Dokumentacja projektu
Bank Krwi

Alicja Borek

Semestr zimowy 2025/2026

1 Projekt koncepcji i założenia

1.1 Zdefiniowanie tematu projektu

Celem projektu jest stworzenie kompletnego systemu informatycznego wspierającego działanie banku krwi. Aplikacja ma na celu cyfryzację procesów związanych z oddawaniem krwi, zarządzaniem magazynem oraz dystrybucją krwi do szpitali.

Główne cele i zadania:

- Rejestracja i ewidencja dawców krwi.
- Obsługa procesu zgłaszania chęci oddania krwi (weryfikacja terminów).
- Rejestrowanie faktycznych oddań oraz wyników badań laboratoryjnych.
- Zarządzanie zapotrzebowaniami zgłaszanymi przez jednostki medyczne (szpitale).
- Monitorowanie stanu magazynowego krwi z uwzględnieniem daty ważności.
- Kontrola dostępu do panelów w zależności od roli użytkownika.

1.2 Analiza wymagań użytkownika

W systemie wyróżniono cztery role użytkowników, z których każda posiada dedykowane funkcjonalności:

- **Dawca**: Może rejestrować się w systemie, zarządzać danymi osobowymi, przeglądać historię oddań, wyniki badań oraz zgłaszać chęć oddania krwi (zapisywać się na termin).
- **Pracownik banku krwi**: Posiada uprawnienia do wprowadzania nowych oddań, wpisywania wyników badań (HIV, HCV itp.), zarządzania stanem magazynowym oraz realizowania zapotrzebowań szpitali.
- **Szpital**: Konto instytucjonalne, które umożliwia zgłaszanie zapotrzebowania na konkretną grupę krwi i śledzenie statusu realizacji (oczekujące/zrealizowane).
- **Administrator**: Zarządza użytkownikami systemu (zakładanie kont pracowniczych i szpitalnych oraz dawcó)w, posiada dostęp do pełnych statystyk systemu.

1.3 Zaprojektowanie funkcji

Baza danych realizuje następujące funkcje podstawowe:

- **Przechowywanie danych**: Bezpieczne składowanie haseł (szyfrowanie ‘pgcrypto‘) i danych wrażliwych.
- **Logika biznesowa (Triggers)**: Automatyczne wyliczanie daty ważności krwi (35 dni), blokowanie zbyt częstych oddań (56 dni przerwy), zmiana statusu krwi.
- **Raportowanie (Views)**: Generowanie widoków dla magazynu, historii badań oraz statystyk zapotrzebowań.

2 Projekt diagramów (konceptualny)

2.1 Budowa i analiza diagramu przepływu danych (DFD)

Analiza przepływu danych (Data Flow Diagram - DFD) na poziomie 1 obrazuje logiczne przetwarzanie informacji w systemie Banku Krwi. Model ten identyfikuje źródła danych, procesy transformujące te dane oraz miejsca ich przechowywania.

2.1.1 Elementy sterujące przepływem

Przepływ danych w systemie nie jest liniowy, lecz sterowany przez zdefiniowane reguły biznesowe zaimplementowane w warstwie aplikacji (Flask) oraz bazy danych (Triggery SQL). Główne elementy sterujące to:

- **Autoryzacja ról**: Mechanizm `@login_required` we Flasku decyduje, czy użytkownik ma prawo uruchomić dany proces (np. tylko Szpital może wysłać zapotrzebowanie).
- **Walidacja czasowa**: Wyzwalacze w bazie danych (np. `blokuj_wczesne_zgloszenie`) sterują przepływem, odrzucając próby wprowadzenia błędnych danych.
- **Wyniki badań**: Warunek logiczny w procesie przetwarzania oddania – wynik pozytywny automatycznie zmienia ścieżkę przepływu na "Utylizacja/Blokada", zamiast "Magazyn dostępny".

2.1.2 Specyfikacja przepływów (Wejścia i Wyjścia)

System podzielono na trzy główne procesy przetwarzania danych:

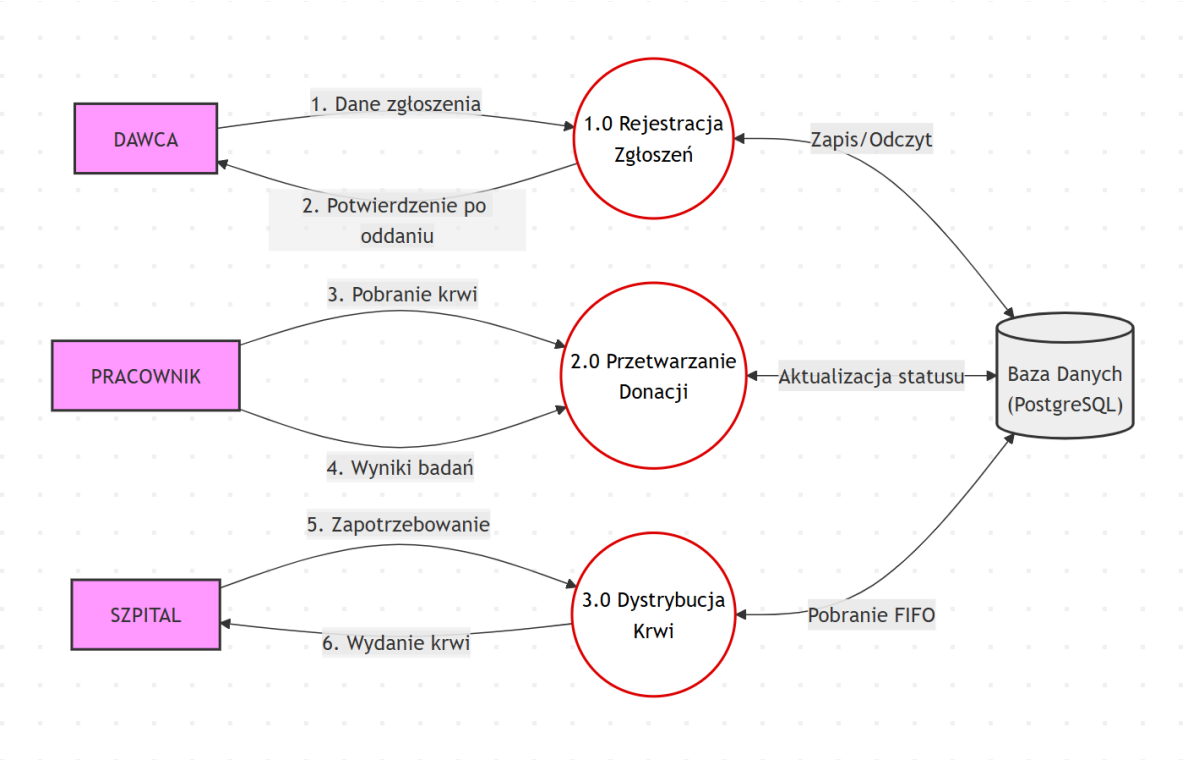
1. **Proces 1.0: Obsługa Dawcy i Zgłoszeń**
   - **Wejście**: Dane osobowe dawcy, deklarowana data wizyty.
   - **Operacje**: Weryfikacja historii w tabeli `Oddania_krwi`, sprawdzenie swoich zgłoszeń, rejestracja w tabeli `Zgloszenia`.
   - **Wyjście**: Zarejestrowanie terminu wizyty ze statusem oczekujące lub komunikat błędu (blokada).
   - **Magazyn danych**: Tabela `Dawcy`, Tabela `Zgloszenia`.

2. **Proces 2.0: Przetwarzanie Donacji i Badań**

   - **Wejście**: Fizyczna jednostka krwi (ilość ml), PESEL dawcy, wyniki badań laboratoryjnych.
   - **Operacje**: Rejestracja oddania przez Pracownika, przypisanie daty ważności (Trigger +35 dni), walidacja wyników badań (Trigger `oznacz_odrzucone_badanie`).
   - **Wyjście**: Jednostka krwi o statusie `'dostepne'` gotowa do wydania lub `'odrzucone'` niezdatna do wydania.
   - **Magazyn danych**: Tabela `Oddania_krwi`, Tabela `Badania`.

3. **Proces 3.0: Dystrybucja i Obsługa Szpitali**

   - **Wejście**: Zapotrzebowanie szpitala (Grupa krwi, Rh, ilość).
   - **Operacje**: Algorytm FIFO (First-In-First-Out) wyszukujący najstarsze pasujące jednostki krwi, aktualizacja stanów magazynowych (zmniejszenie `ilosc_pozostala`), zmiana statusu na `'zrealizowane'`.
   - **Wyjście**: Raport wydania krwi, aktualizacja panelu "Komu pomogłem?" (`Dawca_Szpital`).
   - **Magazyn danych**: Tabela `Zapotrzebowania`, Tabela `Oddanie_Zapotrzebowanie`.



Rysunek 1: Diagram przepływu danych (DFD) poziomu 1 dla systemu Banku Krwi.

2.2 Zdefiniowanie encji i atrybutów

W systemie zdefiniowano następujące klasy encji:

1. **Użytkownicy** (`Uzytkownicy`): Encja nadrzędna przechowująca dane logowania dla wszystkich ról systemowych.
    - Atrybuty: `login`, `haslo` (hash), `rola` (ENUM: ADMIN, PRACOWNIK, SZPITAL, DAWCA), `data_rejestracji`.
2. **Dawcy** (`Dawcy`): Rozszerzenie konta użytkownika o dane medyczne.
    - Atrybuty: `imie`, `nazwisko`, `pesel` (unikalny), `grupa_krwi`, `rh`, `kontakt`, `cel_ml`.
3. **Pracownicy Banku** (`Pracownicy_banku`):
    - Atrybuty: `imie`, `nazwisko`, `stanowisko` (lekarz/laborant).
4. **Szpitale** (`Szpitale`): Placówki zamawiające krew.
    - Atrybuty: `nazwa`, `adres`.
5. **Zgłoszenia** (`Zgloszenia`): Rezerwacje terminów wizyt przez dawców.
    - Atrybuty: `data_zgloszenia`, `status` (oczekujące/zrealizowane).
6. **Oddania Krwi** (`Oddania_krwi`): Fizyczne jednostki krwi w magazynie.
    - Atrybuty: `data_oddania`, `ilosc_ml`, `data_waznosci`, `status` (dostępne, zużyte, przeterminowane, odrzucone$_b$*adanie*), `ilosc_pozostala`.
7. **Badania** (`Badania`): Wyniki testów wirusologicznych dla oddań.
    - Atrybuty: `rodzaj_badania` (HIV, HCV...), `wynik`, `data_badania`.
8. **Zapotrzebowania** (`Zapotrzebowania`): Zamówienia składane przez szpitale.
    - Atrybuty: `grupa_krwi`, `rh`, `ilosc_ml`, `status`, `data_wydania`.

2.3 Zaprojektowanie relacji pomiędzy encjami

2.3.1 Klucze i powiązania (Relacje)

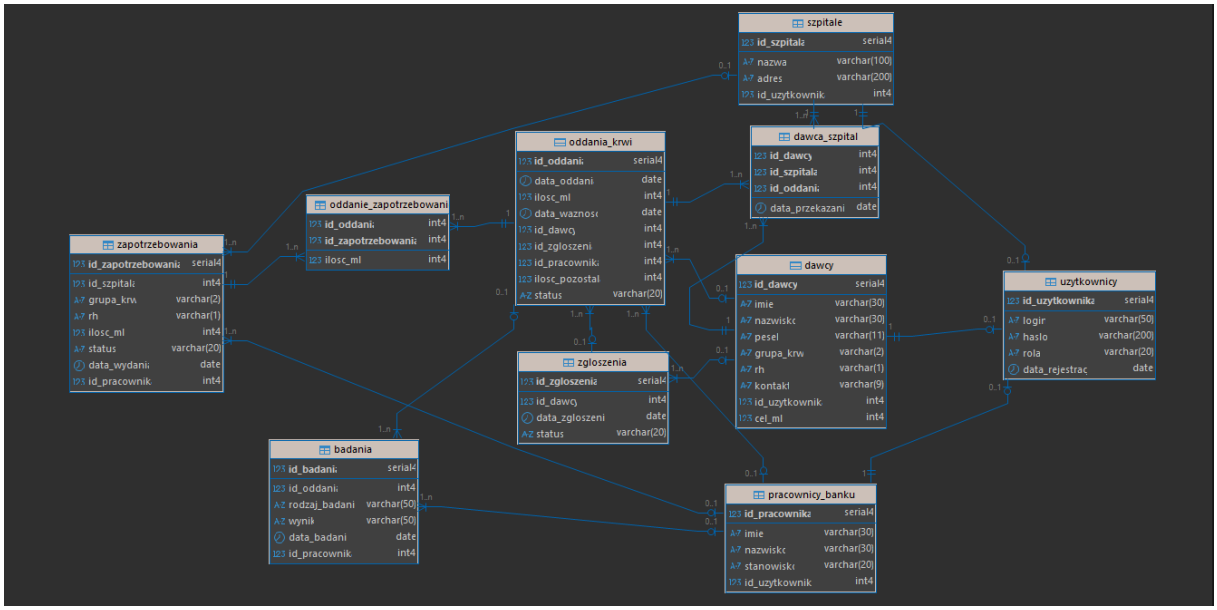
System opiera się na relacjach realizowanych poprzez klucze obce (*Foreign Keys*):

- **Relacja 1:1 (Dziedziczenie ról)**: Tabele `Dawcy`, `Pracownicy_banku` oraz `Szpitale` są powiązane z tabelą `Uzytkownicy` poprzez klucz obcy `id_uzytkownika`, który jest jednocześnie kluczem unikalnym. Pozwala to na oddzielenie danych autoryzacyjnych od danych osobowych.
- **Relacja 1:N (Jeden do wielu)**:
    - Jeden **Dawca** może mieć wiele **Oddań** oraz **Zgłoszeń**.
    - Jedno **Oddanie** może mieć przypisane wiele **Badań**.
    - Jeden **Szpital** może złożyć wiele **Zapotrzebowań**.

2.3.2 Eliminacja powiązań wiele-do-wielu (Tabele asocjacyjne)

Zidentyfikowano obszary, w których zachodziła relacja N:M, i rozwiązano je poprzez wprowadzenie tabel pośredniczących:

- **Realizacja zamówień** (`Oddanie_Zapotrzebowanie`): Pojedyncze zapotrzebowanie (np. 1000 ml) może wymagać krwi z kilku różnych oddań (np. 2 worki po 450 ml + 100 ml z trzeciego). Z kolei jedno oddanie może zostać rozdzielone na kilka mniejszych zapotrzebowań. Tabela ta łączy `id_oddania` z `id_zapotrzebowania`, przechowując informację o konkretnej ilości przekazanej krwi (`ilosc_ml`).
- **Historia przepływu** (`Dawca_Szpital`): Tabela służąca do celów raportowych i śledzenia drogi "Komu pomogłem?". Łączy Dawcę ze Szpitalem poprzez konkretne Oddanie, umożliwiając Dawcy sprawdzenie, do jakiej placówki trafiła jego krew.



Rysunek 2: Diagram związków encji (ERD) obrazujący strukturę bazy danych Banku Krwi.

3 Projekt logiczny

3.1 Projektowanie tabel (Struktura SQL)

Poniżej przedstawiono kod SQL tworzący główne struktury danych. Wykorzystano typy danych dopasowane do domeny (np. 'VARCHAR(11)' dla PESEL, 'CHECK' dla grup krwi).

3.1.1 Tabela Użytkownicy

```
create table Uzytkownicy (
    id_uzytkownika serial primary key,
    login varchar(50) not null unique,
```

```
    haslo varchar(200) not null,
    rola varchar(20) check (rola in ('ADMIN','PRACOWNIK','SZPITAL
       ','DAWCA')),
    data_rejestracji date default current_date
);
```

3.1.2 Tabela Dawcy

```
create table Dawcy (
    id_dawcy serial primary key,
    imie varchar(30) not null,
    nazwisko varchar(30) not null,
    pesel varchar(11) unique not null,
    grupa_krwi varchar(2) check (grupa_krwi in ('A','B','AB','0')
       ),
    rh varchar(1) check (rh in ('+','-')),
    kontakt varchar(9),
    id_uzytkownika int unique references Uzytkownicy(
       id_uzytkownika),
    cel_ml int
);
```

3.1.3 Tabela Pracownicy banku

```
create table Pracownicy_banku (
    id_pracownika serial primary key,
    imie varchar(30) not null,
    nazwisko varchar(30) not null,
    stanowisko varchar(20) check (stanowisko in ('lekarz','
       laborant')),
    id_uzytkownika int unique references Uzytkownicy(
       id_uzytkownika)
);
```

3.1.4 Tabela Szpitale

```
create table Szpitale (
    id_szpitala serial primary key,
    nazwa varchar(100) not null,
    adres varchar(200),
    id_uzytkownika int unique references Uzytkownicy(
       id_uzytkownika)
);
```

3.1.5 Tabela Zgłoszenia

```
create table Zgloszenia (
    id_zgloszenia serial primary key,
    id_dawcy int references Dawcy(id_dawcy),
    data_zgloszenia date default current_date,
    status varchar(20) check (status in ('oczekujace','
       zrealizowane'))
);
```

3.1.6 Tabela Oddania Krwi

```
create table Oddania_krwi (
    id_oddania serial primary key,
    data_oddania date not null,
    ilosc_ml int check (ilosc_ml between 200 and 500),
    data_waznosci date,
    id_dawcy int references Dawcy(id_dawcy),
    id_zgloszenia int references Zgloszenia(id_zgloszenia),
    id_pracownika int references Pracownicy_banku(id_pracownika),
    ilosc_pozostala int default 0,
    status varchar(20) default 'dostepne' check (status in('
       dostepne','zuzyte','przeterminowane','odrzucone_badanie'))
);
```

3.1.7 Tabela Zapotrzebowania

```
create table Zapotrzebowania (
    id_zapotrzebowania serial primary key,
    id_szpitala int references Szpitale(id_szpitala),
    grupa_krwi varchar(2) check (grupa_krwi in ('A','B','AB','0')
       ),
    rh varchar(1) check (rh in ('+','-')),
    ilosc_ml int check (ilosc_ml > 0),
    status varchar(20) check (status in ('oczekujace','
       zrealizowane')),
    data_wydania date,
    id_pracownika int references Pracownicy_banku(id_pracownika)
);
```

3.1.8 Tabela Badania

```
create table Badania (
    id_badania serial primary key,
    id_oddania int references Oddania_krwi(id_oddania),
    rodzaj_badania varchar(50) not null,
    wynik varchar(50) check (wynik in('pozytywny', 'negatywny')),
    data_badania date,
    id_pracownika int references Pracownicy_banku(id_pracownika)
);
```

3.2 Słowniki danych

W bazie zastosowano ograniczenia (‘CHECK constraints‘), które pełnią rolę słowników danych, zapewniając integralność informacji.

Atrybut	**Tabela**	**Dozwolone wartości / Ograniczenia**
rola	Uzytkownicy	’ADMIN’, ’PRACOWNIK’, ’SZPITAL’, ’DAWCA’
grupa_krwi	Dawcy, Zapotrzebowania	’A’, ’B’, ’AB’, ’0’
rh	Dawcy, Zapotrzebowania	’+’, ’-’
stanowisko	Pracownicy_banku	’lekarz’, ’laborant’
status	Zgloszenia	’oczekujace’, ’zrealizowane’
status	Oddania_krwi	’dostepne’, ’zuzyte’, ’przeterminowane’, ’odrzucone_badanie’
ilosc_ml	Oddania_krwi	Przedział 200 - 500
ilosc_ml	Zapotrzebowania, Oddanie_Zapotrzebowanie	Większe od 0

3.3 Normalizacja tabel

Projekt bazy danych został poddany procesowi normalizacji, aby uniknąć redundancji i anomalii.

- **1NF (Pierwsza postać normalna)**: Wszystkie atrybuty są atomowe (np. imię i nazwisko są w osobnych kolumnach, adres szpitala jest pojedynczym ciągiem znaków traktowanym jako całość w tym kontekście).
- **2NF (Druga postać normalna)**: Każda tabela posiada klucz główny, a wszystkie atrybuty niekluczowe zależą od całego klucza. W tabelach asocjacyjnych (np. ‘Oddanie_Zapotrzebowanie‘) atrybuty takie jak ‘ilosc_ml‘ zależą od obu kluczy obcych tworzących klucz główny.
- **3NF (Trzecia postać normalna)**: Usunięto zależności przechodnie. Na przykład dane szczegółowe o dawcy nie są powielane w tabeli ‘Oddania_krwi‘ – znajduje się tam tylko klucz obcy ‘id_dawcy‘. Adres szpitala znajduje się w tabeli ‘Szpitale‘, a nie w ‘Zapotrzebowania‘.

3.4 Denormalizacja

W celu optymalizacji zapytań raportowych wprowadzono elementy denormalizacji w widokach, jednak struktura fizyczna tabel pozostaje znormalizowana. Wyjątkiem optymalizacyjnym jest kolumna ‘ilosc_pozostala‘ w tabeli ‘Oddania_krwi‘, która jest aktualizowana na bieżąco, aby uniknąć każdorazowego sumowania wydań przy sprawdzaniu stanu magazynu.

3.5 Zaprojektowanie operacji na danych - Widoki

Zdefiniowano widoki (Views) w celu uproszczenia warstwy logiki aplikacji i realizacji najczęstsze operacje wyszukiwania:

Stan magazynu:

```
CREATE OR REPLACE VIEW widok_magazyn AS
SELECT
    o.id_oddania,
    d.imie || ' ' || d.nazwisko AS dawca,
    d.grupa_krwi,
    d.rh,
    o.ilosc_ml AS ilosc_poczatkowa,
    o.ilosc_pozostala,
    o.status,
    o.data_waznosci
FROM oddania_krwi o
JOIN dawcy d ON o.id_dawcy = d.id_dawcy;
```

Szczegóły zapotrzebowań:

```
create view Widok_status_zapotrzebowan as
select z.id_zapotrzebowania,
       s.nazwa as szpital,
       z.grupa_krwi,
       z.rh,
       z.ilosc_ml,
       z.status,
       z.data_wydania,
       p.imie || ' ' || p.nazwisko as pracownik
from Zapotrzebowania z
join Szpitale s on z.id_szpitala = s.id_szpitala
left join Pracownicy_banku p on z.id_pracownika = p.id_pracownika
   ;
```

Historia badań:

```
CREATE OR REPLACE VIEW widok_historia_badania AS
SELECT
    b.id_badania,
    o.id_oddania,
    d.imie || ' ' || d.nazwisko AS dawca,
    b.rodzaj_badania,
    b.wynik,
    b.data_badania,
    p.imie || ' ' || p.nazwisko AS pracownik,
    p.id_pracownika
FROM Badania b
JOIN Oddania_krwi o ON b.id_oddania = o.id_oddania
JOIN Dawcy d ON o.id_dawcy = d.id_dawcy
JOIN Pracownicy_banku p ON b.id_pracownika = p.id_pracownika;
```

"Komu pomogłem?":

```
create view Widok_dawcy_szpitale as
```

```
SELECT
    ds.id_dawcy,
    d.imie || ' ' || d.nazwisko AS dawca,
    s.nazwa AS szpital,
    ds.data_przekazania,
    o.ilosc_ml AS ilosc_oddana
FROM dawca_szpital ds
JOIN dawcy d ON ds.id_dawcy = d.id_dawcy
JOIN szpitale s ON ds.id_szpitala = s.id_szpitala
JOIN oddania_krwi o ON ds.id_oddania = o.id_oddania;
```

Aktywność pracownika:

```
create view Widok_pracownicy_aktywnosc as
select p.id_pracownika,
       p.imie || ' ' || p.nazwisko as pracownik,
       p.stanowisko,
       count(distinct o.id_oddania) as liczba_oddan,
       count(distinct b.id_badania) as liczba_badan,
       count(distinct z.id_zapotrzebowania) as
          liczba_zapotrzebowan
from Pracownicy_banku p
left join Oddania_krwi o on p.id_pracownika = o.id_pracownika
left join Badania b on p.id_pracownika = b.id_pracownika
left join Zapotrzebowania z on p.id_pracownika = z.id_pracownika
group by p.id_pracownika, p.imie, p.nazwisko, p.stanowisko;
```

Średnia ilość oddanej krwi:

```
CREATE OR REPLACE VIEW widok_srednia_ilosc AS
SELECT ROUND(AVG(o.ilosc_ml), 2) AS srednia_ml
FROM oddania_krwi o;
```

Daty oddania krwi:

```
create view Widok_dat_oddan as
select id_dawcy,
                min(data_oddania) as pierwsze_oddanie,
       max(data_oddania) as ostatnie_oddanie
from Oddania_krwi
group by 1;
```

Statystyki badań:

```
create view Widok_statystyki_badan as
select
    count(*) filter (where wynik = 'negatywny') as negatywne,
    count(*) filter (where wynik = 'pozytywny') as pozytywne
from Badania;
```

Stan magazynu:

```
CREATE OR REPLACE VIEW widok_stan_krwi AS
SELECT
    d.grupa_krwi,
```

```
    d.rh,
    SUM(o.ilosc_pozostala) AS dostepne_ml
FROM oddania_krwi o
JOIN dawcy d ON o.id_dawcy = d.id_dawcy
WHERE o.status = 'dostepne' AND o.ilosc_pozostala > 0
GROUP BY d.grupa_krwi, d.rh;
```

Suma oddanej krwi danego dawcy:

```
CREATE VIEW widok_suma_ml AS
SELECT
    id_dawcy,
    SUM(ilosc_ml) AS suma_ml
FROM oddania_krwi
GROUP BY id_dawcy;
```

Logistyka oddawanej krwi do konkretnych zapotrzebowań:

```
CREATE OR REPLACE VIEW widok_powiazania AS
SELECT
    oz.id_oddania,
    o.data_oddania,
    o.ilosc_ml AS ilosc_oddana,
    oz.ilosc_ml AS ilosc_przekazana,
    oz.id_zapotrzebowania,
    z.data_wydania AS data_zapotrzebowania,
    s.nazwa AS szpital
FROM oddanie_zapotrzebowanie oz
JOIN oddania_krwi o ON oz.id_oddania = o.id_oddania
JOIN zapotrzebowania z ON oz.id_zapotrzebowania = z.
   id_zapotrzebowania
JOIN szpitale s ON z.id_szpitala = s.id_szpitala
ORDER BY oz.id_zapotrzebowania DESC;
```

Raportowanie dużych zapotrzebowań (Agregacja)

Widok ten wykorzystuje klauzulę `HAVING`, aby filtrować dane już po pogrupowaniu. Służy do identyfikacji szpitali zgłaszających zapotrzebowanie powyżej 1000 ml na daną grupę krwi.

```
create view Widok_zapotrzebowania_duze as
select s.nazwa as szpital,
       z.grupa_krwi,
       z.rh,
       sum(z.ilosc_ml) as suma_ml
from Zapotrzebowania z
join Szpitale s on z.id_szpitala = s.id_szpitala
group by s.nazwa, z.grupa_krwi, z.rh
having sum(z.ilosc_ml) > 1000;
```

3.6 Logika biznesowa - Wyzwalacze (triggery):

Automatycznie wpisuje bieżącą datę wydania przy zmianie statusu zapotrzebowania na 'zrealizowane':

```
create or replace function set_data_wydania()
returns trigger as $$
begin
    if NEW.status = 'zrealizowane' and NEW.data_wydania is null
       then
        NEW.data_wydania := current_date;
    end if;
    return NEW;
end;
$$ language plpgsql;

create trigger trg_set_data_wydania
before insert or update on Zapotrzebowania
for each row
execute function set_data_wydania();
```

Automatyczne ustawienie daty ważności (35 dni):

```
CREATE OR REPLACE FUNCTION ustaw_date_waznosci()
RETURNS trigger AS $$
BEGIN
    -- ZAWSZE ustawiamy datę ważności na 35 dni od daty oddania
    NEW.data_waznosci := NEW.data_oddania + INTERVAL '35 days';
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_oddania_data_waznosci
BEFORE INSERT OR UPDATE ON oddania_krwi
FOR EACH ROW
EXECUTE FUNCTION ustaw_date_waznosci();
```

Zmienia status krwi na 'przeterminowane', jeśli minęła jej data ważności:

```
CREATE OR REPLACE FUNCTION oznacz_przeterminowane()
RETURNS trigger AS $$
BEGIN
    IF NEW.data_waznosci < CURRENT_DATE THEN
        NEW.status := 'przeterminowane';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_przeterminowane
BEFORE UPDATE OR INSERT ON oddania_krwi
FOR EACH ROW
EXECUTE FUNCTION oznacz_przeterminowane();
```

Automatycznie zmienia status oddania na 'odrzucone', jeśli wynik badania jest pozytywny:

```
CREATE OR REPLACE FUNCTION oznacz_odrzucone_badanie()
```

```
RETURNS trigger AS $$
BEGIN
    -- Jeśli wynik badania jest pozytywny -> odrzucamy jednostkę
    IF NEW.wynik = 'pozytywny' THEN
        UPDATE oddania_krwi
        SET status = 'odrzucone_badanie'
        WHERE id_oddania = NEW.id_oddania;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_odrzucone_badanie
AFTER INSERT OR UPDATE ON badania
FOR EACH ROW
EXECUTE FUNCTION oznacz_odrzucone_badanie();
```

Po dodaniu oddania automatycznie zmienia status powiązanego zgłoszenia na 'zrealizowane':

```
CREATE OR REPLACE FUNCTION przyjmij_najstarsze_zgloszenie()
RETURNS trigger AS $$
BEGIN
    -- Aktualizujemy najstarsze oczekujące zgłoszenie danego
       dawcy
    UPDATE zgloszenia
    SET status = 'zrealizowane'
    WHERE id_zgloszenia = (
        SELECT id_zgloszenia
        FROM zgloszenia
        WHERE id_dawcy = NEW.id_dawcy
          AND status = 'oczekujace'
        ORDER BY data_zgloszenia ASC
        LIMIT 1
    );

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_przyjmij_zgloszenie
AFTER INSERT ON oddania_krwi
FOR EACH ROW
EXECUTE FUNCTION przyjmij_najstarsze_zgloszenie();
```

Blokuje możliwość dodania nowego zgłoszenia, jeśli dawca ma już jedno oczekujące:

```
CREATE OR REPLACE FUNCTION blokuj_wiele_zgloszen()
RETURNS trigger AS $$
BEGIN
    IF EXISTS (
```

```
        SELECT 1 FROM zgloszenia
        WHERE id_dawcy = NEW.id_dawcy
          AND status = 'oczekujace'
    ) THEN
        RAISE EXCEPTION 'Dawca ma już aktywne zgłoszenie.';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_1_blokuj_wiele_zgloszen
BEFORE INSERT ON zgloszenia
FOR EACH ROW
EXECUTE FUNCTION blokuj_wiele_zgloszen();
```

Blokada zbyt częstych oddań (56 dni przerwy) Funkcja sprawdza datę ostatniego oddania dla danego dawcy przed dodaniem nowego zgłoszenia.

```
CREATE OR REPLACE FUNCTION blokuj_wczesne_zgloszenie()
RETURNS trigger AS $$
DECLARE
    ostatnie DATE;
BEGIN
    SELECT data_oddania INTO ostatnie
    FROM oddania_krwi
    WHERE id_dawcy = NEW.id_dawcy
    ORDER BY data_oddania DESC
    LIMIT 1;

    IF ostatnie IS NOT NULL AND NEW.data_zgloszenia < ostatnie +
       INTERVAL '56 days' THEN
        RAISE EXCEPTION 'Możesz oddać krew dopiero po %',
           ostatnie + INTERVAL '56 days';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_2_blokuj_wczesne_oddanie
BEFORE INSERT ON zgloszenia
FOR EACH ROW
EXECUTE FUNCTION blokuj_wczesne_zgloszenie();
```

Uniemożliwia rejestrację oddania krwi, jeśli od poprzedniego nie minęło 56 dni:

```
CREATE OR REPLACE FUNCTION blokuj_wczesne_oddanie()
RETURNS trigger AS $$
DECLARE
    ostatnie DATE;
BEGIN
```

```
    SELECT data_oddania INTO ostatnie
    FROM oddania_krwi
    WHERE id_dawcy = NEW.id_dawcy
    ORDER BY data_oddania DESC
    LIMIT 1;

    IF ostatnie IS NOT NULL AND NEW.data_oddania < ostatnie +
       INTERVAL '56 days' THEN
        RAISE EXCEPTION 'Możesz oddać krew dopiero po %',
           ostatnie + INTERVAL '56 days';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_blokuj_wczesne_oddanie
BEFORE INSERT ON oddania_krwi
FOR EACH ROW
EXECUTE FUNCTION blokuj_wczesne_oddanie();
```

Tworzy wpis w historii powiązań dawcy ze szpitalem w momencie przekazania krwi:

```
CREATE OR REPLACE FUNCTION automatyczny_dawca_szpital()
RETURNS trigger AS $$
DECLARE
    v_id_dawcy INT;
    v_id_szpitala INT;
BEGIN
    -- Pobieramy ID dawcy na podstawie oddania
    SELECT id_dawcy INTO v_id_dawcy
    FROM oddania_krwi
    WHERE id_oddania = NEW.id_oddania;

    -- Pobieramy ID szpitala na podstawie zapotrzebowania
    SELECT id_szpitala INTO v_id_szpitala
    FROM zapotrzebowania
    WHERE id_zapotrzebowania = NEW.id_zapotrzebowania;

    -- Wstawiamy rekord do tabeli Dawca_Szpital
    -- (Używamy ON CONFLICT DO NOTHING, żeby nie wywaliło błędu,
       jak już coś tam jest)
    INSERT INTO Dawca_Szpital (id_dawcy, id_szpitala, id_oddania,
        data_przekazania)
    VALUES (v_id_dawcy, v_id_szpitala, NEW.id_oddania,
       CURRENT_DATE)
    ON CONFLICT DO NOTHING;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trg_auto_dawca_szpital
AFTER INSERT ON Oddanie_Zapotrzebowanie
FOR EACH ROW
EXECUTE FUNCTION automatyczny_dawca_szpital();
```

4 Projekt funkcjonalny

4.1 Interfejsy do prezentacji i edycji danych

System został zaprojektowany jako aplikacja webowa. Struktura formularzy obejmuje:

- **Formularz logowania**: Wspólny dla wszystkich ról, dodatkowo dla dawców opcja rejestracji.
- **Panel Dawcy**: Tabela z historią oddań i badań, formularz "Dodaj zgłoszenie".
- **Panel Pracownika**: Formularze "Dodaj oddanie", "Wprowadź wynik badania", tabela zarządzania zapotrzebowaniami oraz magazynem krwi.
- **Panel Szpitala**: Formularz "Zgłoś zapotrzebowanie".

4.2 Wizualizacja danych (Raporty)

Baza danych dostarcza gotowe zestawy danych do raportów poprzez widoki, które wspierają podejmowanie decyzji logistycznych::

- `widok_pracownicy_aktywnosc`: Raport wydajności pracowników (liczba obsłużonych oddań, badań i zapotrzebowań).
- `widok_zapotrzebowania_duze`: Raport dla szpitali zgłaszających duże zapotrzebowanie (> 1000ml). Występuje w formie alertu na panelu pracownika.
- `widok_statystyki_badan`: Zestawienie wyników pozytywnych i negatywnych (np. HIV/HCV).

4.3 Panel sterowania aplikacji

Aplikacja posiada główny Dashboard, który dynamicznie zmienia zawartość w zależności od roli zalogowanego użytkownika (‘id_uzytkownika‘ powiązane z tabelami ról).

4.4 Makropolecenia i automatyzacja procesów

W systemie zaimplementowano szereg makropoleceń w formie funkcji logiki biznesowej (backend Python/Flask), które agregują złożone operacje na bazie danych pod jedną akcją użytkownika.

- **Makropolecenie: Automatyczna Realizacja Zapotrzebowania (Algorytm FIFO)**
  Lokalizacja w kodzie: `/pracownik/zapotrzebowania/zrealizuj/<int:id_zapotrzebowania>`

Działanie: Po wywołaniu funkcji system automatycznie weryfikuje stan magazynowy, pobiera najstarsze dostępne jednostki krwi (zgodnie z zasadą FIFO), dzieli je na wymagane porcje, aktualizuje stany magazynowe (zmniejsza ilość lub oznacza jako „zużyte") oraz zmienia status zapotrzebowania na „zrealizowane" wraz z ustawieniem daty wydania.

- **Makropolecenie: Rejestracja Konta Zintegrowana z Profilem Dawcy**
  Lokalizacja w kodzie: `/register`
  Działanie: System w ramach jednej transakcji atomowej weryfikuje unikalność loginu, tworzy konto użytkownika z zaszyfrowanym hasłem (bcrypt), tworzy powiązany profil dawcy z danymi osobowymi oraz automatycznie loguje użytkownika, inicjując sesję.

- **Makropolecenie: Rejestracja Donacji na podstawie PESEL**
  Lokalizacja w kodzie: `/pracownik/oddania` (metoda POST)
  Działanie: Pracownik wprowadza jedynie numer PESEL i ilość pobranej krwi. Makropolecenie wyszukuje ID dawcy w bazie, waliduje poprawność daty (blokada dat z przyszłości) i rejestruje nowe oddanie, tworząc relację między dawcą, pracownikiem a jednostką krwi.

- **Makropolecenie: Walidacja i Rejestracja Badania**
  Lokalizacja w kodzie: `/pracownik/badania` (metoda POST)
  Działanie: System przyjmuje wyniki badań laboratoryjnych, weryfikuje logiczną poprawność daty i zapisuje wynik w historii. Operacja ta jest ściśle powiązana z triggerami bazy danych, co może skutkować automatyczną dyskwalifikacją jednostki krwi (zmiana statusu na „odrzucone").

5 Dokumentacja

5.1 Wprowadzanie danych

Dane są wprowadzane do systemu na trzy sposoby:

1. **Manualnie**: Poprzez formularze aplikacji webowej (rejestracja, zgłoszenia).

2. **Automatycznie (Triggery)**: System sam uzupełnia pola takie jak ‘data_rejestracji‘ (DEFAULT current_date), ‘data_waznosci‘ czy ‘status‘.

3. **Skrypt inicjalizujący**: Baza została zasilona danymi testowymi przy użyciu instrukcji ‘INSERT‘ zawartych w pliku ‘bank_krwi.sql‘ (m.in. utworzenie kont użytkowników).

5.2 Dokumentacja użytkownika (Instrukcja)

Krótka ścieżka obsługi dla Dawcy:

1. Wejdź na stronę główną i wybierz "Zarejestruj się" lub "Zaloguj się" jeśli masz konto.

2. Po zalogowaniu wybierz zakładkę "Zgłoszenia i historia".

3. Kliknij "Wyślij zgłoszenie" – system sprawdzi, czy upłynął wymagany czas od ostatniego oddania.

4. Po wizycie w punkcie pobrań, historia oddania pojawi się w tej samej zakładce co powyżej.

Krótka ścieżka obsługi dla Pracownika:

1. Zaloguj się na konto pracownicze i sprawdź Panel (system wyświetli alerty o dużych zapotrzebowaniach).

2. Aby przyjąć krew: wejdź w zakładkę "Oddania", wyszukaj dawcę po peselu i dodaj nowe oddanie (system automatycznie ustawi datę ważności).

3. Aby uzupełnić wyniki: przejdź do zakładki "Badania", wybierz oddanie i wprowadź wynik (np. HIV/HCV) – w przypadku wyniku pozytywnego system zablokuje jednostkę.

4. Aby wydać krew: w sekcji "Zapotrzebowania" sprawdź listę oczekujących zamówień i po wydaniu krwi zmień status na "Zrealizowane".

Krótka ścieżka obsługi dla Szpitala:

1. Zaloguj się danymi placówki medycznej.

2. Wybierz opcję "Zgłoś zapotrzebowanie".

3. Wypełnij formularz określając grupę krwi, czynnik Rh oraz ilość (ml).

4. Śledź status swojego zamówienia w tabeli "Moje zapotrzebowania" – gdy pracownik banku wyda krew, status zmieni się z "Oczekujące" na "Zrealizowane".

5.3 Opracowanie dokumentacji technicznej

Projekt zrealizowano w architekturze klient-serwer, uruchamianej w środowisku lokalnym (localhost).

- **Baza danych**: PostgreSQL 14+. Wykorzystano rozszerzenie 'pgcrypto' do bezpiecznego haszowania haseł.
- **Backend**: Python (Flask) łączący się z bazą przez sterownik 'psycopg2'.
- **Struktura plików**:
  - `bank_krwi.sql` – kompletny skrypt inicjalizujący bazę danych. Zawiera definicje DDL (tworzenie tabel, relacji, indeksów), implementację logiki biznesowej po stronie serwera SQL (funkcje PL/pgSQL, wyzwalacze/triggery), definicje widoków raportowych oraz instrukcje DML zasilające system danymi startowym.
  - `db/connection.py` – moduł odpowiedzialny za nawiązywanie połączenia z bazą danych. Wykorzystuje bibliotekę `psycopg2` do komunikacji z PostgreSQL. Centralizuje konfigurację parametrów połączenia (host, port, poświadczenia) oraz automatycznie ustawia ścieżkę wyszukiwania (`search_path`) na schemat `bank_krwi`, co upraszcza zapytania SQL w aplikacji.

– `app.py` – główny plik wykonywalny aplikacji oparty na frameworku Flask. Pełni rolę kontrolera: obsługuje routing (ścieżki URL), zarządza sesjami użytkowników, realizuje mechanizmy autoryzacji (sprawdzanie ról) oraz pośredniczy w wymianie danych między formularzami a bazą danych.

– `templates/` – katalog zawierający szablony widoków HTML. Wykorzystuje silnik szablonów Jinja2 do dynamicznego renderowania danych pobranych z bazy (np. tabele wyników, alerty). Warstwa prezentacji jest zintegrowana z frameworkiem Bootstrap 5, co zapewnia responsywność interfejsu.

– `static/style.css` – arkusz stylów definiujący unikalną szatę graficzną aplikacji. Rozszerza standardowe komponenty frameworka Bootstrap (takie jak tabele i przyciski), nadpisując ich zmienne i wprowadzając dedykowany motyw kolorystyczny oparty na głębokiej czerwieni (`#8a0303`), co zapewnia interfejsowi wygląd adekwatny do tematyki Banku Krwi.

5.4 Wykaz literatury

- Dokumentacja PostgreSQL: `https://www.postgresql.org/docs/`
- Dokumentacja Flask: `https://flask.palletsprojects.com/`
- Materiały z wykładów przedmiotu "Bazy Danych I".
- Dokumentacja Bootstrap 5: `https://getbootstrap.com/`
- Wikipedia: `https://pl.wikipedia.org/wiki/Posta%C4%87_normalna_(bazy_danych)`