Bazy Danych 2
Projekt
Etap 4

Kajetan Olbryt - 250993
Bartosz Sulima - 234310
Konrad Bieganowski - 252710

4 lutego 2022

Spis treści

1 Temat i cel projektu

Temat: "Bazodanowy system obsługi restauracji"

Cel projektu: Zaprojektowanie oraz implementacja bazy danych wraz z interfejsem użytkownika przeznaczonych do obsługiwania systemu restauracji.

2 Opis wymagań jakie powinna spełniać aplikacja ustalone na etapie projektowania

2.1 Wymagania funkcjonalne

- Klienci będą mieli możliwość zobaczenia aktualnego menu.
- Klienci będą mieli możliwość wybrania pozycji z listy i złożenia zamówienia.
- Klienci po złożeniu zamówienia muszą dokonać płatności.
- Klienci będą mieli możliwość rejestracji do systemu.
- Klienci będą mieli możliwość logowania do systemu.
- Klienci mogą zdalnie zarezerwować stolik.
- System może generować tygodniowy grafik pracy dla pracowników nie przebywających na urlopie.
- Pracownicy mogą wyświetlić swój grafik pracy.
- Pracownicy mają możliwość zalogowania się do systemu.
- Kucharze będą mieli wgląd na wybór dań złożonych przez klientów i na skład tych dań.
- Kucharze będą mogli zmieniać stan zamówień z "W trakcie przygotowania" na "Gotowe do odbioru".
- Kelnerzy mogą przeglądać stan stolików i rezerwacji.
- Kelnerzy mogą zająć stolik dla klientów którzy przyszli bez rezerwacji.
- Kelnerzy będą mieli wgląd na status zamówień na miejscu.
- Kelnerzy będą mogli zmieniać stan zamówień z "Gotowe do odbioru" na "Dostarczone".
- Dostawcy będą mieli wgląd na status zamówień na wynos.
- Dostawcy będą mogli zmieniać stan zamówień z "Gotowe do odbioru" na "W drodze" i z "W drodze" na "Dostarczone".
- Szef kuchni będzie dodawać pozycje do menu.
- Szef kuchni będzie usuwać pozycję z menu.
- Szef kuchni będzie edytować poszczególne pozycje w menu.
- Kierownik będzie dodawał pracowników..
- Kierownik będzie usuwał pracowników.

- Kierownik będzie edytował dane pracowników.
- Kierownik będzie dodawał zamówienia do hurtowni.
- Kierownik będzie generował historię zamówień.

2.2 Wymagań niefunkcjonalne

- Musi działać na systemie Windows 10.
- Dostęp do bazy danych powinien być możliwy przez połączenie internetowe.
- Kierownik oraz szef kuchni może być tylko jeden.
- Aby korzystać z systemu, pracownicy muszą być zalogowani.
- Klient przeglądając menu stacjonarnie lub zdalnie, nie musi być zalogowany.
- Klient zamawiając posiłek na dowóz, musi być zalogowany.
- Pracownicy powinni mieć dostęp tylko do tych funkcji które wynikają z ich stanowiska.
- System powinien rejestrować każde logowanie w celu kontroli bezpieczeństwa.
- Hasła przechowywane w bazie danych muszą być hashowane.
- System powinien sprawdzać czy dany użytkownik ma uprawnienia do wykonania pożądanej akcji.

3 Podział na role

Kajetan Olbryt - Programista, pomoc przy tworzeniu bazy danych i zapytań, pomoc w połączeniu bazy danych z programem użytkowym, tworzenie programu użytkowego.

Bartosz Sulima - Programista, tworzenie bazy danych i zapytań, pomoc w połączeniu bazy danych z programem użytkowym, pomoc w tworzeniu programu użytkowego.

Konrad Bieganowski - Programista, pomoc przy tworzeniu bazy danych i zapytań, połączenie bazy danych z programem użytkowym, pomoc w tworzeniu programu użytkowego.

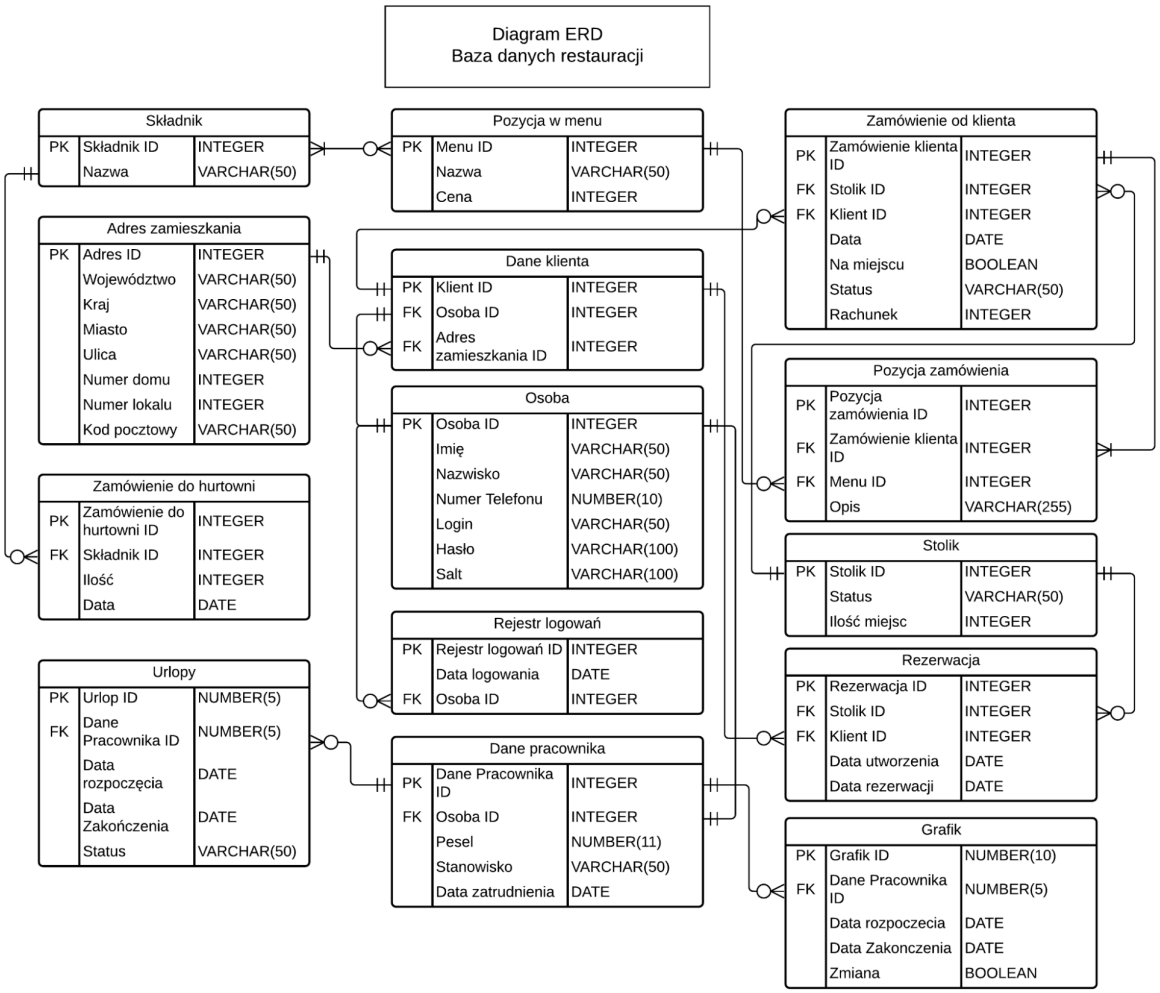
4 Zastosowana Architektura

Zastosowana Architektura to Architektura trójwarstwowa (ang. three-tier architecture lub three-layer architecture) – architektura typu klient-serwer, w której interfejs użytkownika, przetwarzanie danych i składowanie danych są rozwijane w postaci osobnych modułów.

5 Zastosowana technologia

Baza danych jest obsługiwana za pośrednictwem Java DataBase Conectivity (JBDC). Interfejs użytkownika został zrealizowany w postaci aplikacji konsolowej w języku Java. Został wykorzystany framework PostgreSQL.

6 Diagram relacji encji



7 Struktura projektu

Projekt składa się z:

- Bazy danych wykonanej w PostgreSQL.
- Aplikacji Serwerowej, której zadaniem jest odczytywanie informacji przesyłanych przez aplikacje użytkowe, interpretowanie tych informacji, wykonywaniu z pomocą tych informacji interakcji z bazą danych i ostatecznie przesłanie informacji zwrotnej do aplikacji użytkowej.
- Aplikacji Klienckiej, której zadaniem jest obsługa klienta **nie przebywającego** fizycznie w restauracji. Aplikacja ta umożliwia rejestracje i logowanie klientów, którzy po zalogowaniu mogą wyświetlać aktualne menu, zarezerwować stolik w restauracji, złożyć zdalne zamówienia i je opłacić.
- Aplikacji Stołowej, której zadaniem jest obsługa klienta **przebywającego** fizycznie w restauracji. Aplikacja ta umożliwia wyświetlanie aktualnego menu, a także składanie lokalnych zamówień i ich opłacenie.
- Aplikacji Pracowniczej, której zadaniem jest obsługa pracowników restauracji. Aplikacja umożliwia pracowniką zalogowanie się do systemu po czym wyświetla menu z funkcjami ospowiednimi do ich stanowisk.

8 Implementacja

8.1 Hashowanie haseł

Hashowanie haseł przebiega z użyciem algorytmu MD5 z użyciem "salt". Dla każdej nowej osoby w bazie danych generowana jest nowa wartość "salt" której używamy razem z hasłem użytkownika do wygenerowania zahashowanego hasła które przechowujemy w bazie danych dzięki temu w razie włamania do bazy danych nie wyciekną hasła klientów.

```
public static String getSalt()
            throws NoSuchAlgorithmException, NoSuchProviderException {
        SecureRandom sr = SecureRandom.getInstance("SHA1PRNG", "SUN");
        Random random = new Random();
        sr.setSeed(random.nextInt());
        byte[] salt = new byte[16];
        sr.nextBytes(salt);

        return salt.toString();
    }
```

Rysunek 1: Funkcja generowania soli

```
    public static String gethashedpassword(String passwordToHash, String salt) {
        String generatedPassword = null;
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            md.update(salt.getBytes());
            byte[] bytes = md.digest(passwordToHash.getBytes());
            StringBuilder sb = new StringBuilder();

            for (int i = 0; i < bytes.length; i++) {
                sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16)
                        .substring(1));
            }

            generatedPassword = sb.toString();
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        return generatedPassword;
    }
```

Rysunek 2: Funkcja do generowania zahashowanego hasła

8.2 Komunikacja serwer-klient

Komunikacja serwer-klient odbywa się poprzez gniazda(sockets) z wykorzystaniem protokołu TCP/IP.

```
public class PostgreSQLJDBC {
    public static void main(String args[]) {
        int port = 6000;
        Conectionhandler coHandler = new Conectionhandler(port);
        coHandler.start();
    }
}
```

Rysunek 3: Kod funkcji main serwera

```
public class Conectionhandler extends Thread {
    public int SPort;

    public Conectionhandler(int SPort) {
        this.SPort = SPort;

    }
    public void run() {
        try {
            EchoMultiServer server=new EchoMultiServer();
            System.out.println("###################################");
            System.out.println("#***---***!!!!!!!!!!!!!!!***---***#");
            System.out.println("#---***---STARTUJE SERWER---***---#");
            System.out.println("#***---***!!!!!!!!!!!!!!!***---***#");
            System.out.println("###################################");
            server.start(SPort);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Rysunek 4: Kod klasy “Connectionhandler”

Część kodu klasy "EchoMultiServer" odpowiadającego za odbiór wiadomości od klientów, przetwarzanie ich i odesłanie odpowiedzi. Serwer odczytuje wiadomości w formie **[id funkcji do wykonania]***[dane dodatkowe 1]***...***[dane dodatkowe n]** po czym dzieli tę wiadomość na poszczególne fragmenty po czym wykorzystuje je do wybrania zleconej funkcji i jej wykonania.

```
public class EchoMultiServer {
    private ServerSocket serverSocket;

    public void start(int port) throws IOException {
        serverSocket = new ServerSocket(port);
        while (true)
            new EchoClientHandler(serverSocket.accept()).start();
    }

    public void stop() throws IOException {
        serverSocket.close();
    }

public EchoClientHandler(Socket socket) {
            this.clientSocket = socket;
        }

        public void run() {
            try {
                out = new PrintWriter(clientSocket.getOutputStream(), true);

            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                in = new BufferedReader(
                        new InputStreamReader(clientSocket.getInputStream()));

            } catch (IOException e) {
                e.printStackTrace();
            }
            String inputLine;
            try {
                while (true) {
                    inputLine = in.readLine();
                    System.out.println(inputLine);
                    String[] tokenize = inputLine.split("\\*\\*\\*");

int wybor = Integer.parseInt(tokenize[0]);
                    switch (wybor) {
                        case 1: {
                            System.out.println("Proba logowania");
                            int id = Person.logToSystem(tokenize[1], tokenize[2]);
                            int clientversion = Integer.parseInt(tokenize[3]);
                            if (clientversion == 1) {
                                boolean wexist = Worker.checkifexist(String.valueOf(id));

                                if (!wexist) {
                                    LogTable.insert(String.valueOf(id));
                                    String resp = String.valueOf(id);
                                    out.println(resp);
                                } else {
                                    out.println(String.valueOf(-1));
                                }

                            } else if (clientversion == 2) {
                                boolean cexist = Client.checkifexist(String.valueOf(id));

                                if (!cexist) {
                                    String job = Worker.returnJob(String.valueOf(id));
                                    String resp = String.valueOf(id);
                                    LogTable.insert(String.valueOf(id));
                                    resp = resp + "***" + job;
                                    out.println(resp);
                                } else {
                                    out.println("-1***b");
                                }

                            } else {
                                out.println(String.valueOf(-1));
                            }
                        }
                        break;
```

Rysunek 5: Tworzenie nowego serwera akceptującego nowe połączenia, odczytywanie i dekodowanie informacji na przykładzie funkcji logowania.

Kod klasy “GreetClient” odpowiadający za nawiązywanie połączenia z serwerem, przesyłanie wiadomości i odbieranie odpowiedzi a na końcu zamykanie połączenia.

```
public class GreetClient {
    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;

    public void startConnection(String ip, int port) throws IOException {
        clientSocket = new Socket(ip, port);
        out = new PrintWriter(clientSocket.getOutputStream(), true);
        in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
    }

    public String sendMessage(String msg) throws IOException {
        out.println(msg);
        String resp = in.readLine();
        return resp;
    }

    public void stopConnection() throws IOException {
        in.close();
        out.close();
        clientSocket.close();
    }
}
```

Rysunek 6: Klasa GreetClient

8.3 Zabezpieczenia

8.3.1 SQL Injection

W funkcjach w których do wykonania zapytania są używane dane wpisywane przez nieznane osoby (Klientów) do generowania zapytań używamy tak zwanych "prepared statements", dzięki którym możemy uniknąć działań niepożądanych lub szkodliwych.

```
public static int logToSystem(String login, String haslo) {
    int identity = -1;
    Connection c = null;
    Statement stmt = null;
    try {
        Class.forName("org.postgresql.Driver");
        c = DriverManager
                .getConnection("jdbc:postgresql://localhost:5432/r3",
                        "postgres", "bazadanych");
        System.out.println("Opened database successfully");
        String salt = giveSalt(login);
        haslo = SaltedMD5.gethashedpassword(haslo, salt);
        stmt = c.createStatement();

        String selectQuery = "Select o.\"Osoba ID\" from \"Osoba\" o where (o.\"Login\" = ? and o.\"Haslo\"= ?);";
        try (PreparedStatement pstmt =
                     c.prepareStatement(selectQuery);) {
            pstmt.setString(1, login);
            pstmt.setString(2, haslo);
            ResultSet rs = pstmt.executeQuery();
            while (rs.next()) {
                int id = rs.getInt("Osoba ID");
                identity = id;
                System.out.println(id + ". " + identity);
            }
            rs.close();
            pstmt.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        stmt.close();
        c.close();
    } catch (Exception e) {
        System.err.println(e.getClass().getName() + ": " + e.getMessage());
        System.exit(0);
    }
    System.out.println("Select done successfully");
    return identity;
}
```

Rysunek 7: Przykład funkcji logowania używającej "prepared statements"

8.3.2 Zabezpieczenie przed niepoprawnymi znakami i pustymi linijkami danych

W funkcjach w których do tworzenia wiadomości wysyłanych do serwera używane są dane wpisywane przez nieznane osoby (Klientów) przed skonstruowaniem wiadomości czyścimy te linie z niepożądanych znaków (*) i sprawdzamy czy są dłuższe od 0. W przypadku gdy wymagamy od klienta wpisania liczby całkowitej sprawdzamy czy to co aktualnie zostało wpisane do konsoli jest liczbą całkowitą i jeśli nie jest to wyświetlamy komunikat o błędzie i czekamy na wpisanie poprawnych danych.

```
   private static String logToSystem(int port) throws IOException {

        while (true) {
            Scanner console = new Scanner(System.in);
            String msg = String.valueOf(1);
            String login, haslo, podzial;
            podzial = "***";

            System.out.println("Podaj login:");
            login = console.nextLine();
            System.out.println("Podaj haslo:");
            haslo = console.nextLine();

            login = MessageCleaner.removeAsterisks(login);
            haslo = MessageCleaner.removeAsterisks(haslo);

            int hlenght = haslo.length();
            int llenght = login.length();
            if (hlenght < 1 || llenght < 1) {
                System.out.println("Haslo i/lub login nie moga byc puste");
            } else {
                msg = msg + podzial + login + podzial + haslo + podzial + String.valueOf(1);
                String response = msgServer(port, msg);
                int poprawne = Integer.parseInt(response);
                myid = poprawne;
                if (poprawne != -1) {

                    return response;
                }
                System.out.println("Błąd logowania");
            }
        }
    }
```

Rysunek 8: Logowanie klienta

```
public class MessageCleaner {
    public static String removeAsterisks(String message){
        return message.replace("*", "");
    }
}
```

Rysunek 9: Klasa “MessageCleaner” usuwająca niepożądane znaki

```
while (!console.hasNextInt()) {
                console.next();
                System.out.println("Niepoprawny wybór!");
            }
            znak = console.nextInt();
```

Rysunek 10: Zabezpieczenie przed Wpisywaniem do programu znaków nie będących liczbami całkowitymi

8.3.3 Zabezpieczenie przed niepoprawną datą

Klasa "DateValidator" implementuje dwie metody sprawdzające walidację danych, będących datą.

- Metoda "isValid" sprawdza czy podany string można interpretować jako datę o określonym formacie. Metoda jest przydatna w przypadku podania przypadkowego ciągu znaków przez użytkownika. Wykorzystano do tego parsowanie, które po nieudanej operacji zwraca wartość false.

- Metoda "isMonday" sprawdza czy dzień tygodnia podanej daty jest poniedziałkiem. Wykorzystywana jest do tworzenia grafiku przez kierownika. Ponieważ grafik jest tworzony na cały tydzień, to wymagane jest podanie pierwszego dnia tygodnia jakim jest poniedziałek.

Do utworzenia instancji obiektu tej klasy, wymagane jest podanie formatu, jaki data powinna posiadać.

```
public class DateValidator implements IDateValidator {
    private DateTimeFormatter dateFormatter;
    public DateValidator(DateTimeFormatter dateFormatter) {
        this.dateFormatter = dateFormatter;
    }
    @Override
    public boolean isValid(String dateStr) {
        try {
            LocalDate.parse(dateStr, this.dateFormatter);
        } catch (DateTimeParseException e) {
            return false;
        }
        return true;
    }

    @Override
    public boolean isMonday(String dateStr) {
        if(isValid(dateStr)) {
            LocalDate date = LocalDate.parse(dateStr);
            if (date.getDayOfWeek() == DayOfWeek.MONDAY)
                return true;
        }
        return false;
    }
}
```

Rysunek 11: Kod klasy "DateValidator"

Podczas użycia klasy "DateValidator" dodatkowo sprawdzane jest, czy wpisana data już minęła. Do sprawdzenia aktualnej daty wykorzystywany jest czas "Europe/Warsaw". Użytkownik aplikacji jest informowany o wymaganym formacie daty oraz jakiego dnia tygodnia powinien użyć do utworzenia grafiku. Po wpisaniu złej daty, kierownik jest o tym stosownie informowany.

```
  LocalDate today = LocalDate.now( ZoneId.of( "Europe/Warsaw" ) );
    String startDate;
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("uuuu-MM-dd", Locale.US)
            .withResolverStyle(ResolverStyle.STRICT);
    DateValidator dateValidator = new DateValidator(dtf);
    boolean endOfLoop = false;
    do {
        System.out.println("Podaj datę rozpoczynanego tygodnia (Poniedziałek)(RRRR-MM-DD): ");
        startDate = console.next();

        if (dateValidator.isValid(startDate)) {
            LocalDate date2 = LocalDate.parse(startDate, dtf);
            if (date2.isEqual(today) || date2.isAfter(today)){
                if(dateValidator.isMonday(startDate)) {
                    endOfLoop = true;
                } else {
                    System.out.println("Data nie jest poniedziałkiem!");
                }
            } else {
                System.out.println("Ta data już minęła!");
            }
        } else {
            System.out.println("Podano zły format daty!");
        }
```

Rysunek 12: Przykład użycia klasy "DateValidator"

Głównym napotkanym problemem było rozpoznanie przez aplikację lat przestępnych ze wszystkimi regułami. Przykładowo data "2022-02-29" pomimo braku zgodności była akceptowana, a data "2000-02-29" już nie. Rozwiązaniem problemu było użycie metody "withResolverStyle()" podczas tworzenia obiektu "DateTimeFormatter" i przekazania mu argumentu "ResolverStyle.STRICT".

8.3.4 Ograniczanie możliwości wykonania funkcji przez osoby niepożądane

Aplikacja pracownicza przechowuje zmienne określające id osoby i pracę jaką ta osoba wykonuje. W momencie udanego logowania serwer odsyła aplikacji pracowniczej id osoby i stanowisko jaka ta osoba znajduje następnie w zależności od zwróconego stanowiska wyświetla się tej osobie menu z funkcjami dostępnymi tylko pracownikom na danym stanowisku.

```
int id = logToSystem(port);
                    System.out.println("Twoje ID to : " + id);
                    if (myjob.equals("Kierownik")) {
                        menuManager(port, id);
                    } else if (myjob.equals("Szef kuchni")) {
                        menuChef(port, id);
                    } else if (myjob.equals("Kucharz")) {
                        menuCook(port, id);
                    } else if (myjob.equals("Dostawca")) {
                        menuDeliverer(port, id);
                    } else if (myjob.equals("Kelner")) {
                        menuWaiter(port, id);
                    } else {

                    }
```

Rysunek 13: Kod uruchamiający funkcje logowania i wybierający menu z opcjami zależnymi od stanowiska

Dodatkowo w funkcjach zmieniających dane w bazie danych dokonujemy sprawdzenia po stronie serwera czy osoba która wysłała tą komendę ma do tego uprawnienia (czy jego stanowisko się zgadza). Dzieje się tak poprzez wysłaniem wcześniej ustalonego stanowiska razem z komendą do wykonania zadania na serwerze z aplikacji pracowniczej.

```
String iden = tokenize[1];
if (iden.equals("Kucharz")) {
    Cook.mealIsReady(tokenize[2]);
    out.println(String.valueOf(1));

} else {
    out.println(String.valueOf(-1));
}
```

Rysunek 14: Kod sprawdzający czy wiadomość która dotarła do serwera została wysłana przez kucharza

8.3.5 Ograniczanie możliwości przypisania numeru stołu z poza zakresu w bazie danych

Na początku uruchomienia aplikacji stołu pracownik ustawia numer stołu, żeby uniemożliwić wpisania numeru stołu który nie jest w bazie danych zostaje na początku wysłana instrukcja do serwera która zwraca listę numerów stołów która jest następnie porównywana z numerem wpisanym przez pracownika. Jeżeli numer wpisany przez pracownika nie znajduje się na liście to wtedy pracownik musi wpisać numer jeszcze raz.

```
int znak;

        boolean pentla = true;
        Scanner console = new Scanner(System.in);
        ArrayList<Integer> tid = new ArrayList<Integer>();
        String smsg = String.valueOf(30);

        String response1 = msgServer(port, smsg);
        String[] tokenize = response1.split("\\*\\*\\*");
        int licznik = Integer.parseInt(tokenize[0]);
        //pobranie menu i wrzucenie go do arraylista
        for (int i = 0; i < licznik; i++) {
            int id = Integer.parseInt(tokenize[1 + 3 * i]);
            tid.add(id);
        }

        Boolean niepoprawne = true;
        while (niepoprawne) {
            System.out.println("Podaj numer stołu: ");
            while (!console.hasNextInt()) {
                console.next();
                System.out.println("Niepoprawny wybór!");
            }
            myid = console.nextInt();
            for (int i = 0; i < licznik; i++) {
                if (myid == tid.get(i)) {
                    niepoprawne = false;
                    break;
                }
            }
        }
```

Rysunek 15: Kod pobierający listę numerów stołów i sprawdzający czy numer wpisany przez pracownika jest na tej liście

8.3.6 Zapewnienie spójności powiązanych rekordów w bazie danych

Odpowiednie formatowanie kodu tworzącego tabele w bazie danych możemy zapewnić to że nie da się utworzyć rekordów tabeli które posiadają w sobie nieistniejące klucze obce z innych tabel i jednocześnie jeżeli rekordy z tabel gdzie przechowywane są klucze obce na które powołuje się inna tabela zostaną usunięte wtedy automatycznie są usuwane wszystkie rekordy powołujące się na te rekordy z innych tabel.

```
CREATE TABLE "Osoba" (
  "Osoba ID" SERIAL,
  "Imie" VARCHAR(50) not null,
  "Nazwisko" VARCHAR(50) not null,
  "Numer Telefonu" NUMERIC(10),
  "Login" VARCHAR(50) not null unique,
  "Haslo" VARCHAR(100) not null,
  "Salt" VARCHAR(100) not null,
  PRIMARY KEY ("Osoba ID")
);

CREATE TABLE "Dane pracownika" (
  "Dane Pracownika ID" SERIAL,
  "Osoba ID" Integer,
  "Pesel" NUMERIC(11) not null unique,
  "Stanowisko" VARCHAR(50) not null,
  "Data zatrudnienia" DATE,
  PRIMARY KEY ("Dane Pracownika ID"),
  CONSTRAINT "FK_Dane pracownika.Osoba ID"
    FOREIGN KEY ("Osoba ID")
      REFERENCES "Osoba"("Osoba ID")
              on update restrict
        on delete cascade
);
```

Rysunek 16: Przykładowe kody tworzące tabele

Nie da się utworzyć rekordów w tabeli ”Dane pracownika” powołujące się na nieistniejące rekordy z tabeli “Osoba”, jednocześnie usunięcie rekordu z tabeli “Osoba” usunie automatycznie powiązany z nim rekord w tabeli “”Dane pracownika”.

8.4 Utworzenie grafiku przez kierownika

Grafik jest tworzony przez kierownika. Polega on na tworzeniu osobno grafiku dla każdego rodzaju pracownika, nie wliczając szefa kuchni. Algorytm polega na podzieleniu pracowników na dwie części, w których jedna z nich pracuje na zmiany dzienne, a druga na zmiany nocne.

```
String jobName = tokenize[1];
String startDate = tokenize[2];

LocalDate date = LocalDate.parse(startDate);
if (date.getDayOfWeek() != DayOfWeek.MONDAY) {
        out.println(String.valueOf(-1));
}

LocalDate endDate = date.plusDays(4);
List<Integer> listOfWorkerIDs = Schedule.count(jobName, startDate);

for (int i = 0; i < listOfWorkerIDs.size() / 2; i++) {
        Schedule.insert(listOfWorkerIDs.get(i).toString(), startDate, endDate.toString(), "1");
}
for (int i = listOfWorkerIDs.size() / 2; i < listOfWorkerIDs.size(); i++) {
        Schedule.insert(listOfWorkerIDs.get(i).toString(), startDate, endDate.toString(), "0");
}
```

Rysunek 17: Utworzenie grafiku

8.5 Wyświetlanie grafiku jako przykład realizacji wymagania funkcjonalnego

Do wyświetlania grafiku wykorzystywana jest pomocnicza klasa "ScheduleReader", która posiada dwie metody.

- Metoda "readID" jest wykorzystywana do odczytania grafiku przez poszczególnych pracowników. System zwraca z bazy danych tylko te rekordy, które odpowiadają polu "Pracownik ID".
- Metoda "read" odczytuje cały dostępny grafik. Wykorzystuje ją kierownik.

Serwer odczytuje i dzieli wiadomości w formacie:

[ID funkcji do wykonania]*[ID grafiku]***[Numer pracownika]***[Data początkowa]***[Data końcowa]***[Rodzaj zmiany]**

```
public class ScheduleReader {

    public static void readByID(String message) {
        String[] tokenize = message.split("\\*\\*\\*");
        int licznik = Integer.parseInt(tokenize[0]);
        String grafik = "\nTwój grafik:\n";

        if (licznik == 0) {
            System.out.println("Grafik jest pusty\n");
        }
        else {
            for (int i = 0; i < licznik; i++) {
                grafik = grafik + "Od " + tokenize[3 + 5 * i] + " do " + tokenize[4 + 5 * i] + " ";
                if (Objects.equals(tokenize[5 + 5 * i], "t")) {
                    grafik = grafik + "| Zmianny dzienne\n";
                }
                else {
                    grafik = grafik + "| Zmianny nocne\n";
                }

            }
            System.out.println(grafik);
        }
    }

    public static void read(String message) {
        String[] tokenize = message.split("\\*\\*\\*");
        int licznik = Integer.parseInt(tokenize[0]);
        String grafik = "\nCały grafik:\n";

        if (licznik == 0) {
            System.out.println("Grafik jest pusty\n");
        }
        else {
            for (int i = 0; i < licznik; i++) {
                grafik = grafik + "Pracownik o numerze: " + tokenize[2 + 5 * i] + "| Od " +
                    tokenize[3 + 5 * i] + " do " + tokenize[4 + 5 * i] + " ";
                if (Objects.equals(tokenize[5 + 5 * i], "t")) {
                    grafik = grafik + "| Zmianny dzienne\n";
                }
                else {
                    grafik = grafik + "| Zmianny nocne\n";
                }

            }
            System.out.println(grafik);
        }
    }
}
```

Rysunek 18: Klasa ScheduleReader obsługująca wyświetlanie grafiku

```
    public static void showScheduleByID(int port, int id) throws IOException {
        String msg = String.valueOf(26);
        String podzial = "***";
        msg = msg + podzial + String.valueOf(id);

        String response = msgServer(port, msg);
        ScheduleReader.readByID(response);
    }

    public static void showSchedule(int port) throws IOException {
        String msg = String.valueOf(27);
        String podzial = "***";

        String response = msgServer(port, msg);
        ScheduleReader.read(response);
    }
```

Rysunek 19: Wywołanie funkcji wyświetlania grafiku w aplikacji użytkowej

```
 case 26: {
    System.out.println("Wyświetlenie grafiku przez ID");
    int id = Integer.parseInt(tokenize[1]);
    String resp = Schedule.showByID(id);
    out.println(resp);
}
break;

case 27: {
    System.out.println("Wyświetlenie całego grafiku");
    String resp = Schedule.show();
    out.println(resp);
}
break;
```

Rysunek 20: Funkcja wyświetlania grafiku w aplikacji serwerowej

```
public static String show() {
        Connection c = null;
        Statement stmt = null;
        String podzial = "***";
        String ret = "";
        try {
            Class.forName("org.postgresql.Driver");
            c = DriverManager
                    .getConnection("jdbc:postgresql://localhost:5432/r3",
                            "postgres", "bazadanych");
            System.out.println("Opened database successfully");

            stmt = c.createStatement();
            ResultSet rs = stmt.executeQuery("Select * from \"Grafik\";");
            int counter = 0;
            while (rs.next()) {
                counter++;
                String id = rs.getString("Grafik ID");
                String daneid = rs.getString("Dane Pracownika ID");
                String rozp = rs.getString("Data rozpoczecia");
                String zak = rs.getString("Data Zakonczenia");
                String zmiana = rs.getString("Zmiana");

                ret = ret + podzial + id + podzial + daneid + podzial + rozp + podzial + zak + podzial + zmiana;

            }
            ret = String.valueOf(counter) + ret;
            rs.close();
            stmt.close();
            c.close();
        } catch (Exception e) {
            System.err.println(e.getClass().getName() + ": " + e.getMessage());
            System.exit(0);
        }
        System.out.println("Select done successfully");
        return ret;
    }


    public static String showByID(int identity) {
        Connection c = null;
        Statement stmt = null;
        String podzial = "***";
        String ret = "";
        try {
            Class.forName("org.postgresql.Driver");
            c = DriverManager
                    .getConnection("jdbc:postgresql://localhost:5432/r3",
                            "postgres", "bazadanych");
            System.out.println("Opened database successfully");

            stmt = c.createStatement();
            ResultSet rs = stmt.executeQuery("Select * from \"Grafik\" Where \"Dane Pracownika ID\" = '" + identity + "';");
            int counter = 0;
            while (rs.next()) {
                counter++;
                String id = rs.getString("Grafik ID");
                String daneid = rs.getString("Dane Pracownika ID");
                String rozp = rs.getString("Data rozpoczecia");
                String zak = rs.getString("Data Zakonczenia");
                String zmiana = rs.getString("Zmiana");

                ret = ret + podzial + id + podzial + daneid + podzial + rozp + podzial + zak + podzial + zmiana;

            }
            ret = String.valueOf(counter) + ret;
            rs.close();
            stmt.close();
            c.close();
        } catch (Exception e) {
            System.err.println(e.getClass().getName() + ": " + e.getMessage());
            System.exit(0);
        }
        System.out.println("Select done successfully");
        return ret;
    }
```

Rysunek 21: Funkcja tworząca zapytanie do bazy danych w aplikacji serwerowej

9 Testy

Testowanie aplikacji polegało na stworzeniu bazy danych z przykładowymi danymi, uruchomieniu aplikacji serwerowej i każdej wersji z aplikacji użytkowych po czym w aplikacjach użytkowych wykonywano każdą z dostępnych w nich funkcji podając dane poprawne, dane niepoprawne i dane złego typu. Po każdej z akcji obserwowano poprzez program "pgAdmin 4" zmiany w bazie danych dokonanych przez aplikacje użytkowe i to czy informacje wyświetlane przez aplikacje użytkowe pokrywają się z informacjami w bazie danych.

10 Wnioski

Użycie PostgreSQL jako naszej bazy danych było dobrym pomysłem, baza ta zawiera intuicyjny interfejs użytkownika i posiada duża ilość materiałów pomocniczych w internecie co znacząco ułatwiło nam stworzenie tabel i zapełnienie ich danymi testowymi.

Użycie Java DataBase Conectivity (JDBC) do połączenia naszej bazy danych z Miało dobre i złe strony. Pozytywnymi aspektami tej decyzji było to że interfejs ten był stosunkowo łatwy do implementacji i zrozumienia przez wszystkich członków zespołu, o różnych poziomach zaawansowania. Dzięki temu że program ten wykorzystywał zapytania SQL wysyłane do serwera mogliśmy mieć większą kontrolę nad tworzeniem zapytań a także poćwiczyć nasze umiejętności ich pisania. Negatywnym aspektem tej decyzji było to, że tworzenie zapytań SQL dla każdego zastosowania jest dosyć czasochłonnym procesem który jest podatny na błędy ludzkie i który trzeba zabezpieczać przed podaniem do niego niewłaściwych lub szkodliwych danych co w rezultacie spowodowało, że straciliśmy dużo czasu podczas wykonywania etapu 3 projektu.

Projekt ten okazał się bardziej wymagający niż przypuszczaliśmy co spowodowało, że nie udało nam się zaimplementować wszystkich funkcjonalności ustalonych w fazie projektowej do końca trwania kursu.