try {  
 PreparedStatement preparedStatement= connection.prepareStatement(query);  
 ResultSet result = preparedStatement.executeQuery();  
 while (result.next()){  
 int id = result.getInt("orderQuantityID");  
 int productID = result.getInt("productID");  
 String productName = result.getString("productName");  
 String orderStatus = result.getString("orderStatus");  
 int quantityOrdered = result.getInt("quantityOrdered");  
 int quantityInProduction = result.getInt("QuantityInProduction");  
 int quantityFinished = result.getInt("OuantityFinished");  
 orders.add(new Order(id, orderStatus,new Product(productID,productName,quantityOrdered,quantityInProduction,quantityFinished)));  
 }  
  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
 return orders;  
 }  
 public List<Employee> getListOfEmployees(){  
 List<Employee> employees = new ArrayList<>();  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 String query = "SELECT e.employeeID, e.name, e.lastName, r.roleName AS role, z.name AS zone\n" +  
 "FROM Employee e\n" +  
 "JOIN role r ON e.roleID = r.roleID\n" +  
 "JOIN zone z ON e.zoneID = z.zoneID;";  
 try {  
 PreparedStatement preparedStatement = connection.prepareStatement(query);  
 ResultSet result = preparedStatement.executeQuery();  
 while (result.next()) {  
 int id = result.getInt("employeeID");  
 String name = result.getString("name");  
 String lastName = result.getString("lastName");  
 String role = result.getString("role");  
 String zone = result.getString("zone");  
 employees.add(new Employee(id, name,lastName,role,zone));  
 }  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
 return employees;  
 }  
 public List<Task> getListOfTask(){  
 List<Task> tasks = new ArrayList<>();  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 String query = "SELECT \n" +  
 " t.taskID AS taskID,\n" +  
 " t.name AS name,\n" +  
 " t.priority,\n" +  
 " t.description,\n" +  
 " t.norm,\n" +  
 " z.name AS zone\_name,\n" +  
 " t.quantity\n" +  
 "FROM \n" +  
 " Task t\n" +  
 "JOIN \n" +  
 " Zone z ON t.zoneID = z.zoneID;\n";  
 try {  
 PreparedStatement preparedStatement = connection.prepareStatement(query);  
 ResultSet result = preparedStatement.executeQuery();  
 while (result.next()) {  
 int id = result.getInt("taskID");  
 String name = result.getString("name");  
 String priority = result.getString("priority");  
 String description = result.getString("description");  
 int norm = result.getInt("norm");  
 String zoneName = result.getString("zone\_name");  
 int quantity = result.getInt("quantity");  
 tasks.add(new Task(id, name,priority,description,norm,zoneName,quantity));  
 }  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
 return tasks;  
 }  
 public List<Equipment> getListOfEquipment(){  
 List<Equipment> equipments = new ArrayList<>();  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
  
 String query = "SELECT \n" +  
 " e.equipmentID AS equipmentID,\n" +  
 " e.name AS equipmentName,\n" +  
 " e.status,\n" +  
 " z.name AS zoneName\n" +  
 "FROM \n" +  
 " equipment e\n" +  
 "JOIN \n" +  
 " zone z ON e.zoneID = z.zoneID;";  
 try {  
 PreparedStatement preparedStatement = connection.prepareStatement(query);  
 ResultSet result = preparedStatement.executeQuery();  
  
 while (result.next()) {  
 int id = result.getInt("equipmentID");  
 String name = result.getString("equipmentName");  
 String status = result.getString("status");  
 String zoneName = result.getString("zoneName");  
 equipments.add(new Equipment(id, name,status,zoneName));  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
  
 return equipments;  
 }  
 public void setRole(String role) {  
 this.role = role;  
 }  
  
 public void setZone(String zone) {  
 this.zone = zone;  
 }  
  
 public List<Equipment> getListOfEquipmentToTask(String nameProduct) {  
 List<Equipment> equipmentList = new ArrayList<>();  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 String query="SELECT equipment.equipmentID,equipment.name, equipmentCategory.name AS equipment\_category, equipment.status, zone.name AS equipment\_zone\n" +  
 "FROM taskCategory\n" +  
 "JOIN taskEquipmentCategory ON taskCategory.taskCategoryID = taskEquipmentCategory.taskCategoryId\n" +  
 "JOIN equipment ON taskEquipmentCategory.equipmentCategoryId = equipment.equipmentCategoryId\n" +  
 "JOIN equipmentCategory ON equipment.equipmentCategoryId = equipmentCategory.equipmentCategoryID\n" +  
 "JOIN zone ON equipment.zoneId = zone.zoneID\n" +  
 "WHERE taskCategory.name =?;";  
 try {  
 PreparedStatement preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setString(1,nameProduct);  
 ResultSet result = preparedStatement.executeQuery();  
  
 while (result.next()) {  
 int id = result.getInt("equipmentID");  
 String nameComponent = result.getString("name");  
 String status = result.getString("status");  
 String zone = result.getString("equipment\_zone");  
 equipmentList.add(new Equipment(id, nameComponent, status, zone));  
  
  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return equipmentList;  
  
 }  
  
 public List<Component> getListOfComponentToTask(String nameProduct) {  
 List<Component> components = new ArrayList<>();  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 String query="SELECT component.componentID, Component.name, component.quantity \n" +  
 "FROM TaskCategory\n" +  
 "JOIN TaskCategoryComponent ON TaskCategory.taskCategoryID = TaskCategoryComponent.taskCategoryID\n" +  
 "JOIN Component ON Component.componentID = TaskCategoryComponent.componentID\n" +  
 "WHERE TaskCategory.name =?;";  
 try {  
 PreparedStatement preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setString(1,nameProduct);  
 ResultSet result = preparedStatement.executeQuery();  
  
 while (result.next()) {  
 int id = result.getInt("componentID");  
 String nameComponent = result.getString("name");  
 int quantity = result.getInt("quantity");  
 components.add(new Component(id,nameComponent,quantity));  
  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return components;  
 }  
}

Klasa Employee reprezentuje ogólnego pracownika w systemie i zawiera funkcje do zarządzania zadaniami, zamówieniami, pracownikami, wyposażeniem, itp.

Klasa ta stanowi bazę dla bardziej wyspecjalizowanych klas pracowników, takich jak Manager, Admin, i ProductionEmployee. Dziedziczenie pozwala na rozszerzenie funkcjonalności związanej z różnymi rolami pracowników.

W metodach tej klasy wykorzystano następujące zapytania do bazy danych:

"SELECT \n" +  
 " oq.orderQuantityID,\n" +  
 " p.productID,\n" +  
 " p.name AS productName,\n" +  
 " o.Status AS orderStatus,\n" +  
 " oq.quantityOrdered,\n" +  
 " oq.QuantityInProduction,\n" +  
 " oq.OuantityFinished\n" +  
 "FROM \n" +  
 " orderQuantity oq\n" +  
 "JOIN \n" +  
 " product p ON oq.productID = p.productID\n" +  
 "JOIN \n" +  
 " orders o ON oq.orderID = o.OrderID;";

To zapytanie SQL jest używane do pobierania informacji o ilości zamówionych produktów, ich statusie i ilości dostępnych w produkcji i ukończonych.

"SELECT e.employeeID, e.name, e.lastName, r.roleName AS role, z.name AS zone\n" +  
 "FROM Employee e\n" +  
 "JOIN role r ON e.roleID = r.roleID\n" +  
 "JOIN zone z ON e.zoneID = z.zoneID;";

Zapytanie zwraca informacje o pracownikach wraz z ich identyfikatorem, imieniem, nazwiskiem, nazwą roli i nazwą strefy, łącząc dane z tabel Employee, role i zone.

"SELECT \n" +  
 " t.taskID AS taskID,\n" +  
 " t.name AS name,\n" +  
 " t.priority,\n" +  
 " t.description,\n" +  
 " t.norm,\n" +  
 " z.name AS zone\_name,\n" +  
 " t.quantity\n" +  
 "FROM \n" +  
 " Task t\n" +  
 "JOIN \n" +  
 " Zone z ON t.zoneID = z.zoneID;\n";

Zwraca informacje o zadaniach, zawierając identyfikator zadania taskID, nazwę zadania, priorytet, opis, normę, nazwę strefy i ilość. Zadania są pobierane z tabeli Task, a informacje o strefie są pobierane z tabeli Zone poprzez połączenie na podstawie identyfikatora strefy zoneID.

"SELECT \n" +  
 " e.equipmentID AS equipmentID,\n" +  
 " e.name AS equipmentName,\n" +  
 " e.status,\n" +  
 " z.name AS zoneName\n" +  
 "FROM \n" +  
 " equipment e\n" +  
 "JOIN \n" +  
 " zone z ON e.zoneID = z.zoneID;";

To zapytanie SQL zwraca informacje o sprzęcie, obejmujące identyfikator sprzęt, nazwę sprzętu, status oraz nazwę strefy.

"SELECT equipment.equipmentID,equipment.name, equipmentCategory.name AS equipment\_category, equipment.status, zone.name AS equipment\_zone\n" +  
 "FROM taskCategory\n" +  
 "JOIN taskEquipmentCategory ON taskCategory.taskCategoryID = taskEquipmentCategory.taskCategoryId\n" +  
 "JOIN equipment ON taskEquipmentCategory.equipmentCategoryId = equipment.equipmentCategoryId\n" +  
 "JOIN equipmentCategory ON equipment.equipmentCategoryId = equipmentCategory.equipmentCategoryID\n" +  
 "JOIN zone ON equipment.zoneId = zone.zoneID\n" +  
 "WHERE taskCategory.name =?;";

Zwraca informacje o sprzęcie, które jest powiązane z kategorią zadania o określonej nazwie. Wybierane są kolumny: equipmentID, name, equipment\_category, status oraz equipment\_zone. Zapytanie to korzysta z wielu połączeń między różnymi tabelami: taskCategory, taskEquipmentCategory, equipment, equipmentCategory i zone. Warunek WHERE ogranicza wyniki do kategorii zadań o określonej nazwie (taskCategory.name = ?).

"SELECT component.componentID, Component.name, component.quantity \n" +  
 "FROM TaskCategory\n" +  
 "JOIN TaskCategoryComponent ON TaskCategory.taskCategoryID = TaskCategoryComponent.taskCategoryID\n" +  
 "JOIN Component ON Component.componentID = TaskCategoryComponent.componentID\n" +  
 "WHERE TaskCategory.name =?;";

Zwraca informacje o komponentach związanych z kategorią zadania o określonej nazwie.

Klasa Equipment

package server;  
  
public class Equipment extends Inventory{  
 private String status;  
 private String zone;  
 public Equipment(int id, String name, String status, String zone) {  
 super(id, name);  
 this.status = status;  
 this.zone = zone;  
 }  
  
 public String getStatus() {  
 return status;  
 }  
  
 public void setStatus(String status) {  
 this.status = status;  
 }  
  
 public String getZone() {  
 return zone;  
 }  
  
 @Override  
 public void reserve(Task task) {  
  
 }  
  
 @Override  
 public void cancelReservation(Task task) {  
  
 }  
}

Podobnie jak w przypadku klasy Component jest to „przepis” na sprzęt zawierający niezbędne pola, konstruktor i metody takie jak gettery i setery.

Klasa ta dziedziczy po klasie Inventory.

Klasa Inventory

package server;  
  
import java.io.Serializable;  
  
public abstract class Inventory implements Serializable {  
 private int id;  
 private String name;  
  
 public Inventory(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public abstract void reserve(Task task);  
 public abstract void cancelReservation(Task task);  
}

Jest to definicja abstrakcyjnej klasy o nazwie Inventory. Klasa ta implementuje interfejs Serializable, co oznacza, że instancje tej klasy mogą być serializowane. Klasa zawiera pola id oraz name. Posiada także konstruktor inicjalizujący te pola, a także metody umożliwiające rezerwację i anulowanie rezerwacji zasobów związanych z zadaniami. W skrócie, klasa Inventory służy jako abstrakcyjna podstawa dla różnych rodzajów inwentarzy umożliwiając rozszerzanie i implementowanie specyficznej logiki dla konkretnych typów inventarzy.

Klasa Leader

package server;  
  
import java.util.List;  
  
public class Leader extends Employee {  
 public Leader(int id, String name, String lastName, String role, String zone, String login, String password) {  
 super(id, name, lastName, role, zone, login, password);  
 }  
  
 public Leader(int id, String name, String lastName, String role, String zone, String login, String password, Task task) {  
 super(id, name, lastName, role, zone, login, password, task);  
 }  
  
  
  
 public List<Employee> getListOfEmployeeAndTask() {  
return null;  
 }  
  
 public void getTimeReport() {  
  
 }  
  
 public void getZoneReport() {  
  
 }  
public void getEmployeeReport(){  
  
}  
public void getEquipmentStatusReport(){  
  
}  
public void getEquipmentTimeOfUseReport(){  
  
}  
  
}

Klasa ta jest podstawą do wdrożenia operacji które może w naszym systemie wykonywać Leader. Klasa ta będzie rozwijana w dalszej części implementacji aplikacji.

Klasa License

package server;  
  
public class License {  
 private String id;  
 private String name;  
 private String description;  
  
 public License(String id, String name, String description) {  
 this.id = id;  
 this.name = name;  
 this.description = description;  
 }  
 public boolean checkLicense(Employee employee, Task task){  
 return true;  
 }  
}

Klasa ta to również “przepis” jak ma wyglądać i zachowywać się Licencja zawiera pola, konstruktor i metody niezbędne do zaimplementowania aplikacji.

Klasa Login

package server;  
import java.util.Date;  
  
import java.sql.\*;  
import java.text.SimpleDateFormat;  
  
public class Login {  
 private static final String *JDBC\_URL* = "jdbc:mysql://localhost:3306/system";  
 private static final String *USER* = "root";  
 private static final String *PASSWORD* = "";  
 private Connection connection;  
public Login(){  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
}  
 public boolean check(String login, String password){  
  
 String sqlQuery = "SELECT \* FROM employee WHERE login = ? AND password = ?";  
 boolean flag = false;  
 try (PreparedStatement preparedStatement = connection.prepareStatement(sqlQuery)) {  
 preparedStatement.setString(1, login);  
 preparedStatement.setString(2, password);  
 ResultSet result = preparedStatement.executeQuery();  
 if( result.next())  
 flag=true;  
  
 }catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return flag;  
 }  
 public void startLogin(Employee employee){  
 int id = employee.getId();  
 Date data = new Date();  
 SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
 String dataFormat = format.format(data);  
 employee.setStartTime(dataFormat);  
  
 String sqlQuery = "insert into loginHistory (employeeID,startTime) values (?,?)";  
 try {  
 PreparedStatement statement =connection.prepareStatement(sqlQuery);  
 statement.setInt(1,id);  
 statement.setString(2,dataFormat);  
 statement.executeUpdate();  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
 }  
 public void endLogin(Employee employee){  
 int id = employee.getId();  
 Date data = new Date();  
 SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
 String dataFormat = format.format(data);  
 String startTime = employee.getStartTime();  
  
 String sqlQuery = "update loginHistory SET startTime=?,endTime=? where employeeID=?";  
 try {  
 PreparedStatement statement =connection.prepareStatement(sqlQuery);  
 statement.setString(1,startTime);  
 statement.setString(2,dataFormat);  
 statement.setInt(3,id);  
 statement.executeUpdate();  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
 }  
}

Klasa Login obsługuje proces logowania pracowników do systemu. Wykorzystuje połączenie z bazą danych MySQL do weryfikacji loginu i hasła pracownika. Dodatkowo, rejestruje datę i czas rozpoczęcia oraz zakończenia sesji logowania w tabeli loginHistory.

Klasa Manager

package server;  
  
import java.sql.\*;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class Manager extends Leader{  
 private static final String *JDBC\_URL* = "jdbc:mysql://localhost:3306/system";  
 private static final String *USER* = "root";  
 private static final String *PASSWORD* = "";  
 private Connection connection;  
 public Manager(int id, String name, String lastName, String role, String zone, String login, String password) {  
 super(id, name, lastName, role, zone, login, password);  
 }  
  
 public Manager(int id, String name, String lastName, String role, String zone, String login, String password, Task task) {  
 super(id, name, lastName, role, zone, login, password, task);  
 }  
 public void addTaskCategory(String category){  
  
 }  
 public void addNewTask(Task task, Employee employee){  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 String query = "SELECT taskCategory.taskCategoryID\n" +  
 "FROM taskCategory\n" +  
 "WHERE taskCategory.name = (SELECT product.name FROM product WHERE product.productID = ?); ";  
 int productID = task.getProductID();  
 try {  
 PreparedStatement preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setInt(1,productID);  
 ResultSet resultSet = preparedStatement.executeQuery();  
 while (resultSet.next()) {  
 int taskCategoryID = resultSet.getInt("taskCategoryID");  
 query = "select zoneID from zone where name=?";  
 preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setString(1, task.getEquipment().getZone());  
 resultSet = preparedStatement.executeQuery();  
 while (resultSet.next()){  
 int zoneID = resultSet.getInt("zoneID");  
 query = "insert into task (name,priority,description, taskCategory,norm,productID,quantity,orderID,zoneID) values (?,?,?,?,?,?,?,?,?);";  
 preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setString(1, task.getName());  
 preparedStatement.setString(2, task.getPriority());  
 preparedStatement.setString(3, task.getDescription());  
 preparedStatement.setInt(4, taskCategoryID);  
 preparedStatement.setInt(5, task.getNorm());  
 preparedStatement.setInt(6, task.getProductID());  
 preparedStatement.setInt(7, task.getQuantity());  
 preparedStatement.setInt(8, task.getOrderID());  
 preparedStatement.setInt(9,zoneID);  
 preparedStatement.executeUpdate();  
 query = "SELECT taskID\n" +  
 "FROM task\n" +  
 "ORDER BY taskID DESC\n" +  
 "LIMIT 1;\n";  
 preparedStatement = connection.prepareStatement(query);  
 resultSet = preparedStatement.executeQuery();  
 while (resultSet.next()) {  
 int id = resultSet.getInt("taskID");  
 query = "UPDATE orderQuantity SET QuantityInProduction = QuantityInProduction + ?, orderID=2 WHERE orderQuantityID = ?;";  
 preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setInt(1, task.getQuantity());  
 preparedStatement.setInt(2, task.getOrderID());  
 preparedStatement.executeUpdate();  
 query = "insert into taskEquipment(taskID, equipmentID) values (?,?)";  
 preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setInt(1, id);  
 preparedStatement.setInt(2, task.getEquipment().getId());  
 preparedStatement.executeUpdate();  
 for (int i = 0; i < task.getComponent().size(); i++) {  
 query = "insert into taskComponent(taskID, componentID,quantity)values (?,?,?)";  
 preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setInt(1, id);  
 preparedStatement.setInt(2, task.getComponent().get(i).getId());  
 preparedStatement.setInt(3, task.getQuantity());  
 preparedStatement.executeUpdate();  
 }  
 query = "insert into result (resultID, quantityOK, quantityNOK) values (?,0,0)";  
 preparedStatement=connection.prepareStatement(query);  
 preparedStatement.setInt(1,id);  
 preparedStatement.executeUpdate();  
 query="update task set resultID=? where taskID=?";  
 preparedStatement=connection.prepareStatement(query);  
 preparedStatement.setInt(1,id);  
 preparedStatement.setInt(2,id);  
 preparedStatement.executeUpdate();  
 int employeeID = employee.getId();  
 query="insert into taskStatus(taskID,employeeID,stepName,startStep) values (?,?,'available',CURRENT\_TIMESTAMP)";  
 preparedStatement=connection.prepareStatement(query);  
 preparedStatement.setInt(1,id);  
 preparedStatement.setInt(2,employeeID);  
 preparedStatement.executeUpdate();  
  
 }  
 } }  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
 }  
 public List<List<String>> getEquipmentTimeOfUseReport(int id){  
 List<List<String>> useEquipment = new ArrayList<>();  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 String query = "SELECT\n" +  
 " employee.name,\n" +  
 " employee.lastName,\n" +  
 " taskStatus.startStep,\n" +  
 " taskStatus.endStep,\n" +  
 " task.name as taskName,\n" +  
 " TIMESTAMPDIFF(HOUR, taskStatus.startStep, taskStatus.endStep) AS hoursInUse\n" +  
 "FROM\n" +  
 " equipment\n" +  
 "JOIN taskEquipment ON taskEquipment.equipmentID = equipment.equipmentID\n" +  
 "JOIN task ON task.taskID = taskEquipment.taskID\n" +  
 "JOIN taskStatus ON taskStatus.taskID = task.taskID\n" +  
 "JOIN employee ON employee.employeeID = taskStatus.employeeID\n" +  
 "WHERE\n" +  
 " equipment.equipmentID = ?;\n";  
  
try{  
 PreparedStatement preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setInt(1,id);  
 ResultSet resultSet = preparedStatement.executeQuery();  
 while (resultSet.next()){  
 String name = resultSet.getString("name");  
 String lastName = resultSet.getString("lastName");  
 String startDate = resultSet.getString("startStep");  
 String endDate = resultSet.getString("endStep");  
 String taskName = resultSet.getString("taskName");  
 int use = resultSet.getInt("hoursInUse");  
 useEquipment.add(Arrays.*asList*(name,lastName,startDate,endDate,taskName,use+""));  
 }  
 for (List<String> innerList : useEquipment) {  
 for (String value : innerList) {  
 System.*out*.print(value + " ");  
 }  
 System.*out*.println();  
 }  
}catch (SQLException e){  
 e.printStackTrace();  
}  
return useEquipment;  
 }  
 public void addTaskPriority(Task task, String priority){  
  
 }  
 public void changeTaskPriority(Task task, String priority){  
  
 }  
 public void addLicenseToEmployee(Employee employee, License license){  
  
 }  
 public void addEquipment(Equipment equipment){  
  
 }  
 public void changeEquipmentStatus(Equipment equipment){  
 int id = equipment.getId();  
 String status = equipment.getStatus();  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 String query = "update equipment set status=? where equipmentID=?";  
 try{  
 PreparedStatement preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setString(1,status);  
 preparedStatement.setInt(2,id);  
 preparedStatement.executeUpdate();  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
 }  
  
  
}

Klasa Manager rozszerza klasę Leader i reprezentuje menadżera w systemie. Odpowiada za zarządzanie zadaniami, pracownikami, oraz monitorowanie używania sprzętu.

Metody:

1. addTaskCategory(String category): Dodaje nową kategorię zadania do systemu.

2. addNewTask(Task task, Employee employee): Dodaje nowe zadanie do systemu. W tym celu pobiera informacje związane z zadaniem, jego komponentami i przypisuje je do odpowiednich tabel w bazie danych.

3. getEquipmentTimeOfUseReport(int id): Generuje raport o czasie użytkowania danego sprzętu. Wykorzystuje zapytanie SQL, aby pobierać informacje o czasie rozpoczęcia, zakończenia i czasie trwania zadań, w których używano danego sprzętu.

Zapytania SQL:

- SELECT taskCategory.taskCategoryID FROM taskCategory WHERE taskCategory.name = (SELECT product.name FROM product WHERE product.productID = ?);

Pobiera ID kategorii zadania na podstawie nazwy produktu.

- INSERT INTO task (...) i następne zapytania.

Dodają nowe zadanie do bazy danych, wraz z informacjami o kategoriach, strefie, komponentach, rezultatach, statusie oraz przypisują je do pracownika.

- SELECT employee.name, employee.lastName, taskStatus.startStep, taskStatus.endStep, task.name AS taskName, TIMESTAMPDIFF(HOUR, taskStatus.startStep, taskStatus.endStep) AS hoursInUse FROM equipment. Pobiera informacje o czasie użytkowania sprzętu przez pracowników w ramach różnych zadań.

- UPDATE equipment SET status=? WHERE equipmentID=?;

Aktualizuje status sprzętu w bazie danych.

Klasa MySQLDatabaseConnector

package server;  
  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.util.ArrayList;  
import java.util.List;  
  
public class MySQLDatabaseConnector {  
  
 private static final String *JDBC\_URL* = "jdbc:mysql://localhost:3306/system";  
 private static final String *USER* = "root";  
 private static final String *PASSWORD* = "";  
 private Connection connection;  
  
 public MySQLDatabaseConnector() {  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public Employee getUserInfo(String user, String password) {  
 String sqlQuery = "SELECT \* FROM employee WHERE login = ? AND password = ?";  
  
 try (PreparedStatement preparedStatement = connection.prepareStatement(sqlQuery)) {  
 preparedStatement.setString(1, user);  
 preparedStatement.setString(2, password);  
 try (ResultSet result = preparedStatement.executeQuery()) {  
 if (result.next()) {  
 int id = result.getInt("employeeID");  
 String name = result.getString("name");  
 String lastName = result.getString("lastName");  
 int roleID = result.getInt("roleID");  
 int zoneID = result.getInt("zoneID");  
  
 sqlQuery = "Select roleName from role where roleID = ?";  
 try (PreparedStatement roleStatement = connection.prepareStatement(sqlQuery)) {  
 roleStatement.setInt(1, roleID);  
 try (ResultSet roleResult = roleStatement.executeQuery()) {  
 if (roleResult.next()) {  
 String roleName = roleResult.getString("roleName");  
  
 sqlQuery = "Select name from zone where zoneID = ?";  
 try (PreparedStatement zoneStatement = connection.prepareStatement(sqlQuery)) {  
 zoneStatement.setInt(1, zoneID);  
 try (ResultSet zoneResult = zoneStatement.executeQuery()) {  
 if (zoneResult.next()) {  
 String zoneName = zoneResult.getString("name");  
 if(roleName.equals("Production Employee"))  
 return new ProductionEmployee(id, name, lastName,roleName,zoneName,user,password );  
 else if (roleName.equals("Admin"))  
 return new Admin(id, name, lastName,roleName,zoneName,user,password);  
 else if (roleName.equals("Leader"))  
 return new Leader(id, name, lastName,roleName,zoneName,user,password);  
 else if (roleName.equals("Manager")) {  
 return new Manager(id, name, lastName,roleName,zoneName,user,password);  
 }else  
 return new ProductionEmployee(id, name, lastName,roleName,zoneName,user,password);  
  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return null;  
 }  
 public List<String> getRolesList(){  
 List<String> roles = new ArrayList<>();  
 String sqlQuery = "SELECT roleName from role";  
 try {  
 PreparedStatement preparedStatement = connection.prepareStatement(sqlQuery);  
 ResultSet result = preparedStatement.executeQuery();  
 while (result.next()) {  
  
 String name = result.getString("roleName");  
  
 roles.add(name);  
 }  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
 return roles;  
 }  
 public List<String> getZonesList(){  
 List<String> zones = new ArrayList<>();  
 String sqlQuery = "SELECT name from zone";  
 try {  
 PreparedStatement preparedStatement = connection.prepareStatement(sqlQuery);  
 ResultSet result = preparedStatement.executeQuery();  
 while (result.next()) {  
  
 String name = result.getString("name");  
  
 zones.add(name);  
 }  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
 return zones;  
 }  
 public void closeConnection() {  
 try {  
 if (connection != null && !connection.isClosed()) {  
 connection.close();  
 System.*out*.println("Połączenie z bazą danych zostało zamknięte.");  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void updateEmployee(int id, String role, String zone) {  
 String sqlQurey = "SELECT roleID FROM role WHERE roleName=?";  
 try {  
 PreparedStatement preparedStatement = connection.prepareStatement(sqlQurey);  
 preparedStatement.setString(1,role);  
 ResultSet resultSet = preparedStatement.executeQuery();  
 if (resultSet.next()){  
 int idRole = resultSet.getInt("roleID");  
  
 sqlQurey = "SELECT zoneID from zone where name=?";  
 preparedStatement = connection.prepareStatement(sqlQurey);  
 preparedStatement.setString(1,zone);  
 resultSet = preparedStatement.executeQuery();  
 if (resultSet.next()) {  
 int idZone = resultSet.getInt("zoneID");  
  
 sqlQurey = "UPDATE employee set roleID= ?, zoneID= ? where employeeID= ?";  
 preparedStatement = connection.prepareStatement(sqlQurey);  
 preparedStatement.setInt(1, idRole);  
 preparedStatement.setInt(2,idZone);  
 preparedStatement.setInt(3,id);  
 preparedStatement.executeUpdate();  
 }}  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
 }  
}

Klasa MySQLDatabaseConnector służy do komunikacji z bazą danych MySQL w systemie.

1. Konstruktor:

- Inicjalizuje połączenie z bazą danych przy użyciu danych dostępowych (URL, użytkownik, hasło).

- Wyświetla komunikat o pomyślnym połączeniu lub w przypadku błędu wyświetla stosowny wyjątek.

2. getUserInfo(String user, String password)

- Pobiera informacje o pracowniku na podstawie podanego loginu i hasła.

- Wykorzystuje zagnieżdżone zapytania SQL do pobrania roli i strefy przypisanej do pracownika.

3.getRolesList()

- Pobiera listę ról dostępnych w systemie.

4. getZonesList()

- Pobiera listę stref dostępnych w systemie.

5. closeConnection()

- Zamyka połączenie z bazą danych.

6. updateEmployee(int id, String role, String zone)

- Aktualizuje dane pracownika (rolę i strefę) na podstawie podanych parametrów.

Zapytania SQL:

1. Pobranie informacji o pracowniku:

- Zapytanie weryfikujące login i hasło pracownika.

- Następnie pobierane są dodatkowe informacje, takie jak rola i strefa pracownika.

2. Pobranie listy ról:

- zapytanie pobierające nazwy ról z tabeli `role`.

3. Pobranie listy stref:

- zapytanie pobierające nazwy stref z tabeli `zone`.

4. Aktualizacja danych pracownika:

- Wyszukuje ID roli na podstawie nazwy roli.

- Wyszukuje ID strefy na podstawie nazwy strefy.

- Aktualizuje dane pracownika (rolę i strefę) w tabeli `employee`.

Klasa Order

package server;  
  
import java.io.Serializable;  
  
public class Order implements Serializable {  
 private int id;  
 private String status;  
 private Product product;  
  
 public Order(int id, String status, Product product) {  
 this.id = id;  
 this.status = status;  
 this.product = product;  
 }  
  
 public Product getProduct() {  
 return product;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setStatus(String status) {  
 this.status = status;  
 }  
  
 public String getStatus() {  
 return status;  
 }  
}

Klasa Order reprezentuje zamówienie w systemie, zawierając informacje o identyfikatorze zamówienia, statusie oraz przypisanym do niego produkcie, umożliwiając manipulację danymi związanych z zamówieniem.

Klasa Product

package server;  
  
import java.io.Serializable;  
  
public class Product implements Serializable {  
 private int id;  
 private String name;  
 private int quantityOrdered;  
 private int quantityInProduction;  
 private int quantityFinished;  
  
 public Product( String name, int quantityOrdered) {  
 this.name = name;  
 this.quantityOrdered = quantityOrdered;  
 }  
  
 public Product( int id, String name, int quantityOrdered, int quantityInProduction, int quantityFinished) {  
 this.id = id;  
 this.name = name;  
 this.quantityOrdered = quantityOrdered;  
 this.quantityInProduction = quantityInProduction;  
 this.quantityFinished = quantityFinished;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int getQuantityOrdered() {  
 return quantityOrdered;  
 }  
  
 public int getQuantityInProduction() {  
 return quantityInProduction;  
 }  
  
 public int getQuantityFinished() {  
 return quantityFinished;  
 }  
  
 @Override  
 public String toString() {  
 return name +  
 "\t" + quantityOrdered +  
 "\t" + quantityInProduction +  
 "\t" + quantityFinished ;  
 }  
  
 public void setQuantityInProduction(int quantityInProduction) {  
 this.quantityInProduction = quantityInProduction;  
 }  
  
 public int getId() {  
 return id;  
 }  
}

Klasa Product reprezentuje produkt w systemie, przechowując informacje o identyfikatorze, nazwie, ilości zamówionej, ilości w produkcji oraz ilości ukończonej. Klasa umożliwia dostęp do tych danych oraz manipulację nimi, a także zawiera metodę toString(), która zwraca reprezentację tekstową produktu.

Klasa ProductionEmployee

package server;  
  
import java.sql.\*;  
import java.util.ArrayList;  
import java.util.List;  
  
public class ProductionEmployee extends Employee{  
 private static final String *JDBC\_URL* = "jdbc:mysql://localhost:3306/system";  
 private static final String *USER* = "root";  
 private static final String *PASSWORD* = "";  
 private Connection connection;  
 public ProductionEmployee(int id, String name, String lastName, String role, String zone, String login, String password) {  
 super(id, name, lastName, role, zone, login, password);  
 }  
  
 public ProductionEmployee(int id, String name, String lastName, String role, String zone, String login, String password, Task task) {  
 super(id, name, lastName, role, zone, login, password, task);  
 }  
  
 public Task getMyTask(Employee employee)  
 {  
 Task task = null;  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 String query = "SELECT \n" +  
 " task.taskID,\n" +  
 " MAX(task.name) as taskName,\n" +  
 " MAX(task.priority) as taskPriority,\n" +  
 " MAX(task.description) as taskDescription,\n" +  
 " MAX(taskCategory.name) as nameCategory,\n" +  
 " MAX(task.norm) as taskNorm,\n" +  
 " MAX(result.quantityOK) as quantityOK,\n" +  
 " MAX(result.quantityNOK) as quantityNOK,\n" +  
 " MAX(taskStatus.stepName) as status,\n" +  
 " MAX(product.name) as productName,\n" +  
 " MAX(task.quantity) as taskQuantity,\n" +  
 " MAX(zone.name) as zoneName,\n" +  
 " MAX(equipment.name) as equipmentName,\n" +  
 " GROUP\_CONCAT(component.name) as componentName \n" +  
 "FROM \n" +  
 " task\n" +  
 "JOIN \n" +  
 " taskStatus ON taskStatus.taskID = task.taskID\n" +  
 "JOIN \n" +  
 " taskCategory ON taskCategory.taskCategoryID = task.taskCategory\n" +  
 "JOIN \n" +  
 " result ON result.resultID = task.resultID\n" +  
 "JOIN \n" +  
 " product ON product.productID = task.productID\n" +  
 "JOIN \n" +  
 " zone ON zone.zoneID = task.zoneID\n" +  
 "JOIN \n" +  
 " taskEquipment ON taskEquipment.taskID = task.taskID\n" +  
 "JOIN \n" +  
 " equipment ON equipment.equipmentID = taskEquipment.equipmentID\n" +  
 "JOIN \n" +  
 " taskComponent ON taskComponent.taskID = task.taskID\n" +  
 "JOIN \n" +  
 " component ON component.componentID = taskComponent.componentID\n" +  
 "WHERE \n" +  
 " taskStatus.employeeID = ?\n" +  
 " AND taskStatus.stepName = 'in progress' \n" +  
 " AND taskStatus.endStep = '0000-00-00 00-00-00'\n" +  
 "GROUP BY \n" +  
 " task.taskID;\n";  
 try{  
 int employeeID = employee.getId();  
 PreparedStatement preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setInt(1,employeeID);  
 ResultSet resultSet = preparedStatement.executeQuery();  
 while (resultSet.next()){  
 int taskID = resultSet.getInt("taskID");  
 String taskName = resultSet.getString("taskName");  
 String priority = resultSet.getString("taskPriority");  
 String description = resultSet.getString("taskDescription");  
 String category = resultSet.getString("nameCategory");  
 int norm = resultSet.getInt("taskNorm");  
 int quantityOK = resultSet.getInt("quantityOK");  
 int quantityNOK = resultSet.getInt("quantityNOK");  
 String status = resultSet.getString("status");  
 String product = resultSet.getString("productName");  
 int quantity = resultSet.getInt("taskQuantity");  
 String zone = resultSet.getString("zoneName");  
 String equipment = resultSet.getString("equipmentName");  
 String component = resultSet.getString("componentName");  
 task = new Task(taskID,taskName,priority,description,category,norm,quantityOK,quantityNOK,status,product,quantity,zone,equipment,component);  
  
  
 }  
 }catch (SQLException e){  
 e.printStackTrace();  
 }  
return task;  
 }  
 public List<Task> getListOfTask(int employeeID) {  
 List<Task> tasks = new ArrayList<>();  
 try {  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
 connection = DriverManager.*getConnection*(*JDBC\_URL*, *USER*, *PASSWORD*);  
 System.*out*.println("Pomyślnie połączono z bazą danych");  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 String query = "SELECT \n" +  
 " t.taskID,\n" +  
 " t.name,\n" +  
 " t.priority,\n" +  
 " t.description,\n" +  
 " p.name as product,\n" +  
 " t.quantity,\n" +  
 " t.norm,\n" +  
 " ts.stepName as status,\n" +  
 " GROUP\_CONCAT(DISTINCT eq.name) as equipment,\n" +  
 " GROUP\_CONCAT(DISTINCT c.name) as component\n" +  
 "FROM \n" +  
 " task t\n" +  
 "JOIN \n" +  
 " product p ON p.productID = t.productID \n" +  
 "JOIN \n" +  
 " taskstatus ts ON ts.taskID = t.taskID \n" +  
 "JOIN \n" +  
 " taskequipment te ON te.taskID = t.taskID\n" +  
 "JOIN \n" +  
 " equipment eq ON eq.equipmentID = te.equipmentID\n" +  
 "JOIN \n" +  
 " taskcomponent tc ON tc.taskID = t.taskID\n" +  
 "JOIN \n" +  
 " component c ON c.componentID = tc.componentID\n" +  
 "WHERE \n" +  
 " ts.endStep = '0000-00-00 00:00:00' AND ts.stepName = 'available'\n" +  
 " AND t.taskCategory IN (\n" +  
 " SELECT DISTINCT taskcategoryid \n" +  
 " FROM taskcategorylicense \n" +  
 " WHERE licenseID IN (\n" +  
 " SELECT licenseId \n" +  
 " FROM employeelicense \n" +  
 " WHERE employeeId = ? AND expirationDate >= CURRENT\_DATE\n" +  
 " )\n" +  
 " )\n" +  
 " AND eq.equipmentCategoryID IN (\n" +  
 " SELECT equipmentcategoryId \n" +  
 " FROM equipmentcategorylicense \n" +  
 " WHERE licenseID IN (\n" +  
 " SELECT licenseId \n" +  
 " FROM employeelicense \n" +  
 " WHERE employeeId = ? AND expirationDate >= CURRENT\_DATE\n" +  
 " )\n" +  
 " )\n" +  
 " \n" +  
 "GROUP BY \n" +  
 " t.taskID;\n";  
 try {  
 PreparedStatement preparedStatement = connection.prepareStatement(query);  
 preparedStatement.setInt(1, employeeID);  
 preparedStatement.setInt(2,employeeID);  
 ResultSet resultSet = preparedStatement.executeQuery();  
 while (resultSet.next()) {  
 int taskID = resultSet.getInt("taskID");  
 String name = resultSet.getString("name");  
 String priority = resultSet.getString("priority");  
 String description = resultSet.getString("description");  
 String productName = resultSet.getString("product");  
 int quantity = resultSet.getInt("quantity");  
 int norm = resultSet.getInt("norm");  
 String status = resultSet.getString("status");  
 String equipment = resultSet.getString("equipment");  
 String component = resultSet.getString("component");  
 tasks.add(new Task(taskID, name, priority, description, productName, quantity, norm, status, equipment, component));  
 }  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return tasks;  
 }  
}

Klasa ProductionEmployee rozszerza klasę Employee i odpowiada za funkcjonalność pracownika produkcyjnego w systemie. Metody tej klasy umożliwiają pobranie zadania przypisanego do danego pracownika w trakcie produkcji oraz uzyskanie listy dostępnych zadań, które może on podjąć.

1. Konstruktory:

- Konstruktory inicjalizują obiekt klasy ProductionEmployee przy użyciu danych przekazanych jako parametry, w tym również konstruktor z dodatkowym zadaniem Task.

2. getMyTask(Employee employee)

- Pobiera zadanie przypisane do danego pracownika w trakcie produkcji.

- Wykorzystuje zagnieżdżone zapytanie SQL, grupowanie i funkcje agregujące.

3. getListOfTask(int employeeID)

- Pobiera listę dostępnych zadań, które pracownik produkcyjny może podjąć.

- Wykorzystuje zapytanie SQL z warunkami filtrującymi zadania dostępne dla pracownika w zależności od licencji na kategorie zadań, sprzętu i komponentów.

Klasa Result

package server;  
  
public class Result {  
 private int quantityOK;  
 private int quantityNOK;  
 private String information;  
  
 public Result(int quantityOK, int quantityNOK, String information) {  
 this.quantityOK = quantityOK;  
 this.quantityNOK = quantityNOK;  
 this.information = information;  
 }  
}

Klasa Result reprezentuje wynik produkcji, przechowując informacje o ilości poprawnie i niepoprawnie wykonanych elementów oraz ewentualnych dodatkowych informacjach, umożliwiając zapis i manipulację wynikami produkcyjnymi.

Klasa ServerTCP

package server;  
  
import java.io.IOException;  
import java.io.ObjectOutputStream;  
import java.io.PrintWriter;  
import java.net.ServerSocket;  
import java.net.Socket;  
import java.util.List;  
import java.util.Scanner;  
import java.util.concurrent.\*;  
  
public class ServerTCP {  
 private static final int *MAX\_CLIENTS* = 100;  
  
 public static void main(String[] args) {  
 ExecutorService executorService = Executors.*newFixedThreadPool*(*MAX\_CLIENTS*);  
 try (ServerSocket serverSocket = new ServerSocket(12345);) {  
 System.*out*.println("Serwer nasłuchuje na porcie 12345");  
 while (true) {  
 Socket clientSocket = serverSocket.accept();  
 System.*out*.println("Połączono z " + clientSocket);  
 executorService.execute(new ClientHandler(clientSocket));  
  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 executorService.shutdown();  
 }  
 }  
}

Klasa ServerTCP reprezentuje prosty serwer obsługujący połączenia z wieloma klientami przy użyciu protokołu TCP. Serwer nasłuchuje na porcie 12345 i obsługuje przychodzące połączenia, uruchamiając dla każdego z nich nowy wątekClientHandler

1. Polę MAX\_CLIENTS

- Stała definiująca maksymalną liczbę obsługiwanych klientów.

2. Metoda main

- Metoda główna programu, uruchamiająca serwer.

- Tworzy pulę wątków o stałej wielkości ExecutorService do obsługi wielu klientów jednocześnie.

- Tworzy obiekt ServerSocket nasłuchujący na porcie 12345.

- W nieskończonej pętli akceptuje przychodzące połączenia i dla każdego klienta tworzy nowy wątek ClientHandler, który zajmuje się obsługą komunikacji z danym klientem.

3. Pętla while:

- Akceptuje nowe połączenia od klientów.

- Dla każdego klienta tworzony jest nowy wątek ClientHandler i przekazywany mu jest gniazdo klienta Socket.

4. Zarządzanie wątkami:

- Wątki obsługujące klientów są uruchamiane przy użyciu puli wątków, co umożliwia równoczesną obsługę wielu klientów.

- Po zakończeniu programu, pulę wątków zamyka się w bloku finally

Klasa Task

package server;  
  
import java.io.Serializable;  
import java.sql.Time;  
import java.util.List;  
  
public class Task implements Serializable {  
 private String componentName;  
 private String equipmentName;  
 private String product;  
 private int quantityNOK;  
 private int quantityOK;  
 private int taskID;  
 private String name;  
 private String priority;  
 private String description;  
 private int norm;  
 private List<Component> component;  
 private Equipment equipment;  
 private String status;  
 private Time timeInStep;  
 private Employee employee;  
 private String zone;  
 private int quantity;  
 private int productID;  
 private int orderID;  
 private String category;  
  
  
 public Task(int taskID, String name, String priority, String description, int norm, String zone, int quantity) {  
 this.taskID = taskID;  
 this.name = name;  
 this.priority = priority;  
 this.description = description;  
 this.norm = norm;  
 this.zone=zone;  
 this.quantity=quantity;  
 }  
 public Task(int taskID, String name, String priority, String description, int norm, List<Component> component, Equipment equipment, String zone, int quantity, int productID, int orderID) {  
 this.taskID = taskID;  
 this.name = name;  
 this.priority = priority;  
 this.description = description;  
 this.norm = norm;  
 this.component = component;  
 this.equipment=equipment;  
 this.zone=zone;  
 this.quantity=quantity;  
 this.productID=productID;  
 this.orderID=orderID;  
 }  
  
 public Task(int taskID, String name, String priority, String description, int norm, List<Component> component, Equipment equipment, String status, Time timeInStep, Employee employee) {  
 this.taskID = taskID;  
 this.name = name;  
 this.priority = priority;  
 this.description = description;  
 this.norm = norm;  
 this.component = component;  
 this.equipment = equipment;  
 this.status = status;  
 this.timeInStep = timeInStep;  
 this.employee = employee;  
 }  
  
 public Task(int taskID, String taskName, String priority, String description, String category, int norm, int quantityOK,  
 int quantityNOK, String status,String product, int quantity, String zone, String equipment, String component) {  
 this.taskID=taskID;  
 this.name=taskName;  
 this.priority=priority;  
 this.description=description;  
 this.category = category;  
 this.norm=norm;  
 this.quantityOK = quantityOK;  
 this.quantityNOK = quantityNOK;  
 this.product = product;  
 this.quantity = quantity;  
 this.zone =zone;  
 this.equipmentName = equipment;  
 this.componentName = component;  
 this.status=status;  
  
  
 }  
  
 public Task(int taskID, String name, String priority, String description, String productName, int quantity, int norm, String status, String equipment, String component) {  
 this.taskID=taskID;  
 this.name=name;  
 this.priority=priority;  
 this.description=description;  
 this.norm=norm;  
 this.product = productName;  
 this.quantity = quantity;  
 this.equipmentName = equipment;  
 this.componentName = component;  
 this.status = status;  
 }  
  
 public String getStatus() {  
 return status;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int getProductID() {  
 return productID;  
 }  
  
 public int getOrderID() {  
 return orderID;  
 }  
  
 public String getPriority() {  
 return priority;  
 }  
  
 public String getDescription() {  
 return description;  
 }  
  
 public int getNorm() {  
 return norm;  
 }  
  
 public String getZone() {  
 return zone;  
 }  
  
 public List<Component> getComponent() {  
 return component;  
 }  
  
 public Equipment getEquipment() {  
 return equipment;  
 }  
  
 public int getQuantity() {  
 return quantity;  
 }  
  
 public int getTaskID() {  
 return taskID;  
 }  
 public void viewTaskDetails(){  
  
 }  
  
 public int getId() {  
 return taskID;  
 }  
}

Klasa Task reprezentuje zadanie w systemie produkcyjnym i przechowuje informacje takie jak identyfikator zadania, nazwa, priorytet, opis, norma, lista komponentów, przypisane urządzenie, status, czas w danym etapie, pracownik oraz inne parametry związane z produkcją. Klasa ta została zaprojektowana do obsługi różnych aspektów zadań, takich jak ich tworzenie, przypisywanie, monitorowanie postępu czy przeglądanie szczegółów.

W jej implementacji znajdują się różne konstruktory, umożliwiające tworzenie obiektów Task z różnym zestawem informacji zależnie od kontekstu. Metody dostępowe umożliwiają pobieranie danych z obiektu.

Opis działania aplikacji:

**Podsumowanie i wnioski:**

Projekt Systemu Zarządzania Produkcją udanie zrealizował główne wymagania dotyczące różnych aspektów funkcjonalnych oraz technicznych. Poniżej przedstawiamy podsumowanie osiągnięć oraz pewne aspekty, które mogą wymagać dalszej uwagi i rozwoju:

Osiągnięcia:

1. Łączność TCP:

- Projekt zakładał użycie protokołu TCP do komunikacji między klientem a serwerem. Zastosowanie ServerSocket i Socket zostało prawidłowo zaimplementowane, umożliwiając nawiązywanie połączeń między wieloma klientami a serwerem.

2. Aplikacja Klient-Serwer:

- Struktura klienta i serwera została poprawnie zaprojektowana, umożliwiając obsługę wielu klientów równocześnie. Komunikacja między nimi obejmuje przesyłanie obiektów, takich jak zadania czy informacje o użytkownikach.

3.Wielowątkowość:

- Wykorzystanie ExecutorService do obsługi wielu klientów jednocześnie zapewnia efektywną wielowątkowość w serwerze.

4. Interfejs Użytkownika (UI):

- W projekcie zaimplementowano podstawowy interfejs graficzny użytkownika co było jednym z wymagań projektu.

5. Użycie bazy danych

Projekt Systemu Zarządzania Produkcją skutecznie wykorzystuje bazę danych MySQL do przechowywania i pobierania danych związanych z pracownikami, zadaniami, produktami, zamówieniami, wynikami oraz innymi zależnościami w kontekście zarządzania produkcją.

Punkty do Dalszego Rozwoju:

1. Brak Wyświetlania Zadań w UI Pracownika Produkcji:

Nie wyświetlają się zadania w UI Pracownika, a także nie umożliwia zakończenia bieżącego zadania i wyboru nowego. To jest obszar, który wymaga dalszego rozwoju, aby umożliwić efektywne zarządzanie zadaniami przez pracowników produkcji.

2. Niedopracowane UI:

- Kod nie dostarcza pełnej implementacji interfejsu użytkownika. Dalsze prace nad dostosowaniem UI do wymagań biznesowych są konieczne.

Podsumowując:

Projekt oferuje solidną podstawę, ale dla pełnego wdrożenia jako system zarządzania produkcją, wymaga dalszej pracy nad interfejsem użytkownika, funkcjonalnościami związanymi z zarządzaniem zadaniami, a także bardziej rozbudowanym systemem komunikacyjnym i zabezpieczeniami.