Gigster

Dokumentacja projektu

Radosław Firlej

Styczeń 2026

Spis treści

1 Projekt koncepcji, założenia **2**
1.1 Temat projektu, cele i zadania 2
1.2 Analiza wymagań użytkownika 3
1.3 Zaprojektowanie funkcji systemu 4

2 Projekt diagramów **5**
2.1 Diagram przepływu danych (DFD) 5
2.2 Encje i atrybuty 7
2.3 Relacje i diagram ERD 9

3 Projekt logiczny **10**
3.1 Tabele, klucze, indeksy 10
3.2 Słowniki danych 12
3.3 Analiza zależności funkcyjnych i normalizacja 13
3.4 Operacje na danych – kwerendy i widoki 13

4 Projekt funkcjonalny **18**
4.1 Interfejsy do prezentacji, edycji i obsługi danych 18
4.2 Wizualizacja danych 18
4.3 Zdefiniowanie panelu sterowania aplikacji 18
4.4 Makropolecenia 19

5 Dokumentacja **20**
5.1 Wprowadzanie danych 20
5.2 Dokumentacja użytkownika 20
5.3 Dokumentacja techniczna 22
5.4 Wykaz literatury 23

1 Projekt koncepcji, założenia

1.1 Temat projektu, cele i zadania

Tematem projektu jest aplikacja **Gigster**, służąca do zarządzania bazą danych koncertów, artystów i miejsc oraz sprzedaży biletów. System łączy w sobie cechy platformy informacyjnej dla użytkowników oraz panelu administracyjnego dla organizatorów wydarzeń.

Cele projektu:

- Stworzenie spójnej i integralnej bazy danych PostgreSQL przechowującej informacje o wydarzeniach.
- Implementacja funkcjonalnego interfejsu użytkownika (GUI)
- Zapewnienie bezpieczeństwa systemu. Wdrożenie mechanizmów ochrony danych osobowych i transakcyjnych oraz walidacji uprawnień (podział ról: użytkownik/administrator).
- Dostarczenie narzędzi dla administratorów w celu zarządzania danymi i monitorowania statystyk.

Zadania realizowane przez system:

System realizuje kluczowe operacje mające na celu usprawnienie dystrybucji biletów oraz zarządzania wydarzeniami.

- Wyświetlanie listy dostępnych koncertów w formie kart z możliwością podglądu szczegółów wydarzenia.
- Dodawanie, edycja i usuwanie koncertów, artystów, miejsc oraz użytkowników poprzez formularze.
- Wyszukiwanie koncertów według podanych kryteriów
- Realizacja zakupu biletu, od wyboru wydarzenia, poprzez weryfikację dostępności, aż po finalizację zamówienia i przypisanie biletu do konta użytkownika.
- Dynamiczne generowanie raportów sprzedażowych i rankingów popularności.

1.2 Analiza wymagań użytkownika

- Wymagania funkcjonalne dla administratora:
  - Dodawanie, edytowanie informacji oraz usuwanie rekordów dotyczących artystów, miejsc koncertowych i konkretnych wydarzeń.
  - Przesyłanie zdjęć promujących artystów, miejsca i koncerty, z automatycznym czyszczeniem serwera ze starych plików.
  - Możliwość zmiany uprawnień (ról) użytkowników oraz usuwania kont.

  - Monitorowanie sprzedaży biletów, wpływów finansowych oraz popularności wydarzeń poprzez zestawienia tabelaryczne.

- Wymagania funkcjonalne dla użytkownika:
  - Możliwość rejestracji nowego konta oraz logowania się do systemu.
  - Dostęp do aktualnej listy koncertów, wykonawców oraz miejsc, w których odbywają się wydarzenia.
  - Możliwość zawężania listy koncertów według kryteriów takich jak konkretny artysta, gatunek muzyczny, miasto lub nazwa obiektu.
  - Dostęp do rozszerzonych informacji o koncercie, w tym opisu, dokładnego czasu, lokalizacji oraz aktualnej dostępności biletów.
  - Wybór liczby biletów i realizacja transakcji, która skutkuje automatycznym wygenerowaniem biletów w systemie.
  - Możliwość podglądu wszystkich zakupionych biletów wraz z informacją o dacie transakcji i cenie.

- Wymagania systemowe:
  - Realizacja interfejsu w modelu MPA z wykorzystaniem silnika szablonów Jinja2.
  - Logika serwerowa oparta na frameworku Flask (Python). Serwer odpowiada za komunikację z bazą PostgreSQL oraz dynamiczne generowanie kodu HTML.
  - Wykorzystanie relacyjnego systemu PostgreSQL jako głównego repozytorium danych, wspierającego widoki oraz wyzwalacze.
  - Egzekwowanie poprawności informacji na poziomie silnika SQL poprzez stosowanie kluczy głównych, obcych oraz unikatowych indeksów, a także wyzwalaczy.

1.3 Zaprojektowanie funkcji systemu

- Funkcje związane z użytkownikami:
  - rejestracja nowych użytkowników z automatycznym haszowaniem haseł,
  - logowanie oraz zarządzanie sesją użytkownika,

  - przypisanie roli (user, admin), determinujące dostęp do funkcji aplikacji,

- Funkcje związane z koncertami, artystami i miejscami:
  - dodawanie, edycja oraz usuwanie przez administratora,
  - powiązanie koncertu z artystą oraz konkretnym miejscem (salą) w relacyjnym modelu bazy danych,
  - obsługa zdjęć artystów i obiektów oraz plakatów koncertowych

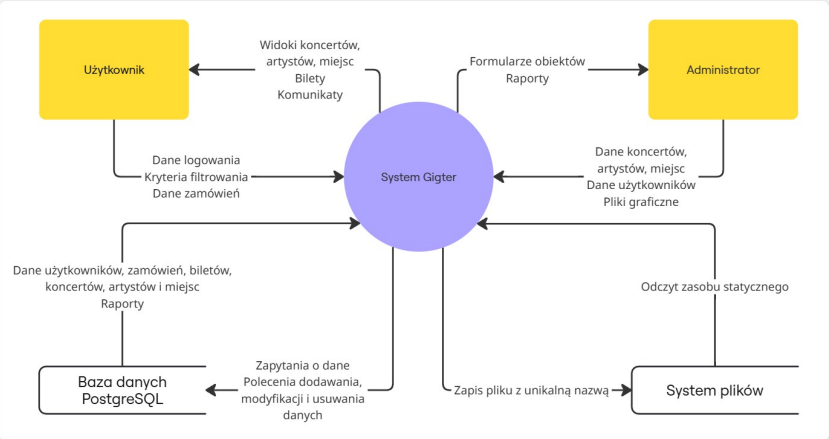
- Funkcje procesu zakupowego i biletów:
  - tworzenie nowych zamówień przypisanych do zalogowanego użytkownika,
  - blokada sprzedaży biletów po wyczerpaniu limitu miejsc w danej sali (wyzwalacz pilnujący pojemności),
  - automatyczne generowanie biletów z unikalnym identyfikatorem po finalizacji transakcji,
  - wyliczanie łącznej kwoty zamówienia na podstawie cen biletów przypisanych do koncertów,

- Funkcje raportowe (widoki):
  - raport obłożenia sal (procentowe wypełnienie miejsc dla każdego koncertu),
  - ranking popularności artystów na podstawie liczby sprzedanych biletów i wygenerowanego przychodu,
  - raport rankingowy miast generujący zestawienie dochodów w ujęciu terytorialnym.
  - miesięczna analiza sprzedaży biletów pozwalająca na śledzenie trendów finansowych,

2 Projekt diagramów

2.1 Diagram przepływu danych (DFD)

Diagram przedstawia architekturę logiczną systemu, w której proces centralny „System Gigster" pełni rolę pośrednika między aktorami zewnętrznymi a magazynami danych.



Rysunek 1: Diagram przepływu danych.

- Określenie wejść i wyjść systemu
  - Wejścia danych:
    - Użytkownik przesyła dane logowania, kryteria filtrowania wydarzeń oraz parametry zamówienia (identyfikator koncertu i liczba biletów).
    - Administrator wprowadza dane strukturalne poprzez formularze obiektów (artyści, miejsca, koncerty), zarządza kontami użytkowników oraz przesyła pliki graficzne promujące wydarzenia.
  - Wyjścia danych:
    - Do użytkownika: System generuje dynamiczne widoki koncertów i artystów, przesyła cyfrowe bilety oraz wyświetla komunikaty statusowe.
    - Do administratora: System udostępnia formularze edycyjne oraz przetworzone raporty analityczne dotyczące sprzedaży i obłożenia sal.
- Operacje i przetwarzanie danych
  - Przekształcanie jawnych haseł w bezpieczne hashe przed ich weryfikacją w bazie danych.
  - Przetwarzanie żądań zakupu biletów, obejmujące walidację liczby sztuk i wywołanie transakcji bazodanowych.
  - Generowanie unikalnych nazw plików i fizyczny zapis obrazów w systemie plików.

  - Pobieranie i formatowanie informacji z widoków SQL w celu prezentacji statystyk w panelu administratora.

- Przechowywanie danych
  - Baza danych PostgreSQL odpowiada za przechowywanie danych relacyjnych. Realizuje zapytania typu INSERT, UPDATE, DELETE i SELECT oraz wykonuje wewnętrzne operacje raportowe.
  - System plików służy do przechowywania zdjęć.
  - Dekoratory tras w app.py decydują, który strumień danych zostanie skierowany do konkretnej funkcji przetwarzającej.
  - Mechanizm sesji steruje dostępem do przepływów administracyjnych, blokując nieuprawnionym użytkownikom możliwość modyfikacji bazy danych.
  - Elementy sterujące na poziomie silnika SQL, które mogą przerwać przepływ danych w przypadku naruszenia reguł.
  - Skrypty po stronie klienta sterują asynchronicznym przepływem danych (np. podgląd zdjęć, dynamiczne obliczanie kwoty zamówienia), co optymalizuje komunikację z serwerem.

2.2 Encje i atrybuty

- **uzytkownicy:**
  - `id` (SERIAL, PK): unikalny identyfikator użytkownika.
  - `nazwa` (VARCHAR): unikalna nazwa użytkownika służąca do autoryzacji w systemie.
  - `haslo` (TEXT): przechowywane w formie hasha hasło.
  - `rola` (VARCHAR): poziom uprawnień w systemie ('user' dla klientów, 'admin' dla zarządców).

- **gatunki:**
  - `id` (SERIAL, PK): unikalny identyfikator gatunku muzycznego.
  - `nazwa` (VARCHAR): nazwa kategorii muzycznej.

- **miejsca:**
  - `id` (SERIAL, PK): unikalny identyfikator obiektu.

  - `nazwa` (VARCHAR): nazwa obiektu.
  - `miasto` (VARCHAR): lokalizacja geograficzna obiektu.
  - `adres` (VARCHAR): dokładne dane adresowe.
  - `pojemnosc` (INTEGER): maksymalna liczba dostępnych miejsc.
  - `zdjecie` (VARCHAR): nazwa zdjęcia obiektu.

- **artysci:**
  - `id` (SERIAL, PK): unikalny identyfikator artysty.
  - `nazwa` (VARCHAR): nazwa wykonawcy.
  - `id_gatunku` (FK): powiązanie z encją `gatunki`.
  - `zdjecie` (VARCHAR): nazwa zdjęcia artysty.

- **koncerty:**
  - `id` (SERIAL, PK): unikalny identyfikator wydarzenia.
  - `id_artysty` (FK): wykonawca występujący na koncercie.
  - `id_miejsca` (FK): lokalizacja, w której odbywa się wydarzenie.
  - `data` (DATE): data koncertu.
  - `czas` (TIME): czas rozpoczęcia koncertu.
  - `opis` (VARCHAR): nazwa koncertu.
  - `zdjecie` (VARCHAR): nazwa zdjęcia koncertu.
  - `cena_biletu` (NUMERIC): cena biletu.

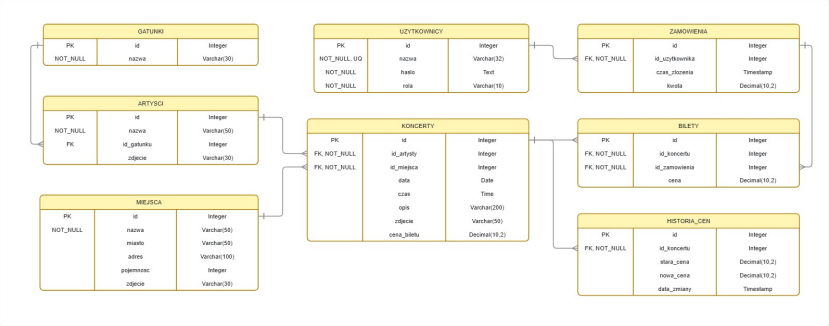
- **zamowienia:**
  - `id` (SERIAL, PK): numer zamówienia.
  - `id_uzytkownika` (FK): kupujący dokonujący transakcji.
  - `czas_zlozenia` (TIMESTAMP): data i godzina finalizacji zakupu.
  - `kwota` (NUMERIC): łączna wartość zamówienia, aktualizowana automatycznie przez trigger.

- **bilety:**
  - `id` (SERIAL, PK): unikalny identyfikator biletu.
  - `id_koncertu` (FK): powiązanie z konkretnym wydarzeniem.
  - `id_zamowienia` (FK): przynależność do konkretnej transakcji.

  - `cena` (NUMERIC): cena zakupu biletu w momencie transakcji.

- **historia_cen:**
  - `id` (SERIAL, PK): numer wpisu .
  - `id_koncertu` (FK): koncert, którego dotyczy zmiana.
  - `stara_cena` (NUMERIC): wartość przed zmianą ceny.
  - `nowa_cena` (NUMERIC): wartość po zmianie ceny.
  - `data_zmiany` (TIMESTAMP): moment dokonania aktualizacji.

2.3 Relacje i diagram ERD



Rysunek 2: Diagram przepływu danych.

- `uzytkownicy` ↔ `koncerty` (N:M): Jeden użytkownik może kupić bilety na wiele różnych koncertów, a na jeden koncert bilety może kupić wielu różnych użytkowników.
- `artysci` ↔ `miejsca` (N:M): Jeden artysta może występować w wielu różnych miejscach, a jedno miejsce może gościć wielu różnych artystów.
- `gatunki` → `artysci` (1:N): Każdy artysta jest przypisany do jednego gatunku muzycznego, natomiast jeden gatunek może reprezentować wielu wykonawców.
- `miejsca` → `koncerty` (1:N): Jeden obiekt koncertowy może gościć wiele wydarzeń w różnych terminach.
- `artysci` → `koncerty` (1:N): Jeden artysta może posiadać zaplanowanych wiele koncertów.

- `koncerty` → `bilety` (1:N): Każdy koncert generuje określoną liczbę biletów.
- `uzytkownicy` → `zamowienia` (1:N): Każdy użytkownik może złożyć wiele zamówień w systemie.
- `zamowienia` → `bilety` (1:N): W ramach jednego zamówienia użytkownik może nabyć wiele biletów na ten sam koncert.
- `koncerty` → `historia_cen` (1:N): Każda zmiana ceny biletu dla danego koncertu tworzy nowy wpis w logu historycznym.

3 Projekt logiczny

3.1 Tabele, klucze, indeksy

- Klucze główne: Wszystkie tabele wykorzystują typ `SERIAL` (automatyczna inkrementacja), co zapewnia unikalność i stałą długość identyfikatorów.
- Klucze obce: Powiązania realizowane są przez typ `INTEGER` z klauzulą `ON DELETE CASCADE`, co gwarantuje automatyczne usuwanie rekordów zależnych.
- Indeksy: System automatycznie tworzy indeksy dla wszystkich kluczy głównych oraz kolumn z ograniczeniem `UNIQUE`, co optymalizuje proces logowania i wyszukiwania.

Skrypt SQL tworzący strukturę bazy

```
-- Gatunki muzyczne
CREATE TABLE gatunki (
    id SERIAL PRIMARY KEY,
    nazwa VARCHAR(30) NOT NULL
);

-- Obiekty koncertowe wraz z ich lokalizacją i limitami miejsc
CREATE TABLE miejsca (
    id SERIAL PRIMARY KEY,
    nazwa VARCHAR(50) NOT NULL,
    miasto VARCHAR(50),
    adres VARCHAR(100),
    pojemnosc INTEGER,
    zdjecie VARCHAR(30)
);

-- Dane kont użytkowników z podziałem na uprawnienia (user/admin)
CREATE TABLE uzytkownicy (
    id SERIAL PRIMARY KEY,
    nazwa VARCHAR(32) NOT NULL UNIQUE,
    haslo TEXT NOT NULL,
    rola VARCHAR(10) NOT NULL DEFAULT 'user' CHECK (rola IN ('user', 'admin'))
);

-- Wykonawcy muzyczni
CREATE TABLE artysci (
    id SERIAL PRIMARY KEY,
    nazwa VARCHAR(50) NOT NULL,
    id_gatunku INTEGER REFERENCES gatunki(id) ON DELETE CASCADE,
    zdjecie VARCHAR(30)
);

-- Wydarzenia koncertowe łączące artystów z miejscami i terminami
CREATE TABLE koncerty (
    id SERIAL PRIMARY KEY,
    id_artysty INTEGER NOT NULL REFERENCES artysci(id) ON DELETE CASCADE,
    id_miejsca INTEGER NOT NULL REFERENCES miejsca(id) ON DELETE CASCADE,
    data DATE,
    czas TIME,
```

```
    opis VARCHAR(200),
    zdjecie VARCHAR(50),
    cena_biletu NUMERIC(10, 2) DEFAULT 100.00 CHECK (cena_biletu >= 0)
);

-- Zamówienia składane przez użytkowników
CREATE TABLE zamowienia (
    id SERIAL PRIMARY KEY,
    id_uzytkownika INTEGER NOT NULL REFERENCES uzytkownicy(id) ON DELETE CASCADE
    czas_zlozenia TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    kwota NUMERIC(10, 2)
);

-- Bilety powiązane z konkretnymi koncertami i zamówieniami
CREATE TABLE bilety (
    id SERIAL PRIMARY KEY,
    id_koncertu INTEGER NOT NULL REFERENCES koncerty(id) ON DELETE CASCADE,
    id_zamowienia INTEGER NOT NULL REFERENCES zamowienia(id) ON DELETE CASCADE,
    cena NUMERIC(10, 2)
);

-- Tabela do przechowywania historii zmian cen biletów
CREATE TABLE historia_cen (
    id SERIAL PRIMARY KEY,
    id_koncertu INTEGER NOT NULL REFERENCES koncerty(id) ON DELETE CASCADE,
    stara_cena NUMERIC(10,2),
    nowa_cena NUMERIC(10,2),
    data_zmiany TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3.2 Słowniki danych

Słownik danych definiuje dopuszczalne dziedziny wartości oraz ograniczenia nałożone na kolumny.

- uzytkownicy.rola: dziedzina = {'user', 'admin'}. Ograniczenie: `CHECK`.
- koncerty.cena_biletu: dziedzina = liczby zmiennoprzecinkowe dodatnie. Ograniczenie: `CHECK >= 0`.
- miejsca.pojemnosc: dziedzina = liczby całkowite dodatnie.

- zamowienia.czas_zlozenia: dziedzina = znacznik czasu (`TIMESTAMP`). Domyślnie: `CURRENT_TIMESTAMP`.

3.3 Analiza zależności funkcyjnych i normalizacja

Projekt bazy danych został poddany procesowi dekompozycji, osiągając Trzecią Postać Normalną (3NF).

1. **1NF**: Wszystkie atrybuty w tabelach są atomowe (np. adres w tabeli `miejsca` jest pojedynczym ciągiem znaków, a każda wejściówka w tabeli `bilety` to osobny rekord).
2. **2NF**: Wszystkie atrybuty niekluczowe są w pełni zależne funkcyjnie od całego klucza głównego.
3. **3NF**: Brak zależności tranzytywnych. Przykład: Atrybut `cena` w tabeli `bilety` jest przechowywany niezależnie od `cena_biletu` w tabeli `koncerty`, aby zachować historyczną wartość w momencie zakupu, co eliminuje zależność od późniejszych zmian w tabeli koncertów.

3.4 Operacje na danych – kwerendy i widoki

```
-- Zestawienie procentowe frekwencji na koncertach na podstawie sprzedanych bilet
CREATE OR REPLACE VIEW w_oblozenie_koncertow AS
SELECT
    k.opis AS koncert,
    a.nazwa AS artysta,
    m.nazwa AS miejsce,
    COUNT(b.id) AS zajete_miejsca,
    m.pojemnosc,
    ROUND((COUNT(b.id)::numeric / m.pojemnosc::numeric) * 100, 2) AS procent
FROM miejsca m
JOIN koncerty k ON m.id = k.id_miejsca
JOIN artysci a ON k.id_artysty = a.id
JOIN bilety b ON k.id = b.id_koncertu
GROUP BY koncert, artysta, miejsce, m.pojemnosc
HAVING COUNT(b.id) > 0
ORDER BY procent DESC;

-- Ranking finansowy i ilościowy popularności poszczególnych wykonawców
CREATE OR REPLACE VIEW w_popularnosc_artystow AS
SELECT
```

```
    a.nazwa AS artysta,
    COUNT(b.id) AS liczba_biletow,
    SUM(b.cena) AS kwota
FROM artysci a
JOIN koncerty k ON a.id = k.id_artysty
JOIN bilety b ON k.id = b.id_koncertu
GROUP BY a.nazwa
ORDER BY liczba_biletow DESC;

-- Chronologiczne podsumowanie przychodów w ujęciu miesięcznym
CREATE OR REPLACE VIEW w_sprzedaz_miesieczna AS
SELECT
    TO_CHAR(z.czas_zlozenia, 'YYYY-MM') AS miesiac,
    COUNT(b.id) AS liczba_biletow,
    SUM(b.cena) AS przychod
FROM zamowienia z
JOIN bilety b ON z.id = b.id_zamowienia
GROUP BY miesiac
ORDER BY miesiac DESC;

-- Analiza geograficzna sprzedaży biletów i średnich cen w miastach
CREATE OR REPLACE VIEW w_ranking_miast AS
SELECT
    m.miasto,
    COUNT(b.id) AS liczba_sprzedanych_biletow,
    SUM(b.cena) AS laczny_przychod,
    ROUND(AVG(b.cena), 2) AS srednia_cena_biletu
FROM miejsca m
JOIN koncerty k ON m.id = k.id_miejsca
JOIN bilety b ON k.id = b.id_koncertu
GROUP BY m.miasto
ORDER BY laczny_przychod DESC;

-- Automatyczna synchronizacja łącznej kwoty zamówienia po dodaniu biletu
CREATE OR REPLACE FUNCTION aktualizuj_kwote_zamowienia()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE zamowienia
    SET kwota = (SELECT SUM(cena) FROM bilety WHERE id_zamowienia = NEW.id_zamow
    WHERE id = NEW.id_zamowienia;
    RETURN NEW;
```

```
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_aktualizuj_kwote
AFTER INSERT OR UPDATE ON bilety
FOR EACH ROW
EXECUTE FUNCTION aktualizuj_kwote_zamowienia();

-- Weryfikacja dostępności wolnych miejsc przed finalizacją sprzedaży biletu
CREATE OR REPLACE FUNCTION sprawdz_limit_miejsc()
RETURNS TRIGGER AS $$
DECLARE
    v_zajete INTEGER;
    v_limit INTEGER;
BEGIN
    SELECT COUNT(*) INTO v_zajete FROM bilety WHERE id_koncertu = NEW.id_koncert

    SELECT m.pojemnosc INTO v_limit
    FROM miejsca m
    JOIN koncerty k ON k.id_miejsca = m.id
    WHERE k.id = NEW.id_koncertu;

    IF v_zajete >= v_limit THEN
        RAISE EXCEPTION 'Brak wolnych miejsc na ten koncert (Limit: %)', v_limit
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_limit_miejsc
BEFORE INSERT ON bilety
FOR EACH ROW
EXECUTE FUNCTION sprawdz_limit_miejsc();

-- Automatyczna archiwizacja zmian cen biletów w tabeli historycznej
CREATE OR REPLACE FUNCTION loguj_zmiane_ceny()
RETURNS TRIGGER AS $$
BEGIN
    IF OLD.cena_biletu <> NEW.cena_biletu THEN
        INSERT INTO historia_cen(id_koncertu, stara_cena, nowa_cena)
```

```
        VALUES (OLD.id, OLD.cena_biletu, NEW.cena_biletu);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_log_ceny
AFTER UPDATE ON koncerty
FOR EACH ROW
EXECUTE FUNCTION loguj_zmiane_ceny();

-- Sprawdzenie konfliktów w obiekcie koncertowym
CREATE OR REPLACE FUNCTION sprawdz_konfikt_miejsca()
RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (
        SELECT 1 FROM koncerty
        WHERE id_miejsca = NEW.id_miejsca
          AND data = NEW.data
          AND id <> COALESCE(NEW.id, -1)
    ) THEN
        RAISE EXCEPTION 'Miejsce jest już zarezerwowane w tym dniu.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_brak_konfliktow
BEFORE INSERT OR UPDATE ON koncerty
FOR EACH ROW
EXECUTE FUNCTION sprawdz_konfikt_miejsca();

-- Walidacja daty koncertu
CREATE OR REPLACE FUNCTION waliduj_date_koncertu()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.data < CURRENT_DATE THEN
        RAISE EXCEPTION 'Koncert nie może odbywać się w przeszłości';
    END IF;
    RETURN NEW;
END;
```

```
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_walidacja_daty
BEFORE INSERT OR UPDATE ON koncerty
FOR EACH ROW
EXECUTE FUNCTION waliduj_date_koncertu();
```

Widoki:

- `w_oblozenie_koncertow`: Oblicza procentowe zestawienie frekwencji na koncertach. Porównuje liczbę rekordów w tabeli `bilety` z atrybutem `pojemnosc` w tabeli `miejsca`.
- `w_popularnosc_artystow`: Realizuje ranking finansowy i ilościowy popularności wykonawców, sumując przychody ze sprzedanych biletów przypisanych do konkretnego artysty.
- `w_sprzedaz_miesieczna`: Dostarcza chronologiczne podsumowanie przychodów w ujęciu miesięcznym.
- `w_ranking_miast`: Wykonuje analizę geograficzną sprzedaży, wyliczając łączny przychód oraz średnią cenę biletu dla każdego miasta zarejestrowanego w bazie.

Wyzwalacze i funkcje proceduralne:

- `trg_aktualizuj_kwote`: Po każdej operacji `INSERT` lub `UPDATE` w tabeli `bilety`, system automatycznie przelicza sumę cen biletów i aktualizuje pole `kwota` w powiązanym rekordzie tabeli `zamowienia`.
- `trg_limit_miejsc`): Wyzwalacz typu `BEFORE INSERT` sprawdza, czy liczba zajętych miejsc nie osiągnęła limitu obiektu. W przypadku braku wolnych miejsc zgłaszany jest wyjątek blokujący transakcję.
- `trg_log_ceny`): Rejestruje każdą zmianę ceny biletu w tabeli `koncerty`. Jeśli nowa cena różni się od starej, rekord z historycznymi wartościami jest dopisywany do tabeli `historia_cen`.
- `trg_brak_konfliktow`): Uniemożliwia dodanie dwóch wydarzeń w tym samym miejscu i o tej samej dacie, dbając o unikalność wykorzystania obiektów koncertowych.
- `trg_walidacja_daty`): Blokuje wprowadzanie koncertów z datą wsteczną względem `CURRENT_DATE`, co zapewnia logiczną spójność harmonogramu.

4 Projekt funkcjonalny

4.1 Interfejsy do prezentacji, edycji i obsługi danych

System *Gigster* wykorzystuje responsywne interfejsy webowe oparte na szablonach Jinja2, które dynamicznie generują treść na podstawie stanów bazy danych. Interfejs został podzielony na moduły prezentacyjne dla klientów oraz moduły edycyjne dla administratorów.

- Formularz autoryzacji (`login.html`, `register.html`): Służy do wprowadzania danych uwierzytelniających. Przepływ danych kończy się weryfikacją hasła (hash) w tabeli `uzytkownicy` i zainicjowaniem sesji użytkownika.
- Interfejs zakupowy (`order.html`): Formularz, który na podstawie wybranego `id_koncertu` prezentuje dane o wydarzeniu i pozwala na określenie liczby biletów. Powiązany jest z funkcją, która w czasie rzeczywistym przelicza kwotę zamówienia przed jego finalizacją.
- Formularze CRUD (`concerts.html`, `artists.html`, `venues.html`): Wykorzystują okna modalne do dodawania i edycji rekordów. Umożliwiają jednoczesną aktualizację danych tekstowych w PostgreSQL oraz przesyłanie plików graficznych do systemu plików.

4.2 Wizualizacja danych

Wizualizacja danych w systemie odbywa się poprzez generowanie dynamicznych raportów, które pobierają dane z uprzednio zdefiniowanych widoków SQL. Pozwala to na separację złożonej logiki obliczeniowej od warstwy prezentacji.

4.3 Zdefiniowanie panelu sterowania aplikacji

- Menu nawigacyjne pomiędzy modułami głównymi:
  - Koncerty: dostęp do pełnej listy nadchodzących wydarzeń wraz z filtrowaniem.
  - Artyści: katalog wykonawców zarejestrowanych w systemie.
  - Miejsca: baza obiektów koncertowych i ich lokalizacji.
- Dostęp do funkcji zakupowych i osobistych:

  - Proces zakupu: dostęp do formularza zamówienia biletów bezpośrednio z widoku koncertu.
  - Moje bilety: sekcja pozwalająca użytkownikowi na podgląd historii zakupów i pobranie biletów.

- Moduł administracyjny:
  - Zarządzanie bazą danych: pełny dostęp do operacji CRUD dla artystów, miejsc i koncertów.
  - Analityka i Raporty: dostęp do generowanych w czasie rzeczywistym raportów obłożenia, sprzedaży miesięcznej oraz popularności artystów.
  - Zarządzanie użytkownikami: możliwość edycji uprawnień (nadawanie roli admina) oraz usuwania kont użytkowników.

- Mechanizmy sterowania i bezpieczeństwa:
  - Nawigacja warunkowa: Wykorzystanie instrukcji `{% if session.get('role') == 'admin' %}` w szablonie bazowym pozwala na ukrywanie lub wyświetlanie zaawansowanych funkcji sterowania w zależności od uprawnień.
  - Zintegrowane wyszukiwanie: Formularz wyszukiwania w panelu sterowania umożliwia szybkie odfiltrowanie danych z tabel bazodanowych bez konieczności przełączania modułów.

4.4 Makropolecenia

- Makro integracji plików (`save_image`): Procedura backendowa, która automatycznie generuje unikalny identyfikator UUID dla pliku, sprawdza jego rozszerzenie, zapisuje go na dysku i aktualizuje ścieżkę w bazie danych jednym wywołaniem.

- Makro transakcyjne (`confirmPurchase`): Skrypt frontendowy, który automatyzuje proces zakupu poprzez zebranie danych z sesji, weryfikację dostępności miejsc (trigger `trg_limit_miejsc`) i wygenerowanie biletów w ramach jednej transakcji atomowej.

- Makro archiwizacyjne (`loguj_zmiane_ceny`): Wyzwalacz bazodanowy działający jako makro w tle, który automatycznie tworzy wpisy w `historia_cen` przy każdej modyfikacji rekordu koncertu, eliminując potrzebę ręcznego logowania zmian przez administratora.

- Automatyzacja formularzy: Skrypt JavaScript, który przy wyborze artysty w formularzu edycji automatycznie ładuje jego obecne zdjęcie i gatunek muzyczny, przyspieszając proces aktualizacji danych.

5 Dokumentacja

5.1 Wprowadzanie danych

- Wprowadzanie manualne: Odbywa się za pośrednictwem formularzy webowych. Administrator wprowadza dane strukturalne dotyczące artystów, miejsc i koncertów. Użytkownicy manualnie wprowadzają dane rejestracyjne oraz określają parametry zamówienia (liczba biletów).
- Wprowadzanie automatyczne:
  - System automatycznie generuje unikalne identyfikatory plików graficznych za pomocą modułu `uuid`, co eliminuje konflikty nazw w systemie plików.
  - Baza danych automatycznie uzupełnia znacznik czasu (`CURRENT_TIMESTAMP`) w momencie składania zamówienia.
  - Wyzwalacze SQL automatycznie obliczają sumaryczne kwoty zamówień oraz archiwizują zmiany cen w tabeli historycznej.
- Zapis referencyjny: Podczas przesyłania zdjęć, system zapisuje fizyczny plik w katalogu `static/images/`, a do bazy danych automatycznie trafia jedynie nazwa pliku.

5.2 Dokumentacja użytkownika

- Logowanie i rejestracja
  1. Ekran startowy: Po uruchomieniu aplikacji użytkownik widzi stronę główną z listą nadchodzących koncertów. Dostęp do funkcji zakupowych wymaga autoryzacji.
  2. Logowanie:
     - Użytkownik podaje nazwę użytkownika oraz hasło.
     - System weryfikuje dane – hasło jest sprawdzane z hashem w tabeli `uzytkownicy`.
     - W przypadku błędnych danych wyświetlany jest komunikat: „Nieprawidłowa nazwa użytkownika lub hasło”.

3. Rejestracja:
   - Użytkownik podaje unikalny login oraz hasło.
   - System automatycznie hashuje hasło przed zapisem do bazy danych PostgreSQL.
   - Nowe konto domyślnie otrzymuje rolę `'user'`.

- Przeglądanie koncertów i artystów

1. Lista wydarzeń (`concerts.html`): Użytkownik może przeglądać wszystkie dostępne koncerty wraz z informacją o artyście, miejscu, dacie oraz cenie biletu.
2. Filtrowanie i wyszukiwanie:
   - Użytkownik może filtrować listę według miasta (tabela `miejsca`), gatunku muzycznego (tabela `gatunki`) lub konkretnego wykonawcy.
3. Katalogi pomocnicze: Użytkownik ma dostęp do osobnych widoków prezentujących szczegółowe profile artystów (`artists.html`) oraz obiektów koncertowych (`venues.html`).

- Proces zakupu biletu

1. Wybór koncertu: Po kliknięciu przycisku „Kup bilet" przy wybranym wydarzeniu, użytkownik zostaje przekierowany do formularza zamówienia (`order.html`).
2. Konfiguracja zamówienia:
   - Użytkownik wybiera liczbę sztuk biletów.
   - System dynamicznie oblicza i wyświetla łączną kwotę do zapłaty (funkcja JavaScript `calculateTotal`).
3. Finalizacja i walidacja:
   - Po kliknięciu „Finalizuj zakup", system uruchamia transakcję bazodanową.
   - Wyzwalacz `trg_limit_miejsc`: Przed zapisem biletu system sprawdza, czy suma sprzedanych biletów nie przekroczyła pojemności obiektu (`pojemnosc` w tabeli `miejsca`).
   - Jeśli limit został osiągnięty, system wyświetla blokadę: „Brak wolnych miejsc na ten koncert".
4. Potwierdzenie: Po poprawnym zakupie bilety są widoczne w module „Moje Bilety" (`tickets.html`).

- Panel sterowania i raportowanie (Administrator)
  Administrator loguje się przez ten sam formularz, lecz po wykryciu roli `'admin'` w sesji, uzyskuje dostęp do panelu sterowania (`dashboard.html`).

  Zarządzanie danymi strukturalnymi:

  – Moduł CRUD: Administrator może dodawać, edytować i usuwać artystów, miejsca oraz koncerty.

  Raporty globalne (Widoki SQL):

  – Frekwencja (`w_oblozenie_koncertow`): Procentowe zestawienie sprzedaży biletów względem pojemności sal.
  – Popularność (`w_popularnosc_artystow`): Ranking artystów generujących największy przychód.
  – Analiza czasowa (`w_sprzedaz_miesieczna`): Podsumowanie przychodów pogrupowane według miesięcy.
  – Zarządzanie użytkownikami: Lista wszystkich kont z możliwością nadawania uprawnień administracyjnych lub usuwania użytkowników.

5.3 Dokumentacja techniczna

- Technologie
  – Backend: Python (Flask)
  – Baza danych: PostgreSQL
  – Frontend: HTML, CSS, JavaScript
  – Silnik szablonów: Jinja2
  – Połączenie z bazą danych: psycopg
- Struktura projektu

```
gigster/
|-- app.py              # Główny plik aplikacji (logika serwera, trasy,
|-- database.sql        # Skrypt bazy PostgreSQL (tabele, widoki, trigge
|-- requirements.txt    # Lista bibliotek i zależności
|-- .gitignore          # Pliki pomijane przez system kontroli wersji Gi
|-- static/             # Zasoby statyczne aplikacji
|   |-- fonts/          # Niestandardowe czcionki interfejsu
|   |-- images/         # Zdjęcia dynamiczne (plakaty koncertów, artyści
```

```
|   |-- script.js       # Logika frontendowa i obsługa AJAX
|   |-- style.css       # Arkusze stylów CSS
|-- templates/          # Szablony HTML (Jinja2)
|   |-- artists.html    # Katalog wykonawców
|   |-- base.html       # Szablon bazowy (navbar, stopka)
|   |-- concerts.html   # Widok główny z listą koncertów
|   |-- dashboard.html  # Panel administracyjny i raporty
|   |-- login.html      # Formularz autoryzacji i rejestracji
|   |-- order.html      # Formularz składania zamówienia
|   |-- tickets.html    # Widok zakupionych biletów użytkownika
|   |-- venues.html     # Katalog obiektów koncertowych
|-- README.md           # Instrukcja wdrożenia i konfiguracji
```

5.4 Wykaz literatury

1. Dokumentacja projektu Flask: *https://flask.palletsprojects.com/*
2. Dokumentacja systemu PostgreSQL: *https://www.postgresql.org/docs/*
3. Dokumentacja biblioteki Psycopg: *https://www.psycopg.org/docs/*