UNIWERSYTET RZESZOWSKI

WYDZIAŁ NAUK ŚCIŚLYCH I TECHNICZNYCH

INSTYTUT INFORMATYKI

Yevhen Marchak & Mykhailo Kleban

134945, 134922

Bazy danych

Dokumentacja projektu System zarządzania magazynem

Praca projektowa

Praca wykonana pod kierunkiem

dr inz. Piotr Grochowalski

Rzeszów 2026

Spis treści

1. Specyfikacja tematu projektu

1.1. Cel projektu

Celem projektu było zaprojektowanie oraz implementacja relacyjnej bazy danych dla systemu zarządzania magazynem. Projekt skupia się na stworzeniu spójnej, logicznej oraz znormalizowanej struktury danych, umożliwiającej efektywne przechowywanie i przetwarzanie informacji związanych z funkcjonowaniem magazynu.

Głównym założeniem projektu było opracowanie bazy danych, która umożliwia zarządzanie stanami magazynowymi, produktami, kategoriami, klientami, pracownikami oraz procesami przyjęcia i wydania towarów. System wspiera realizację podstawowych operacji na danych (CRUD), a także zapytań problemowych pozwalających na analizę danych magazynowych.

Celem jest praktyczne zastosowanie wiedzy z zakresu projektowania relacyjnych baz danych, definiowania relacji pomiędzy encjami, normalizacji danych oraz integracji bazy danych z warstwą aplikacyjną.

1.2. Zakres projektu

Zakres projektu obejmuje zaprojektowanie oraz implementację relacyjnej bazy danych systemu zarządzania magazynem. Baza danych umożliwia przechowywanie informacji niezbędnych do obsługi procesów magazynowych oraz zapewnia integralność i spójność danych.

W ramach projektu zrealizowano:

- definicję struktur tabel wraz z kluczami głównymi i obcymi,
- zaprojektowanie relacji pomiędzy encjami systemu,
- implementację operacji CRUD dla kluczowych tabel,
- realizację zapytań problemowych o charakterze analitycznym,
- opracowanie diagramu związków encji (ERD),
- integrację bazy danych z aplikacją posiadającą interfejs graficzny.

Projekt obejmuje warstwę bazodanową oraz aplikacyjną. Interfejs użytkownika umożliwia wygodną obsługę systemu, natomiast logika biznesowa opiera się na zapytaniach SQL wykonywanych na relacyjnej bazie danych.

1.3. Technologie i narzędzia

Do realizacji projektu wykorzystano następujące technologie oraz narzędzia:

- **PostgreSQL** – relacyjny system zarządzania bazą danych,
- **pgAdmin** – narzędzie administracyjne do zarządzania bazą danych,
- **SQL** – język zapytań do definiowania struktury bazy danych oraz operacji na danych,
- **JetBrains IntelliJ IDEA** – środowisko programistyczne wykorzystane do implementacji aplikacji, logiki systemu oraz interfejsu graficznego użytkownika (GUI),
- **Visual Studio Code** – środowisko używane do tworzenia dokumentacji projektu w systemie LaTeX.

1.4. Instrukcja uruchomienia projektu

Aby poprawnie uruchomić system na lokalnym środowisku, należy postępować zgodnie z poniższymi krokami:

1. **Pobranie kodu źródłowego:**
   - Pobierz projekt z repozytorium GitHub, klikając przycisk *Code*, a następnie wybierając opcję *Download ZIP*.
   - Rozpakuj pobrany folder ProjectBazyDanych-2rok-.
   - Otwórz rozpakowany folder w wybranym środowisku programistycznym (zalecane: JetBrains IntelliJ IDEA).
2. **Przygotowanie relacyjnej bazy danych (PostgreSQL):**
   - Uruchom narzędzie pgAdmin i utwórz nową, pustą bazę danych o nazwie SystemZarzadzania-Magazynem.
   - Kliknij prawym przyciskiem myszy na utworzoną bazę danych i wybierz opcję *Query Tool*. wklej zawartość pliku "BD_file.sql"i wykonad kod (F5).
   - Wklej zawartość pliku SQL zawierającego definicje tabel, relacji oraz ograniczeń integralności i wykonaj kod (klawisz *F5*).
3. **Konfiguracja połączenia z bazą danych:**
   - Otwórz projekt aplikacji w środowisku JetBrains IntelliJ IDEA.
   - Skonfiguruj parametry połączenia z bazą danych, takie jak: nazwa bazy danych, użytkownik oraz hasło.
   - Upewnij się, że aplikacja posiada poprawny dostęp do serwera PostgreSQL.
4. **Uruchomienie aplikacji:**
   - Upewnij się, że serwer bazy danych PostgreSQL jest uruchomiony.
   - Uruchom aplikację z poziomu środowiska JetBrains IntelliJ IDEA.
   - Po uruchomieniu aplikacji możliwa jest interakcja z systemem poprzez interfejs graficzny użytkownika (GUI).

Po wykonaniu powyższych kroków system zarządzania magazynem jest gotowy do pracy i umożliwia realizację operacji na danych magazynowych.

2. Aspekt projektowy baz danych

2.1. Zdefiniowane struktury tabel

W niniejszym podrozdziale przedstawiono szczegółowy opis tabel wchodzących w skład relacyjnej bazy danych systemu zarządzania magazynem. Każda tabela została zaprojektowana w sposób zapewniający spójność danych oraz poprawne odwzorowanie procesów magazynowych.

2.1.1. Tabela `categories`

Tabela `categories` przechowuje informacje o kategoriach produktów dostępnych w magazynie.

- `category_id` (`BIGSERIAL`, PK) — unikalny identyfikator kategorii,
- `name` (`text`, NOT NULL) — nazwa kategorii,
- `description` (`text`) — opis kategorii.

2.1.2. Tabela `suppliers`

Tabela `suppliers` zawiera dane dotyczące dostawców towarów.

- `supplier_id` (`BIGSERIAL`, PK) — identyfikator dostawcy,
- `company_name` (`text`, NOT NULL) — nazwa firmy dostawcy,
- `address` (`text`) — adres dostawcy,
- `phone` (`text`) — numer telefonu,
- `email` (`text`) — adres e-mail,
- `tax_id` (`text`) — numer podatkowy.

2.1.3. Tabela `clients`

Tabela `clients` przechowuje dane klientów systemu magazynowego.

- `client_id` (`BIGSERIAL`, PK) — identyfikator klienta,
- `company_name` (`text`, NOT NULL) — nazwa firmy klienta,
- `delivery_address` (`text`) — adres dostawy,
- `phone` (`text`) — numer telefonu,
- `email` (`text`) — adres e-mail,
- `tax_id` (`text`) — numer podatkowy.

2.1.4. Tabela `employees`

Tabela `employees` zawiera informacje o pracownikach magazynu.

- `employee_id` (`BIGSERIAL`, PK) — identyfikator pracownika,
- `first_name` (`text`, NOT NULL) — imię,
- `last_name` (`text`, NOT NULL) — nazwisko,
- `position` (`varchar(100)`, NOT NULL) — stanowisko,
- `hire_date` (`date`) — data zatrudnienia,
- `phone` (`text`) — numer telefonu,
- `email` (`text`) — adres e-mail.

2.1.5. Tabela `locations`

Tabela `locations` definiuje lokalizacje magazynowe.

- `location_id` (`BIGSERIAL`, PK) — identyfikator lokalizacji,
- `location_code` (`text`, NOT NULL) — kod lokalizacji,
- `location_type` (`varchar(50)`, NOT NULL) — typ lokalizacji,
- `max_capacity` (`numeric`) — maksymalna pojemność lokalizacji.

2.1.6. Tabela `shipments`

Tabela `shipments` rejestruje wydania towarów z magazynu do klientów. Zawiera informacje dotyczące realizowanych wysyłek oraz osób odpowiedzialnych za ich obsługę.

- `shipment_id` (`BIGSERIAL`, PK) — identyfikator wydania,
- `client_id` (`bigint`, NOT NULL, FK → `clients.client_id`) — klient,
- `employee_id` (`bigint`, FK → `employees.employee_id`) — pracownik realizujący wydanie,
- `shipment_date` (`date`, NOT NULL) — data wydania,
- `status` (`varchar(50)`, NOT NULL) — status wydania (np. „oczekujące”, „zrealizowane”).

2.1.7. Tabela `shipment_details`

Tabela `shipment_details` przechowuje szczegółowe informacje dotyczące produktów wydawanych w ramach konkretnego wydania magazynowego. Stanowi tabelę pośrednią realizującą relację wiele do wielu pomiędzy wydaniami a produktami.

- `shipment_id` (`bigint`, NOT NULL, FK → `shipments.shipment_id`),
- `product_id` (`bigint`, NOT NULL, FK → `products.product_id`),
- `quantity` (`numeric`, NOT NULL) — ilość wydanego produktu.

2.1.8. Tabela `products`

Tabela `products` przechowuje informacje o produktach.

- `product_id` (`BIGSERIAL`, PK) — identyfikator produktu,
- `sku` (`text`, NOT NULL) — kod SKU produktu,
- `name` (`text`, NOT NULL) — nazwa produktu,
- `description` (`text`) — opis produktu,
- `category_id` (`bigint`, FK → `categories.category_id`) — kategoria produktu,
- `supplier_id` (`bigint`, FK → `suppliers.supplier_id`) — dostawca produktu,
- `weight` (`numeric`) — waga produktu,
- `dimensions` (`text`) — wymiary produktu.

2.1.9. Tabela `inventory`

Tabela `inventory` przechowuje informacje o stanach magazynowych.

- `inventory_id` (`BIGSERIAL`, PK) — identyfikator rekordu magazynowego,
- `product_id` (`bigint`, FK → `products.product_id`) — produkt,
- `location_id` (`bigint`, FK → `locations.location_id`) — lokalizacja magazynowa,
- `quantity` (`numeric`, NOT NULL) — ilość produktu,
- `last_updated` (`timestamptz`) — data ostatniej aktualizacji.

2.1.10. Tabela `receipts`

Tabela `receipts` rejestruje przyjęcia towarów do magazynu.

- `receipt_id` (`BIGSERIAL`, PK) — identyfikator przyjęcia,
- `supplier_id` (`bigint`, FK → `suppliers.supplier_id`) — dostawca,
- `employee_id` (`bigint`, FK → `employees.employee_id`) — pracownik przyjmujący,
- `receipt_date` (`date`, NOT NULL) — data przyjęcia,
- `external_invoice_no` (`text`) — numer faktury zewnętrznej,
- `status` (`varchar(50)`, NOT NULL) — status przyjęcia.

2.1.11. Tabela `receipt_details`

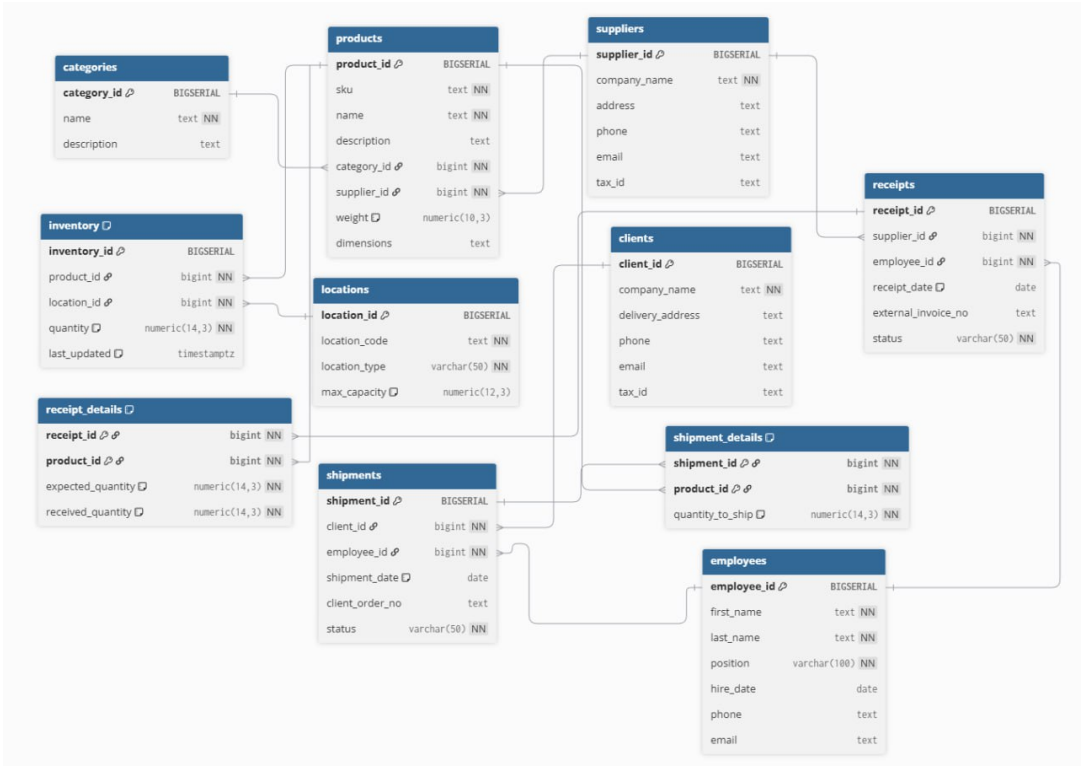
Tabela `receipt_details` zawiera szczegóły przyjęć magazynowych.

- `receipt_id` (`bigint`, FK → `receipts.receipt_id`),
- `product_id` (`bigint`, FK → `products.product_id`),
- `expected_quantity` (`numeric`, NOT NULL) — ilość oczekiwana,
- `received_quantity` (`numeric`) — ilość faktycznie otrzymana.

2.2. Diagram związków encji

Struktura relacyjnej bazy danych systemu zarządzania magazynem została przedstawiona w postaci diagramu związków encji (ERD). Diagram obrazuje encje występujące w systemie oraz relacje pomiędzy nimi, w tym klucze główne oraz obce.

Diagram ERD stanowi graficzne odwzorowanie struktury bazy danych i ułatwia analizę zależności pomiędzy poszczególnymi tabelami.



Rys. 2.1. Diagram związków encji (ERD) systemu zarządzania magazynem

2.3. Powiązania pomiędzy tabelami

Relacje pomiędzy tabelami w systemie zarządzania magazynem zostały zaprojektowane w sposób odzwierciedlający rzeczywiste zależności występujące w procesach magazynowych. Zastosowanie kluczy obcych zapewnia integralność referencyjną danych oraz eliminuje możliwość występowania niespójnych rekordów.

Poniżej przedstawiono najważniejsze relacje pomiędzy encjami systemu:

- Relacja typu jeden do wielu (1:N) pomiędzy tabelą `categories` a tabelą `products`. Jedna kategoria może być przypisana do wielu produktów, natomiast każdy produkt należy do jednej kategorii.
- Relacja typu jeden do wielu (1:N) pomiędzy tabelą `suppliers` a tabelą `products`. Jeden dostawca może dostarczać wiele produktów, natomiast każdy produkt posiada jednego dostawcę.
- Relacja typu jeden do wielu (1:N) pomiędzy tabelą `locations` a tabelą `inventory`. Jedna lokalizacja magazynowa może zawierać wiele rekordów stanów magazynowych.
- Relacja typu jeden do wielu (1:N) pomiędzy tabelą `products` a tabelą `inventory`. Każdy produkt może występować w wielu lokalizacjach magazynowych.

- Relacja typu jeden do wielu (1:N) pomiędzy tabelą `suppliers` a tabelą `receipts`. Jeden dostawca może realizować wiele przyjęć magazynowych.
- Relacja typu jeden do wielu (1:N) pomiędzy tabelą `employees` a tabelą `receipts`. Jeden pracownik może obsługiwać wiele przyjęć towarów.
- Relacja typu wiele do wielu (N:M) pomiędzy tabelą `receipts` a tabelą `products`, zrealizowana za pomocą tabeli pośredniej `receipt_details`. Tabela ta przechowuje informacje o ilościach oczekiwanych oraz faktycznie otrzymanych produktów.
- Relacja typu jeden do wielu (1:N) pomiędzy tabelą `clients` a tabelą `shipments`. Jeden klient może posiadać wiele wydań magazynowych.
- Relacja typu jeden do wielu (1:N) pomiędzy tabelą `employees` a tabelą `shipments`. Jeden pracownik może realizować wiele wydań towarów.
- Relacja typu wiele do wielu (N:M) pomiędzy tabelą `shipments` a tabelą `products`, zrealizowana za pomocą tabeli pośredniej `shipment_details`. Tabela ta przechowuje informacje o ilości wydanych produktów.

Zaprojektowane relacje umożliwiają poprawne odwzorowanie procesów przyjęcia oraz wydania towarów, a także zapewniają spójność i integralność danych w całym systemie.

2.4. Kod realizujący bazę danych

W niniejszym podrozdziale przedstawiono kompletną definicję struktur tabel relacyjnej bazy danych systemu zarządzania magazynem. Zaprezentowane listingi zawierają instrukcje `CREATE TABLE` odpowiedzialne za utworzenie wszystkich encji systemu wraz z kluczami głównymi, kluczami obcymi oraz ograniczeniami integralności.

Listing 2.1. Definicja tabeli categories

```
CREATE TABLE categories (
    category_id BIGSERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    description TEXT
);
```

Listing 2.2. Definicja tabeli suppliers

```
CREATE TABLE suppliers (
    supplier_id BIGSERIAL PRIMARY KEY,
    company_name TEXT NOT NULL,
    address TEXT,
    phone TEXT,
    email TEXT,
    tax_id TEXT
);
```

Listing 2.3. Definicja tabeli clients

```
CREATE TABLE clients (
    client_id BIGSERIAL PRIMARY KEY,
    company_name TEXT NOT NULL,
    delivery_address TEXT,
```

```
    phone TEXT,
    email TEXT,
    tax_id TEXT
);
```

Listing 2.4. Definicja tabeli employees

```
CREATE TABLE employees (
    employee_id BIGSERIAL PRIMARY KEY,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    position VARCHAR(100) NOT NULL,
    hire_date DATE,
    phone TEXT,
    email TEXT
);
```

Listing 2.5. Definicja tabeli locations

```
CREATE TABLE locations (
    location_id BIGSERIAL PRIMARY KEY,
    location_code TEXT NOT NULL UNIQUE,
    location_type VARCHAR(50) NOT NULL,
    max_capacity NUMERIC CHECK (max_capacity >= 0)
);
```

Listing 2.6. Definicja tabeli products

```
CREATE TABLE products (
    product_id BIGSERIAL PRIMARY KEY,
    sku TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    description TEXT,
    category_id BIGINT NOT NULL,
    supplier_id BIGINT,
    weight NUMERIC CHECK (weight >= 0),
    dimensions TEXT,
    FOREIGN KEY (category_id) REFERENCES categories(category_id),
    FOREIGN KEY (supplier_id) REFERENCES suppliers(supplier_id)
);
```

Listing 2.7. Definicja tabeli inventory

```
CREATE TABLE inventory (
    inventory_id BIGSERIAL PRIMARY KEY,
    product_id BIGINT NOT NULL,
    location_id BIGINT NOT NULL,
    quantity NUMERIC NOT NULL CHECK (quantity >= 0),
    last_updated TIMESTAMPTZ DEFAULT now(),
    UNIQUE (product_id, location_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id),
    FOREIGN KEY (location_id) REFERENCES locations(location_id)
);
```

Listing 2.8. Definicja tabeli receipts

```
CREATE TABLE receipts (
    receipt_id BIGSERIAL PRIMARY KEY,
```

```
    supplier_id BIGINT NOT NULL,
    employee_id BIGINT,
    receipt_date DATE DEFAULT CURRENT_DATE,
    external_invoice_no TEXT,
    status VARCHAR(50) NOT NULL,
    FOREIGN KEY (supplier_id) REFERENCES suppliers(supplier_id),
    FOREIGN KEY (employee_id) REFERENCES employees(employee_id)
);
```

Listing 2.9. Definicja tabeli receipt_details

```
CREATE TABLE receipt_details (
    receipt_id BIGINT NOT NULL,
    product_id BIGINT NOT NULL,
    expected_quantity NUMERIC NOT NULL CHECK (expected_quantity >= 0),
    received_quantity NUMERIC NOT NULL CHECK (received_quantity >= 0),
    PRIMARY KEY (receipt_id, product_id),
    FOREIGN KEY (receipt_id) REFERENCES receipts(receipt_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

Listing 2.10. Definicja tabeli shipments

```
CREATE TABLE shipments (
    shipment_id BIGSERIAL PRIMARY KEY,
    client_id BIGINT NOT NULL,
    employee_id BIGINT,
    shipment_date DATE DEFAULT CURRENT_DATE,
    client_order_no TEXT,
    status VARCHAR(50) NOT NULL,
    FOREIGN KEY (client_id) REFERENCES clients(client_id),
    FOREIGN KEY (employee_id) REFERENCES employees(employee_id)
);
```

Listing 2.11. Definicja tabeli shipment_details

```
CREATE TABLE shipment_details (
    shipment_id BIGINT NOT NULL,
    product_id BIGINT NOT NULL,
    quantity_to_ship NUMERIC NOT NULL CHECK (quantity_to_ship >= 0),
    PRIMARY KEY (shipment_id, product_id),
    FOREIGN KEY (shipment_id) REFERENCES shipments(shipment_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

2.5. Normalizacja bazy danych

Proces normalizacji bazy danych został przeprowadzony w celu eliminacji redundancji danych, zapewnienia spójności informacji oraz poprawy integralności logicznej systemu. Projektowana baza danych systemu zarządzania magazynem spełnia założenia trzech pierwszych postaci normalnych: 1NF, 2NF oraz 3NF.

2.5.1. Pierwsza postać normalna (1NF)

Pierwsza postać normalna wymaga, aby:

- wszystkie atrybuty zawierały wartości atomowe,
- w tabelach nie występowały powtarzające się grupy danych,
- każdy rekord był jednoznacznie identyfikowany przez klucz główny.

W zaprojektowanej bazie danych wszystkie tabele posiadają klucze główne (np. `category_id`, `product_id`, `shipment_id`), a każdy atrybut przechowuje pojedynczą, niepodzielną wartość. Nie występują pola wielowartościowe ani listy danych w jednym atrybucie.

2.5.2. Druga postać normalna (2NF)

Druga postać normalna wymaga spełnienia warunków 1NF oraz braku zależności częściowych atrybutów niekluczowych od części klucza głównego.

W bazie danych zależności częściowe zostały wyeliminowane poprzez:

- rozdzielenie danych opisowych do osobnych tabel (np. `categories`, `suppliers`),
- zastosowanie tabel pośrednich z kluczami złożonymi, takich jak `receipt_details` oraz `shipment_details`.

W tabelach posiadających klucz złożony wszystkie atrybuty niekluczowe zależą od całego klucza głównego, a nie od jego fragmentu.

2.5.3. Trzecia postać normalna (3NF)

Trzecia postać normalna wymaga, aby w tabelach nie występowały zależności przechodnie pomiędzy atrybutami niekluczowymi.

W zaprojektowanej bazie danych:

- dane klientów, pracowników, dostawców i produktów zostały umieszczone w oddzielnych tabelach,
- informacje zależne od innych encji są przechowywane wyłącznie poprzez klucze obce,
- nie występują atrybuty, które mogłyby być wyznaczane na podstawie innych atrybutów niekluczowych w tej samej tabeli.

Dzięki temu baza danych spełnia wymagania trzeciej postaci normalnej, zapewniając wysoką spójność danych oraz łatwość dalszej rozbudowy systemu.

2.5.4. Podsumowanie

Zaprojektowana struktura bazy danych systemu zarządzania magazynem została poprawnie znormalizowana do trzeciej postaci normalnej (3NF). Zastosowana normalizacja minimalizuje redundancję danych, ułatwia utrzymanie spójności informacji oraz poprawia wydajność operacji na bazie danych.

3. Zapytania problemowe i ich realizacja

W niniejszym rozdziale przedstawiono zapytania problemowe, które odzwierciedlają rzeczywiste potrzeby biznesowe systemu zarządzania magazynem. Dla każdego problemu zaprezentowano opis zagadnienia, kolejne kroki rozwiązania oraz implementację w postaci funkcji lub procedury składowanej w języku PL/pgSQL.

3.1. Premia dla pracowników za obsłużone wysyłki

- wyszukanie pracowników realizujących wysyłki w danym miesiącu,
- zliczenie liczby wysyłek dla każdego pracownika,
- wyznaczenie procentu premii na podstawie liczby wysyłek.

Listing 3.1. Funkcja obliczająca premię pracownika

```
CREATE OR REPLACE FUNCTION public.calculate_employee_bonus(
    p_month INT,
    p_year INT
)
RETURNS TABLE (
    employee_id BIGINT,
    shipments_count INT,
    bonus_percent INT
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        s.employee_id,
        COUNT(*)::INT AS shipments_count,
        CASE
            WHEN COUNT(*) > 100 THEN 30
            WHEN COUNT(*) BETWEEN 50 AND 100 THEN 15
            ELSE 0
        END AS bonus_percent
    FROM public.shipments s
    WHERE EXTRACT(MONTH FROM s.shipment_date) = p_month
      AND EXTRACT(YEAR  FROM s.shipment_date) = p_year
      AND s.employee_id IS NOT NULL
    GROUP BY s.employee_id;
END;
$$;
```

3.2. Automatyczne oznaczanie przeterminowanych wysyłek

- wyszukanie wysyłek starszych niż 7 dni,
- sprawdzenie ich aktualnego statusu,

- aktualizacja statusu na `OVERDUE`.

Listing 3.2. Procedura oznaczania przeterminowanych wysyłek

```
CREATE OR REPLACE PROCEDURE public.mark_overdue_shipments()
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE public.shipments
    SET status = 'OVERDUE'
    WHERE shipment_date < CURRENT_DATE - INTERVAL '7 days'
      AND status <> 'DELIVERED';
END;
$$;
```

3.3. Wykrywanie braków magazynowych

- pobranie aktualnych stanów magazynowych,
- porównanie ilości z progiem minimalnym,
- zwrócenie produktów wymagających uzupełnienia.

Listing 3.3. Funkcja wykrywająca braki magazynowe

```
CREATE OR REPLACE FUNCTION public.find_low_stock_products(
    p_min_quantity NUMERIC
)
RETURNS TABLE (
    product_id BIGINT,
    location_id BIGINT,
    quantity NUMERIC
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        i.product_id,
        i.location_id,
        i.quantity
    FROM public.inventory i
    WHERE i.quantity < p_min_quantity;
END;
$$;
```

3.4. Obliczenie stopnia realizacji dostawy

- pobranie ilości oczekiwanych i faktycznie odebranych produktów,
- zsumowanie wartości dla całej dostawy,
- obliczenie procentowego stopnia realizacji przyjęcia.

Listing 3.4. Funkcja obliczająca stopień realizacji dostawy

```
CREATE OR REPLACE FUNCTION public.receipt_completion_percentage(
    p_receipt_id BIGINT
)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
DECLARE
    expected_sum NUMERIC := 0;
    received_sum NUMERIC := 0;
BEGIN
    SELECT
        COALESCE(SUM(expected_quantity), 0),
        COALESCE(SUM(received_quantity), 0)
    INTO expected_sum, received_sum
    FROM public.receipt_details
    WHERE receipt_id = p_receipt_id;

    IF expected_sum = 0 THEN
        RETURN 0;
    END IF;

    RETURN ROUND((received_sum / expected_sum) * 100, 2);
END;
$$;
```

3.5. Najaktywniejsi klienci w danym okresie

- pobranie wysyłek z wybranego okresu,
- zliczenie liczby wysyłek dla każdego klienta,
- wybranie klientów przekraczających ustalony próg aktywności.

Listing 3.5. Funkcja identyfikująca najaktywniejszych klientów

```
CREATE OR REPLACE FUNCTION public.get_top_clients(
    p_from DATE,
    p_to DATE,
    p_min_shipments INT
)
RETURNS TABLE (
    client_id BIGINT,
    shipments_count INT
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        s.client_id,
        COUNT(*)::INT AS shipments_count
    FROM public.shipments s
    WHERE s.shipment_date BETWEEN p_from AND p_to
    GROUP BY s.client_id
    HAVING COUNT(*) >= p_min_shipments;
```

```
END;
$$;
```

3.6. Kontrola pojemności lokalizacji magazynowej

- zsumowanie ilości produktów przypisanych do lokalizacji,
- pobranie maksymalnej pojemności lokalizacji,
- porównanie wartości i zwrócenie wyniku kontroli.

Listing 3.6. Funkcja kontrolująca pojemność lokalizacji

```
CREATE OR REPLACE FUNCTION public.check_location_capacity(
    p_location_id BIGINT
)
RETURNS BOOLEAN
LANGUAGE plpgsql
AS $$
DECLARE
    total_quantity NUMERIC := 0;
    v_max_capacity NUMERIC;
BEGIN
    SELECT COALESCE(SUM(i.quantity), 0)
    INTO total_quantity
    FROM public.inventory i
    WHERE i.location_id = p_location_id;

    SELECT l.max_capacity
    INTO v_max_capacity
    FROM public.locations l
    WHERE l.location_id = p_location_id;

    IF v_max_capacity IS NULL THEN
        RETURN TRUE;
    END IF;

    RETURN total_quantity <= v_max_capacity;
END;
$$;
```

3.7. Aktualizacja stanu magazynu po przyjęciu dostawy

- pobranie danych o faktycznie odebranych ilościach produktów,
- dopasowanie produktów do odpowiednich rekordów magazynowych,
- aktualizacja ilości oraz daty ostatniej modyfikacji.

Listing 3.7. Procedura aktualizująca stany magazynowe po przyjęciu

```
CREATE OR REPLACE PROCEDURE public.update_inventory_after_receipt(
    p_receipt_id BIGINT
)
LANGUAGE plpgsql
```

```
AS $$
BEGIN
    UPDATE public.inventory i
    SET quantity = i.quantity + rd.received_quantity,
        last_updated = NOW()
    FROM public.receipt_details rd
    WHERE rd.receipt_id = p_receipt_id
      AND rd.product_id = i.product_id;
END;
$$;
```

3.8. Ocena produktywności pracowników

- pobranie danych o wysyłkach realizowanych przez pracowników,
- zliczenie liczby unikalnych klientów,
- określenie poziomu produktywności pracownika.

Listing 3.8. Funkcja oceniająca produktywność pracowników

```
CREATE OR REPLACE FUNCTION public.employee_productivity()
RETURNS TABLE (
    employee_id BIGINT,
    clients_count INT,
    productivity_level TEXT
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        s.employee_id,
        COUNT(DISTINCT s.client_id)::INT AS clients_count,
        CASE
            WHEN COUNT(DISTINCT s.client_id) > 20 THEN 'HIGH'
            WHEN COUNT(DISTINCT s.client_id) BETWEEN 10 AND 20 THEN 'MEDIUM'
            ELSE 'LOW'
        END AS productivity_level
    FROM public.shipments s
    WHERE s.employee_id IS NOT NULL
    GROUP BY s.employee_id;
END;
$$;
```

3.9. Najczęściej wysyłane produkty

- pobranie danych dotyczących wysyłanych produktów,
- zsumowanie ilości wysyłek dla poszczególnych produktów,
- wybranie produktów przekraczających ustalony próg.

Listing 3.9. Funkcja identyfikująca najczęściej wysyłane produkty

```
CREATE OR REPLACE FUNCTION public.most_shipped_products(
    p_min_quantity NUMERIC
)
RETURNS TABLE (
    product_id BIGINT,
    total_quantity NUMERIC
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        sd.product_id,
        SUM(sd.quantity_to_ship) AS total_quantity
    FROM public.shipment_details sd
    GROUP BY sd.product_id
    HAVING SUM(sd.quantity_to_ship) >= p_min_quantity;
END;
$$;
```

3.10. Walidacja poprawności dostawy

- pobranie danych dotyczących ilości oczekiwanych i odebranych,
- sprawdzenie poprawności wartości,
- zatwierdzenie lub odrzucenie dostawy.

Listing 3.10. Funkcja walidująca poprawność dostawy

```
CREATE OR REPLACE FUNCTION public.validate_receipt(
    p_receipt_id BIGINT
)
RETURNS BOOLEAN
LANGUAGE plpgsql
AS $$
BEGIN
    IF EXISTS (
        SELECT 1
        FROM public.receipt_details
        WHERE receipt_id = p_receipt_id
          AND received_quantity > expected_quantity
    ) THEN
        RETURN FALSE;
    END IF;

    RETURN TRUE;
END;
$$;
```

3.11. Wymagania CRUD systemu

System zarządzania magazynem musi zapewniać pełną obsługę operacji typu CRUD (Create, Read, Update, Delete) dla wszystkich kluczowych encji występujących w bazie danych. Operacje te są niezbędne do prawidłowego funkcjonowania systemu oraz realizacji zapytań problemowych przedstawionych w dalszej części rozdziału.

Create (tworzenie danych)

Operacje tworzenia danych umożliwiają dodawanie nowych rekordów do bazy danych, takich jak:

- nowi klienci i dostawcy,
- produkty oraz ich kategorie,
- lokalizacje magazynowe,
- przyjęcia towarów (receipts) oraz wysyłki (shipments),
- szczegóły przyjęć i wysyłek.

Operacje `CREATE` realizowane są przy użyciu procedur składowanych, które dodatkowo wykonują walidację danych wejściowych.

Read (odczyt danych)

Operacje odczytu danych umożliwiają pobieranie informacji niezbędnych do analizy stanu magazynu i podejmowania decyzji biznesowych. Obejmują one m.in.:

- przegląd aktualnych stanów magazynowych,
- listę klientów, dostawców i pracowników,
- historię wysyłek i przyjęć towarów,
- dane wykorzystywane w zapytaniach analitycznych.

Operacje `READ` realizowane są za pomocą funkcji zwracających pojedyncze rekordy lub zbiory danych.

Update (aktualizacja danych)

Operacje aktualizacji danych umożliwiają modyfikację istniejących rekordów w bazie danych, w szczególności:

- zmianę statusów wysyłek i dostaw,
- aktualizację stanów magazynowych,
- korektę danych klientów, produktów i lokalizacji,
- rejestrację faktycznie odebranych ilości towarów.

Operacje `UPDATE` są realizowane przez procedury składowane, co zapewnia spójność danych i kontrolę poprawności modyfikacji.

Delete (usuwanie danych)

Operacje usuwania danych pozwalają na eliminację nieaktualnych lub błędnie wprowadzonych rekordów, takich jak:

- nieużywane produkty lub kategorie,
- błędnie dodane rekordy magazynowe,
- anulowane wysyłki lub przyjęcia.

Operacje `DELETE` są zabezpieczone mechanizmami integralności referencyjnej, co zapobiega usuwaniu danych powiązanych z innymi encjami systemu.

Spełnienie powyższych wymagań CRUD stanowi podstawę do realizacji zaawansowanych funkcjonalności systemu, w tym zapytań problemowych opisanych w kolejnych podrozdziałach.

4. Przedstawienie powstałej bazy danych i wdrożonych mechanizmów

4.1. Realizacja mechanizmu CRUD

W systemie zarządzania magazynem zaimplementowano mechanizm CRUD (*Create, Read, Update, Delete*), który umożliwia wykonywanie podstawowych operacji na danych przechowywanych w relacyjnej bazie danych PostgreSQL. Mechanizm ten został zrealizowany przy użyciu procedur oraz funkcji składowanych napisanych w języku *PL/pgSQL*.

W celu czytelnego zaprezentowania działania mechanizmu CRUD, jego implementację przedstawiono na przykładzie tabeli **clients**, przechowującej dane klientów obsługiwanych przez system. Dla tej tabeli przygotowano komplet procedur i funkcji umożliwiających dodawanie, odczyt, aktualizację oraz usuwanie danych.

4.1.1. Operacja Create

Operacja *Create* odpowiada za dodawanie nowych rekordów do tabeli `clients`. Zrealizowano ją w postaci procedury składowanej, która przyjmuje dane klienta jako parametry wejściowe. Przed zapisaniem danych wykonywana jest walidacja poprawności najważniejszych pól, takich jak nazwa firmy oraz adres e-mail. Dzięki temu system zapobiega zapisywaniu niepoprawnych danych w bazie.

Listing 4.1. Procedura dodawania klienta

```
CREATE OR REPLACE PROCEDURE public.clients_create(
    IN p_company_name text,
    IN p_delivery_address text,
    IN p_phone text,
    IN p_email text,
    IN p_tax_id text
)
LANGUAGE plpgsql
AS $$
BEGIN
    IF p_company_name IS NULL OR trim(p_company_name) = '' THEN
        RAISE EXCEPTION 'Company␣name␣cannot␣be␣empty';
    END IF;

    IF p_email IS NOT NULL AND position('@' IN p_email) = 0 THEN
        RAISE EXCEPTION 'Invalid␣email␣format:␣%', p_email;
    END IF;

    INSERT INTO clients(
        company_name, delivery_address, phone, email, tax_id
    )
    VALUES (
        p_company_name, p_delivery_address, p_phone, p_email, p_tax_id
    );
END;
$$;
```

4.1.2. Operacja Read

Operacja *Read* umożliwia odczyt danych klienta z bazy danych. Została ona zaimplementowana w postaci funkcji składowanej, która na podstawie identyfikatora klienta zwraca kompletny rekord zawierający jego dane. W przypadku próby odczytu nieistniejącego rekordu funkcja zgłasza wyjątek informujący o braku danych.

Listing 4.2. Funkcja odczytu danych klienta

```
CREATE OR REPLACE FUNCTION public.clients_read_one(
    p_client_id bigint
)
RETURNS public.clients
LANGUAGE plpgsql
AS $$
DECLARE
    v_row clients;
BEGIN
    SELECT *
    INTO v_row
    FROM clients
    WHERE client_id = p_client_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Client␣with␣id␣%␣not␣found', p_client_id;
    END IF;

    RETURN v_row;
END;
$$;
```

4.1.3. Operacja Update

Operacja *Update* odpowiada za aktualizację istniejących danych klienta. Została zrealizowana w formie procedury składowanej, która umożliwia modyfikację wybranych pól rekordu. Procedura sprawdza istnienie klienta w bazie danych oraz wykonuje podstawową walidację danych wejściowych przed dokonaniem aktualizacji.

Listing 4.3. Procedura aktualizacji danych klienta

```
CREATE OR REPLACE PROCEDURE public.clients_update(
    IN p_client_id bigint,
    IN p_company_name text,
    IN p_delivery_address text,
    IN p_phone text,
    IN p_email text,
    IN p_tax_id text
)
LANGUAGE plpgsql
AS $$
BEGIN
    IF NOT EXISTS (
        SELECT 1 FROM clients WHERE client_id = p_client_id
    ) THEN
        RAISE EXCEPTION 'Client␣with␣id␣%␣does␣not␣exist', p_client_id;
    END IF;
```

```
    UPDATE clients
    SET company_name     = COALESCE(p_company_name, company_name),
        delivery_address = COALESCE(p_delivery_address, delivery_address),
        phone            = COALESCE(p_phone, phone),
        email            = COALESCE(p_email, email),
        tax_id           = COALESCE(p_tax_id, tax_id)
    WHERE client_id = p_client_id;
END;
$$;
```

4.1.4. Operacja Delete

Operacja *Delete* służy do usuwania danych klienta z systemu. Została ona zaimplementowana w postaci procedury składowanej, która przed usunięciem rekordu sprawdza jego istnienie. Dodatkowo obsługiwana jest sytuacja naruszenia integralności referencyjnej, np. gdy klient jest powiązany z wysyłkami, co uniemożliwia jego usunięcie.

Listing 4.4. Procedura usuwania klienta

```
CREATE OR REPLACE PROCEDURE public.clients_delete(
    IN p_client_id bigint
)
LANGUAGE plpgsql
AS $$
BEGIN
    IF NOT EXISTS (
        SELECT 1 FROM clients WHERE client_id = p_client_id
    ) THEN
        RAISE EXCEPTION 'Client␣with␣id␣%␣does␣not␣exist', p_client_id;
    END IF;

    DELETE FROM clients WHERE client_id = p_client_id;

EXCEPTION
    WHEN foreign_key_violation THEN
        RAISE EXCEPTION
            'Cannot␣delete␣client␣%,␣client␣is␣used␣in␣shipments',
            p_client_id;
END;
$$;
```

Bibliografia

Spis rysunków

Spis listingów