Uniwersytet Rzeszowski
Kolegium Nauk Przyrodniczych
Instytut Informatyki

Bazy Danych

Projekt: Zarządzanie sklepem rowerowym

Wykonał:

Sebastian Szmigielski,SS131521

Oleksandr Komkov,OK131454

Prowadzący: dr inż. Piotr Grochowalski

Rzeszów 2025

Spis treści

1. Opis założeń projektu

Cel projektu

Celem projektu jest opracowanie kompleksowego systemu zarządzania serwisem rowerowym, który umożliwi przechowywanie i organizowanie informacji o rowerach, klientach, usługach serwisowych, zleceniach oraz pracownikach. System zapewni użytkownikom dostęp do kluczowych informacji, eliminując konieczność ręcznego monitorowania danych i zwiększając efektywność zarządzania serwisem.

Opis problemu

Głównym problemem, który zostanie rozwiązany, jest brak centralnego i zautomatyzowanego systemu do zarządzania serwisem rowerowym. Aktualnie wiele serwisów rowerowych korzysta z arkuszy kalkulacyjnych lub rozproszonych systemów, co prowadzi do problemów związanych z błędami w danych, trudnościami w aktualizacji informacji oraz brakiem dostępu do kluczowych statystyk w czasie rzeczywistym.

Problem jest istotny, ponieważ nieefektywne zarządzanie danymi serwisowymi wpływa negatywnie na organizację pracy serwisu, co może prowadzić do opóźnień w realizacji zleceń, błędów w obsłudze klientów oraz trudności w analizie efektywności pracowników. Dowody na istnienie problemu obejmują trudności serwisów rowerowych w gromadzeniu i aktualizowaniu danych, a także niską dostępność dokładnych statystyk serwisowych dla menedżerów oraz pracowników.

Proponowane rozwiązanie

Aby rozwiązać ten problem, konieczne jest opracowanie zintegrowanego systemu do zarządzania serwisem rowerowym, który umożliwi:

- Automatyczne przetwarzanie i aktualizację danych serwisowych w bazie danych,
- Dynamiczne generowanie statystyk dotyczących rowerów, klientów, usług i pracowników,
- Dostęp do informacji o zleceniach w czasie rzeczywistym,
- Zarządzanie danymi rowerów, klientów, usług i pracowników w jednym miejscu,
- Eliminację błędów wynikających z ręcznego przetwarzania danych.

System będzie wykorzystywał nowoczesne technologie, w tym Flask, REST API oraz PostgreSQL, zapewniając wysoką wydajność, skalowalność oraz bezpieczeństwo przechowywanych danych.

Kroki realizacji projektu:

1. **Analiza wymagań i projektowanie systemu:**
    - Określenie kluczowych informacji aplikacji.
    - Zaprojektowanie struktury bazy danych oraz API.
    - Opracowanie modeli danych dla rowerów, klientów, usług, zleceń i pracowników.

2. **Opracowanie prototypu aplikacji:**
    - Stworzenie podstawowej wersji interfejsu użytkownika.
    - Wdrożenie pierwszych funkcjonalności backendu oraz bazy danych.
    - Testy działania podstawowych operacji CRUD.
3. **Implementacja kluczowych funkcji systemu:**
    - Zarządzanie danymi (dodawanie, usuwanie, edycja).
    - Rejestrowanie zleceń serwisowych oraz generowanie statystyk.
    - Obsługa historii serwisowej rowerów.
    - Obsługa przypisywania zleceń do pracowników.

Efekt końcowy

Efektem końcowym projektu będzie w pełni funkcjonalna aplikacja "Serwis Rowerowy", która umożliwi kompleksowe zarządzanie serwisem rowerowym, znacząco zwiększając efektywność pracy oraz dostępność statystyk dla użytkowników. Dzięki zastosowaniu nowoczesnych technologii system zapewni szybki dostęp do aktualnych informacji o zleceniach, rowerach, klientach, usługach i pracownikach, co wpłynie na lepszą organizację pracy serwisu i zwiększy satysfakcję klientów.

Przykładowe funkcjonalności systemu:

1. **Śledzenie rowerów i ich właścicieli:**
    - Dzięki tabelom ROWERY i KLIENCI można śledzić, który klient jest właścicielem konkretnego roweru.
2. **Zarządzanie zleceniami serwisowymi:**
    - Tabela ZLECENIA pozwala na przypisanie roweru do konkretnej usługi oraz śledzenie statusu zlecenia.
3. **Obsługa pracowników:**
    - Tabela PRACOWNICY gromadzi informacje o pracownikach, ich stanowiskach oraz przypisanych zleceniach.
4. **Zarządzanie usługami:**
    - Tabela USŁUGI umożliwia zapisanie listy dostępnych usług wraz z ich ceną i opisem.

Relacje między tabelami:

1. **Relacja między ROWERY a KLIENCI:**
    - Jeden klient może mieć wiele rowerów, ale każdy rower należy do jednego klienta.
2. **Relacja między ZLECENIA a ROWERY:**
    - Jeden rower może mieć wiele zleceń, ale każde zlecenie jest przypisane do jednego roweru.
3. **Relacja między ZLECENIA a USŁUGI:**
    - Jedno zlecenie może dotyczyć wielu usług, ale każda usługa jest przypisana do jednego zlecenia.

4. **Relacja między ZLECENIA a PRACOWNICY:**
    - Jedno zlecenie jest przypisane do jednego pracownika, ale pracownik może mieć wiele zleceń.

2. Opis struktury projektu

Poniżej przedstawiono informacje dotyczące struktury projektu. Obejmują one używane środowisko programistyczne, narzędzia, minimalne wymagania sprzętowe, hierarchie klas oraz strukturę przechowywanych danych.

Środowisko programistyczne:

- **Oprogramowanie:** Visual Studio Code wraz z Docker
- **Język programowania:** Python 3.10 (backend), JavaScript (frontend)
- **Framework backendowy:** Flask 2.1.1
- **Framework frontendowy:** React
- **Baza danych:** PostgreSQL 13
- **Architektura aplikacji:** REST API

Narzędzia i biblioteki:

- **Backend:**
    - **Flask 2.1.1** – framework do budowy aplikacji webowych
    - **Flask-RESTful** – rozszerzenie Flask do tworzenia API REST
    - **psycopg2** – biblioteka do komunikacji z bazą danych PostgreSQL
- **Frontend:**
    - **React** – biblioteka do budowy interfejsów użytkownika
    - **Tailwind CSS** – framework CSS do stylizacji
    - **Vite** – narzędzie do budowania aplikacji
- **Zarządzanie zależnościami:**
    - **pip** – system zarządzania pakietami dla Pythona
    - **npm** – system zarządzania pakietami dla JavaScript
- **Kontrola wersji:**
    - **Git** – repozytorium kodu źródłowego

Struktura aplikacji:

Aplikacja jest zbudowana w oparciu o architekturę REST API, co umożliwia łatwą komunikację z klientami poprzez standardowe metody HTTP. Poniżej przedstawiono główne komponenty aplikacji:

- **Backend:**
  - **backend/main.py:** Główny plik aplikacji Flask, inicjalizujący aplikację i rejestrujący blueprinty.
  - **backend/db/db_conn.py:** Plik zawierający konfigurację połączenia z bazą danych.
  - **backend/db/db_utils.py:** Plik zawierający funkcje pomocnicze do wykonywania procedur składowanych.
  - **backend/db/misc/populate_db.py:** Skrypt do populacji bazy danych przykładowymi danymi.
  - **backend/db/misc/procedures_gen.py:** Skrypt do tworzenia procedur składowanych w bazie danych.
  - **Modele:**
    - **backend/models/client.py:** Model klienta.
    - **backend/models/bike.py:** Model roweru.
    - **backend/models/service.py:** Model usługi.
    - **backend/models/employee.py:** Model pracownika.
    - **backend/models/order.py:** Model zamówienia.
  - **Trasy:**
    - **backend/routes/client_routes.py:** Trasy związane z zarządzaniem klientami.
    - **backend/routes/bike_routes.py:** Trasy związane z zarządzaniem rowerami.
    - **backend/routes/service_routes.py:** Trasy związane z zarządzaniem usługami.
    - **backend/routes/employee_routes.py:** Trasy związane z zarządzaniem pracownikami.
    - **backend/routes/order_routes.py:** Trasy związane z zarządzaniem zamówieniami.
    - **backend/routes/export_routes.py:** Trasa do eksportu danych do pliku JSON.
- **Frontend:**
  - **frontend/shop_admin_app/src/App.jsx:** Główny plik aplikacji React, zawierający konfigurację routingu.
  - **frontend/shop_admin_app/src/components/Clients.jsx:** Komponent do zarządzania klientami.

- **frontend/shop_admin_app/src/components/Bikes.jsx:** Komponent do zarządzania rowerami.
- **frontend/shop_admin_app/src/components/Orders.jsx:** Komponent do zarządzania zamówieniami.
- **frontend/shop_admin_app/src/components/Services.jsx:** Komponent do zarządzania usługami.
- **frontend/shop_admin_app/src/components/Employees.jsx:** Komponent do zarządzania pracownikami.
- **frontend/shop_admin_app/src/components/CRUD_tabview.jsx:** Komponent do generowania widoków CRUD.

Minimalne wymagania sprzętowe:

- **Procesor:** Intel Core i3 lub równoważny
- **Pamięć RAM:** 4 GB
- **Dysk twardy:** 1 GB wolnego miejsca
- **System operacyjny:** Windows 10, macOS 10.15, lub Linux

Struktura bazy danych

Tabele:

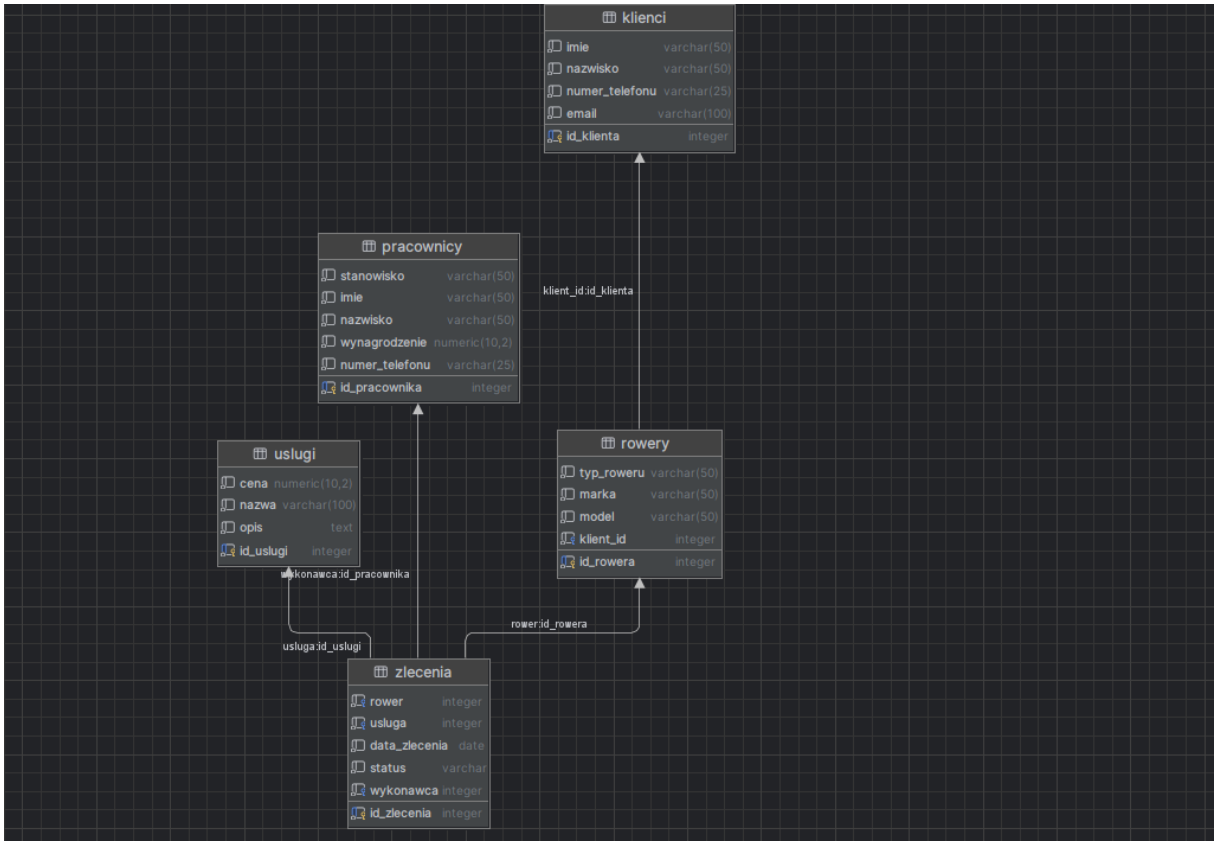
- **Tabela sklep.klienci:** Tabela przechowuje informacje o klientach sklepu. Zawiera kolumny takie jak id_klienta (klucz główny), imie, nazwisko, numer_telefonu oraz email. Każda kolumna ma odpowiedni typ danych i ograniczenia, takie jak not null dla kolumn, które muszą być wypełnione.
- **Tabela sklep.uslugi:** Tabela przechowuje informacje o usługach oferowanych przez sklep. Zawiera kolumny id_uslugi (klucz główny), cena, nazwa oraz opis. Kolumny cena, nazwa i opis są obowiązkowe (not null).
- **Tabela sklep.pracownicy:** Tabela przechowuje informacje o pracownikach sklepu. Zawiera kolumny id_pracownika (klucz główny), stanowisko, imie, nazwisko, wynagrodzenie oraz numer_telefonu. Kolumna stanowisko ma dodatkowe ograniczenie sprawdzające, które pozwala na wprowadzenie tylko określonych wartości (np. Mechanik, Księgowość, Kierownik, Sprzedawca, Magazynier).
- **Tabela sklep.rowery:** Tabela przechowuje informacje o rowerach należących do klientów. Zawiera kolumny id_rowera (klucz główny), typ_roweru, marka, model oraz klient_id, który jest kluczem obcym odnoszącym się do tabeli sklep.klienci.
- **Tabela sklep.zlecenia:** Tabela przechowuje informacje o zleceniach na usługi. Zawiera kolumny id_zlecenia (klucz główny), rower (klucz obcy odnoszący się do tabeli sklep.rowery), usluga (klucz obcy odnoszący się do tabeli sklep.uslugi), data_zlecenia, status oraz wykonawca (klucz obcy odnoszący się do tabeli sklep.pracownicy). Kolumna status ma dodatkowe ograniczenie sprawdzające, które pozwala na wprowadzenie tylko określonych wartości

Procedury (funkcje):

- **Dodawanie zlecenia:** Procedura dodaj_zlecenie sprawdza, czy podane identyfikatory roweru, usługi i pracownika istnieją w odpowiednich tabelach. Jeśli tak, dodaje nowe zlecenie do tabeli sklep.zlecenia.
- **Usuwanie zlecenia:** Procedura usun_zlecenie sprawdza, czy podany identyfikator zlecenia istnieje w tabeli sklep.zlecenia. Jeśli tak, usuwa zlecenie z tabeli.
- **Dodawanie klienta:** Procedura dodaj_klienta dodaje nowego klienta do tabeli sklep.klienci z podanymi danymi (imię, nazwisko, numer telefonu, email).
- **Aktualizowanie klienta:** Procedura aktualizuj_klienta sprawdza, czy podany identyfikator klienta istnieje w tabeli sklep.klienci. Jeśli tak, aktualizuje dane klienta.
- **Usuwanie klienta:** Procedura usun_klienta sprawdza, czy podany identyfikator klienta istnieje w tabeli sklep.klienci. Jeśli tak, usuwa klienta oraz powiązane z nim rowery i zlecenia.
- **Dodawanie roweru:** Procedura dodaj_rower sprawdza, czy podany identyfikator klienta istnieje w tabeli sklep.klienci. Jeśli tak, dodaje nowy rower do tabeli sklep.rowery.
- **Aktualizowanie roweru:** Procedura aktualizuj_rower sprawdza, czy podany identyfikator roweru istnieje w tabeli sklep.rowery oraz czy podany identyfikator klienta istnieje w tabeli sklep.klienci. Jeśli tak, aktualizuje dane roweru.
- **Usuwanie roweru:** Procedura usun_rower sprawdza, czy podany identyfikator roweru istnieje w tabeli sklep.rowery. Jeśli tak, usuwa rower oraz powiązane z nim zlecenia.
- **Dodawanie usługi:** Procedura dodaj_usluge dodaje nową usługę do tabeli sklep.uslugi z podanymi danymi (cena, nazwa, opis).
- **Aktualizowanie usługi:** Procedura aktualizuj_usluge sprawdza, czy podany identyfikator usługi istnieje w tabeli sklep.uslugi. Jeśli tak, aktualizuje dane usługi.
- **Usuwanie usługi:** Procedura usun_usluge sprawdza, czy podany identyfikator usługi istnieje w tabeli sklep.uslugi. Jeśli tak, usuwa usługę oraz powiązane z nią zlecenia.
- **Dodawanie pracownika:** Procedura dodaj_pracownika dodaje nowego pracownika do tabeli sklep.pracownicy z podanymi danymi (stanowisko, imię, nazwisko, wynagrodzenie, numer telefonu).
- **Aktualizowanie pracownika:** Procedura aktualizuj_pracownika sprawdza, czy podany identyfikator pracownika istnieje w tabeli sklep.pracownicy. Jeśli tak, aktualizuje dane pracownika.
- **Usuwanie pracownika:** Procedura usun_pracownika sprawdza, czy podany identyfikator pracownika istnieje w tabeli sklep.pracownicy. Jeśli tak, usuwa pracownika oraz powiązane z nim zlecenia.
- **Wyszukiwanie klientów z rowerami:** Procedura wyszukaj_klientow_z_rowerami zwraca listę klientów wraz z ich rowerami.
- **Lista zleceń:** Procedura lista_zlecen zwraca listę zleceń wraz z nazwą usługi, datą zlecenia oraz wykonawcą.
- **Lista rowerów klienta:** Procedura lista_rowerow_klienta zwraca listę rowerów należących do danego klienta.

- **Lista rowerów z klientem:** Procedura lista_rowerow_z_klientem zwraca listę rowerów wraz z informacjami o klientach.
- **Szczegóły zamówień:** Procedura szczegoly_zamowien zwraca szczegółowe informacje o zamówieniach, w tym dane roweru, klienta, usługi, datę zlecenia, status oraz wykonawcę.

Na rysunku 1 przedstawiono diagram ERB bazy danych



Rysunek 1- diagram ERB

3. Uruchamianie aplikacji

Poniżej przedstawiono kroki, które należy wykonać, aby uruchomić aplikację, oraz opis procesu uruchamiania.

Krok 1: Przygotowanie środowiska

Przed uruchomieniem aplikacji należy upewnić się, że na komputerze są zainstalowane następujące oprogramowanie:

- Docker
- Docker Compose

Z poziomu Windows wystaczy zainstalować Docker Desktop tutaj

Krok 2: Uruchomienie Docker Compose

Aby uruchomić aplikację, należy użyć Docker Compose, który automatycznie skonfiguruje i uruchomi wszystkie niezbędne usługi, takie jak baza danych, backend i frontend. W katalogu głównym projektu należy uruchomić następujące polecenie:

docker-compose up –build

Krok 3: Sprawdzenie działania aplikacji

Po uruchomieniu Docker Compose, aplikacja będzie dostępna pod następującymi adresami:

- **Backend:** http://localhost:5000
- **Frontend:** http://localhost:8000

Proces uruchamiania aplikacji

1. **Docker Compose:**
    - **Baza danych PostgreSQL:** Docker Compose uruchamia kontener z bazą danych PostgreSQL, tworząc nową bazę danych z użytkownikiem i hasłem zdefiniowanymi w pliku konfiguracyjnym. Kontener bazy danych jest odpowiedzialny za przechowywanie wszystkich danych aplikacji.
    - **Backend:** Docker Compose buduje obraz dla backendu, instalując wszystkie wymagane zależności. Następnie uruchamia aplikację Flask, która obsługuje logikę biznesową i komunikację z bazą danych.
    - **Frontend:** Docker Compose buduje obraz dla frontendowej aplikacji React, instalując wszystkie zależności. Następnie uruchamia serwer deweloperski Vite, który obsługuje interfejs użytkownika.
2. **Inicjalizacja bazy danych:**
    - Po uruchomieniu aplikacji Flask, backend łączy się z bazą danych PostgreSQL przy użyciu zmiennej środowiskowej DATABASE_URL.
    - Backend tworzy schemat sklep oraz tabele w bazie danych, jeśli jeszcze nie istnieją.
    - Skrypt inicjalizacyjny wstawia przykładowe dane do tabel oraz tworzy procedury składowane w bazie danych, co umożliwia wykonywanie operacji na danych.
3. **Rejestracja tras i modeli:**
    - Przed przystąpieniem do inicjacji jest uruchamiany skrypt sprawdzający połączenie z bazą danych. Przy oczekiwaniu trwającym ponad 30 sekund (z możliwością edycji w pliku DockerFile backendu) backend się wyłącza, wymuszając ponowne włączenie aplikacji
    - Backend rejestruje modele danych, takie jak Client, Bike, Service, Employee i Order. Modele te reprezentują struktury danych przechowywanych w bazie danych.
    - Backend rejestruje również trasy API, takie jak client_routes, bike_routes, service_routes, employee_routes, order_routes i

export_routes. Trasy te umożliwiają wykonywanie operacji CRUD (Create, Read, Update, Delete) na danych za pomocą zapytań http.

4. **Frontend:**
    - Aplikacja React uruchamia się na serwerze deweloperskim Vite, umożliwiając dostęp do interfejsu użytkownika pod adresem http://localhost:8000.
    - Interfejs użytkownika umożliwia zarządzanie klientami, rowerami, usługami, pracownikami i zamówieniami za pomocą odpowiednich komponentów. Użytkownicy mogą dodawać, edytować i usuwać dane.

4. Działanie aplikacji

Opis działania strony frontendowej

Strona frontendowa aplikacji jest zbudowana przy użyciu biblioteki React i frameworka Vite. Poniżej przedstawiono ogólny opis działania strony, w tym menu, tabeli, edycji, dodawania danych oraz różnych typów pól formularzy.

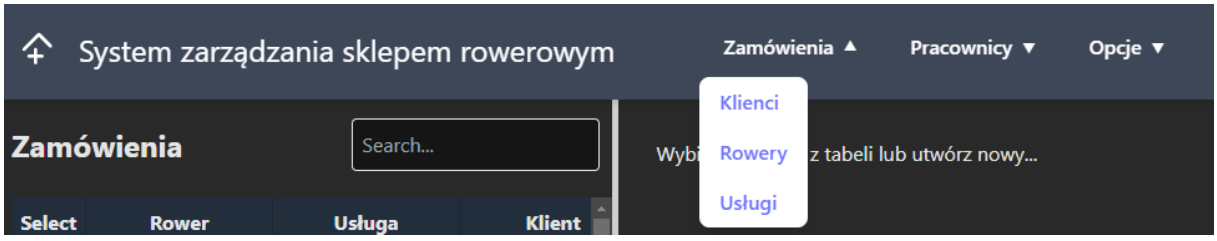
Menu nawigacyjne

Menu nawigacyjne znajduje się na górze strony i umożliwia użytkownikowi przechodzenie między różnymi sekcjami aplikacji. Menu jest responsywne i dostosowuje się do rozmiaru ekranu. W widoku mobilnym menu jest ukryte i można je otworzyć za pomocą przycisku "☰", a wciśnięcie przycisku „Home” powoduje przekierowanie na stronę główną aplikacji.

Menu zostało podzielone na następujące sekcje:

- **Zamówienia:** Dane dotyczące informacji klientów i dla klientów, takich jak lista klientów, wszystkie rowery oraz dostępne usługi.
- **Pracownicy:** Informacje dotyczące obecnych pracowników oraz przetwarzanych zamówień
- **Opcje:** Możliwość pobrania bazy danych w formie pliku JSON.

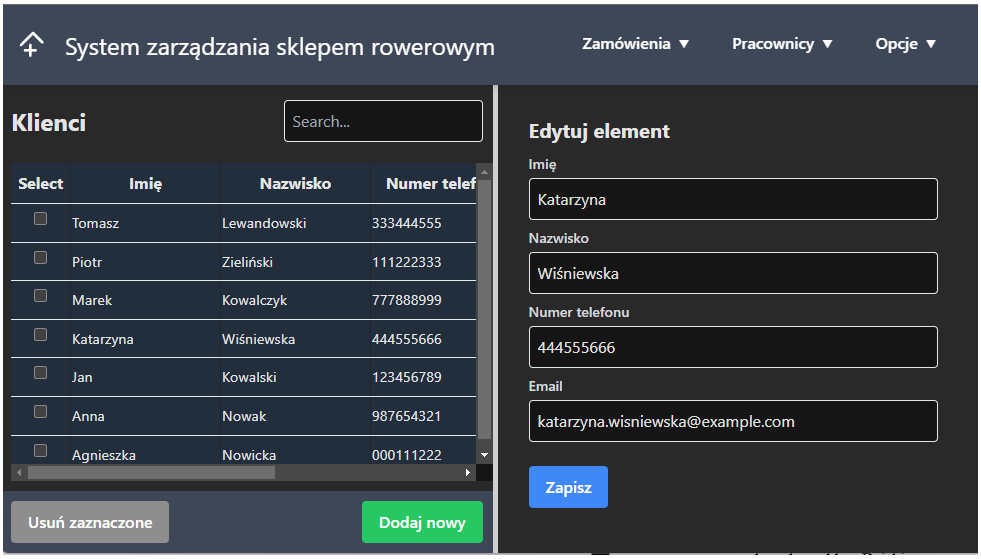
Na rysunku 2 przedstawiono menu nawigacyjne w widoku desktopowym.



Rysunek 2- menu nawigacyjne z rozwiniętym menu "Zamówienia"

Widok główny edycji danych

Po wejściu w daną zakładkę w zależności od trybu widoku (desktop/mobilny) ukaże się nam albo tabela z danymi po lewej stronie i fragment na edycję/dodanie danych po prawej stronie z możliwością dynamicznej zmiany rozmiaru danego fragmentu okna używając suwaka na środku ekranu, albo widok tabeli przy której wciśnięcie danego wpisu powoduje wyskoczenie popupa na edycję danych. Wszelkie późniejsze wyjaśnienia działania zostały oparte na widoku desktopowym. Na rysunku 3 przedstawiono przykładowy widok z tabelą klientów:

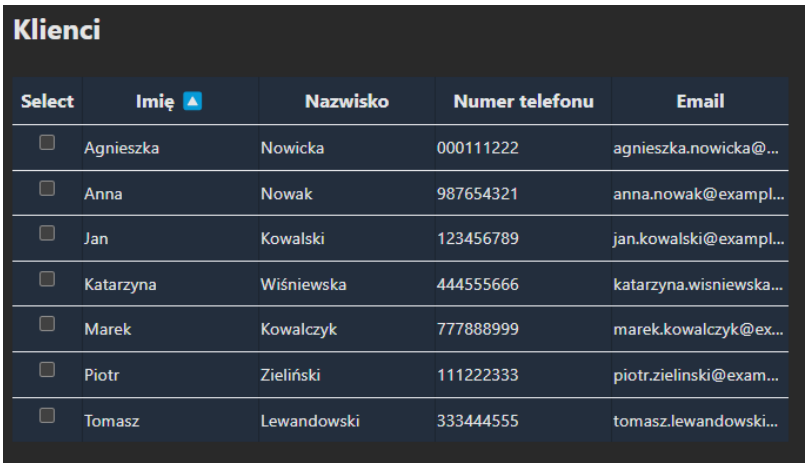


Rysunek 3 - okno aplikacji z tabelą "Klienci"

Tabela

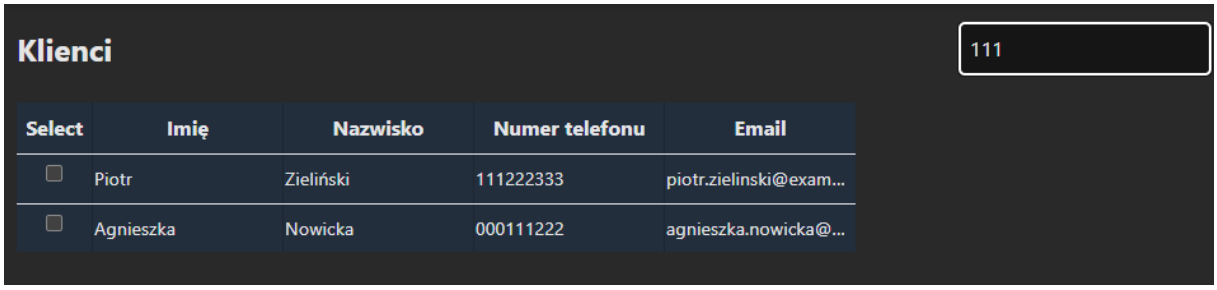
Tabela jest głównym elementem interfejsu użytkownika, który wyświetla dane z wybranej sekcji (np. lista klientów, rowerów, usług, pracowników lub zamówień). Tabela jest interaktywna i umożliwia sortowanie oraz filtrowanie danych.

Sortowanie danych odbywa się poprzez kliknięcie danego nagłówka kolumny, wyświetla się w takim momencie strzałka wskazująca, czy dane są obecnie sortowane rosnąco czy malejąco. Na rysunku 4 wskazano przykładowe sortowanie po imionach klientów:



Rysunek 4- sortowanie od "A" do "Z" klientów

Filtrowanie odbywa się poprzez wpisanie znaków w inpucie „search” nad tabelą. Wpisanie znaku przeszukuje wszystkie kolumny w której występuje dany ciąg, ukrywając z widoku te, które nie spełniają danego warunku w dowolnej kolumnie danych. Na rysunku 5 przedstawiono przykładowe wyszukanie poprzez użycie ciągu numerycznego „111” celem przeszukania numeru telefonu:

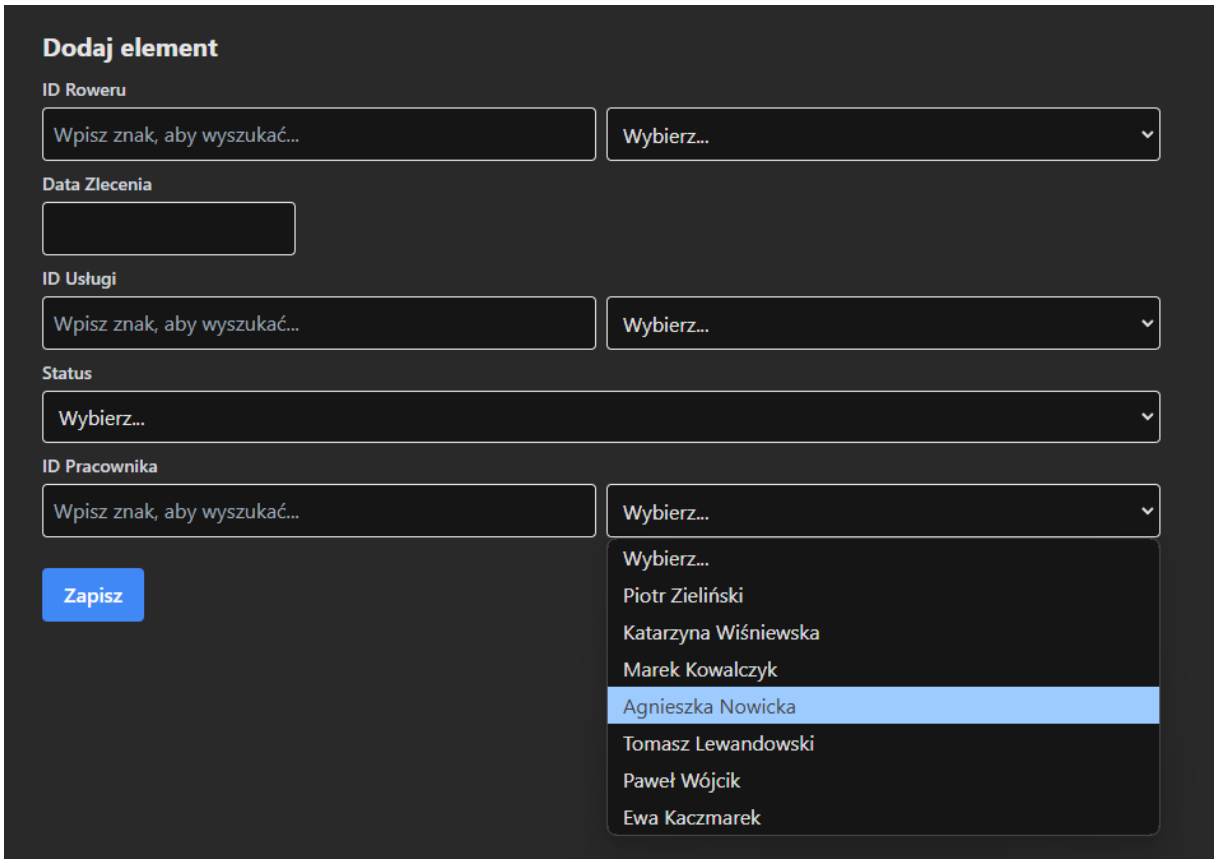


Rysunek 5- Filtrowanie danych

Edycja i dodawanie danych

Użytkownik może edytować istniejące dane lub dodawać nowe dane za pomocą formularzy. Formularze są wyświetlane po kliknięciu na wiersz tabeli (edycja) lub przycisk "Dodaj nowy" (dodawanie). Formularze zawierają różne typy pól, takie jak pola tekstowe czy pola wyboru dla danych wymagających jeden z danych typów danych. Ono edycji i dodania danych można modyfikować w kodzie w zależności od potrzeby.

Na rysunku 6 przedstawiono przykładowy formularz dodania danych



Rysunek 6 - Dodanie danych dla zleceń

Frontend realizuje również sprawdzanie danych – w przypadku, w którym zostają w danym entry wpisane niepoprawne dane (np.: imię w polu numeru telefonu) dane nie zostaną wysłane do bazy danych.

Pola formularzy

1. **Pola tekstowe:** Umożliwiają wprowadzanie tekstu. Przykładem jest pole "Imię" w formularzu klienta.
2. **Combo:** Umożliwia wybór jednej opcji z rozwijanej listy. Przykładem jest pole "Stanowisko" w formularzu pracownika.
3. **SearchCombo:** Umożliwia wyszukiwanie i wybór jednej opcji z rozwijanej listy. Po wybraniu danego wpisu danych aktualizuje się pole wyszukiwania na ID danego wpisu Przykładem jest pole "Klient" w formularzu roweru, gdzie po wybraniu danych z wyszukania daje się do pola ID danegy klient.
4. **Date:** Umożliwia wybór daty za pomocą kalendarza. Przykładem jest pole "Data Zlecenia" w formularzu zamówienia.

5. Działanie szczegółowe komponentów aplikacji

Tworzenie bazy danych, dodawanie danych i procedur

Inicjalizacja bazy danych

1. **Konfiguracja połączenia z bazą danych:** W pliku *db_conn.py* znajduje się funkcja *init_db*, która konfiguruje połączenie z bazą danych PostgreSQL przy użyciu zmiennej środowiskowej DATABASE_URL zdefiniowanej w pliku docker-compose. Funkcja ta ustawia również schemat sklep.

```
def init_db(app):
    database_url = os.getenv("DATABASE_URL")
    app.config['SQLALCHEMY_DATABASE_URI'] = database_url
    db.init_app(app)
    with app.app_context():
        db.session.execute(text("CREATE SCHEMA IF NOT EXISTS sklep"))
        db.session.commit()
```

2. **Inicjalizacja aplikacji Flask:** W pliku main.py aplikacja Flask jest inicjalizowana, a połączenie z bazą danych jest konfigurowane:

```
app = Flask(__name__)
initialize_db_conn(app)
```

3. **Tworzenie tabel:** Modele danych są importowane i rejestrowane w aplikacji Flask. Modele te definiują strukturę tabel w bazie danych:

```
from models.client import Client
from models.bike import Bike
from models.service import Service
from models.employee import Employee
from models.order import Order
```

Przykładowy model z ograniczeniami pracownika wygląda następująco:

```
class Employee(db.Model):
    __tablename__ = "pracownicy"
    __table_args__ = {'schema': 'sklep'}
    id_pracownika = db.Column(db.Integer, primary_key=True)
    stanowisko = db.Column(db.String(50), nullable=False)
    __table_args__ = (
        db.CheckConstraint(
            """stanowisko IN ('Mechanik',
            'Księgowość',
            'Kierownik',
            'Sprzedawca',
            'Magazynier')""",
            name='check_stanowisko'
        ),
        {'schema': 'sklep'}
    )
    imie = db.Column(db.String(50), nullable=False)
    nazwisko = db.Column(db.String(50), nullable=False)
    wynagrodzenie = db.Column(db.Numeric(10, 2), nullable=False)
    numer_telefonu = db.Column(db.String(25), nullable=False)
    zlecenia = db.relationship('Order', backref='sklep.wykonawca',
lazy=True)
```

4. **Inicjalizacja bazy danych:** Funkcja initialize_database tworzy tabele w bazie danych, jeśli jeszcze nie istnieją, oraz wstawia przykładowe dane:

```
def initialize_database():
    db.create_all()
    if not any(db.session.query(model).first() for model in [Client, Bike,
Service, Employee, Order]):
        populate_db.run()
        procedures_gen.create_procedures()
```

Dodawanie danych i procedur

1. **Skrypt do populacji bazy danych:** W pliku populate_db.py znajduje się skrypt, który wstawia przykładowe dane do tabel. Skrypt łączy się z bazą danych, resetuje sekwencje i wstawia dane:

```
def run():
    conn = psycopg2.connect(database_url)
    cur = conn.cursor()
    cur.execute("SET search_path TO sklep;")
    cur.executemany("INSERT INTO klienci (imie, nazwisko, numer_telefonu,
email) VALUES (%s, %s, %s, %s)", klienci)
    cur.executemany("INSERT INTO uslugi (cena, nazwa, opis) VALUES (%s, %s,
%s)", uslugi)
    cur.executemany("INSERT INTO pracownicy (stanowisko, imie, nazwisko, wyna-
grodzenie, numer_telefonu) VALUES (%s, %s, %s, %s, %s)", pracownicy)
    cur.executemany("INSERT INTO rowery (typ_roweru, marka, model, klient_id)
VALUES (%s, %s, %s, %s)", rowery)
    cur.executemany("INSERT INTO zlecenia (rower, usluga, data_zlecenia,
status, wykonawca) VALUES (%s, %s, %s, %s, %s)", zlecenia)
    conn.commit()
    cur.close()
    conn.close()
```

Przykładowe dane dawane do tabeli wyglądają następująco:

```
    pracownicy = [
        ('Mechanik', 'Piotr', 'Zieliński', 3000.00, '123123123'),
        ('Sprzedawca', 'Katarzyna', 'Wiśniewska', 2500.00, '321321321'),
        ('Mechanik', 'Marek', 'Kowalczyk', 3200.00, '456456456'),
        ('Mechanik', 'Agnieszka', 'Nowicka', 2600.00, '654654654'),
        ('Kierownik', 'Tomasz', 'Lewandowski', 4000.00, '789789789'),
        ('Magazynier', 'Paweł', 'Wójcik', 2800.00, '987987987'),
        ('Księgowość', 'Ewa', 'Kaczmarek', 3500.00, '123321123')
    ]
```

2. **Skrypt do tworzenia procedur:** W pliku procedures_gen.py znajduje się skrypt, który tworzy procedury składowane w bazie danych. Skrypt łączy się z bazą danych i wykonuje zapytania SQL tworzące procedury:

```
def create_procedures():
    conn = psycopg2.connect(database_url)
    cursor = conn.cursor()
    cursor.execute("SET search_path TO sklep;")
    cursor.execute("""
        CREATE OR REPLACE FUNCTION dodaj_klienta(p_imie VARCHAR(50),
p_nazwisko VARCHAR(50), p_numer_telefonu VARCHAR(25), p_email VARCHAR(100))
        RETURNS VOID AS $$
        BEGIN
            INSERT INTO KLIENCI (Imie, Nazwisko, Numer_telefonu, Email) VALUES
(p_imie, p_nazwisko, p_numer_telefonu, p_email);
        END;
        $$ LANGUAGE plpgsql;
    """)
    cursor.execute("""
        CREATE OR REPLACE FUNCTION dodaj_rower(p_typ_roweru VARCHAR(50),
p_marka VARCHAR(50), p_model VARCHAR(50), p_klient INT)
        RETURNS VOID AS $$
        BEGIN
            INSERT INTO ROWERY (Typ_roweru, Marka, Model, Klient_id) VALUES
(p_typ_roweru, p_marka, p_model, p_klient);
        END;
        $$ LANGUAGE plpgsql;
    """)
      # …pozostałe procedury
    conn.commit()
    cursor.close()
    conn.close()
```

Opis uruchamiania backendu i komunikacji z frontendem

Tworzenie aplikacji backendowej

1. **Inicjalizacja aplikacji Flask:** W pliku main.py aplikacja Flask jest inicjalizowana, a połączenie z bazą danych jest konfigurowane. Backend oczekuje na poprawne połączenie z bazą danych, przy 1 jej utworzeniu należy oczekiwać aż się utworzy zanim będzie można wykonać połączenie. W przypadku wystąpienia problemów aplikacja się nie uruchomi. W tym pliku również są dawane endpointy do aplikacji z użyciem przekierowania http do /api (np.; localhost:5000/api/bikes):

```python
from flask import Flask
import os
from db.db_conn import init_db as initialize_db_conn, db
from flask_cors import CORS
import db.misc.populate_db as populate_db
import db.misc.procedures_gen as procedures_gen
import time
from sqlalchemy import text

# Inicjalizacja aplikacji Flask
# (…)

# Importowanie modeli
# (…)

# Importowanie endpointów
from routes.client_routes import client_routes
from routes.bike_routes import bike_routes
# (…)

# Rejestracja blueprintów
app.register_blueprint(client_routes, url_prefix="/api")
app.register_blueprint(bike_routes, url_prefix="/api")
# (…)

# Sprawdzanie połączenia z bazą danych
def wait_for_db_connection(timeout=120):
    start_time = time.time()
    while time.time() - start_time < timeout:
        try:
            # Inicjalizacja bazy danych
            initialize_db_conn(app)
            # Próba wykonania zapytania do bazy danych
            db.session.execute(text('SELECT 1'))
            print("Połączenie z bazą danych zostało nawiązane.")
            return True
        except Exception as e:
            print(f"Nie można połączyć się z bazą danych: {e}. Ponowna
próba za 5 sekund...")
            time.sleep(5)
    print(f"Nie udało się połączyć z bazą danych w ciągu {timeout} se-
kund.")
    return False

# Inicjalizacja bazy danych (opisana wcześniej)
# (…)
```

```
if __name__ == "__main__":
    with app.app_context():
        if wait_for_db_connection():
            initialize_database()
        else:
            print("Aplikacja nie może zostać uruchomiona bez połączenia z
bazą danych.")
            exit(1)
    app.run(host="0.0.0.0", port=5000, debug=True)
```

Komunikacja z frontendem

1. **Endpointy API:** Aplikacja backendowa udostępnia różne endpointy API do zarządzania danymi klientów, rowerów, usług, pracowników i zamówień. Przykładowe endpointy znajdują się w plikach w katalogu routes.

2. **Przykładowe trasy:**
    - **Klienci:** Trasa /api/clients obsługuje operacje GET, POST, PUT i DELETE. Dla operacji CRUD, w kolejności: Read, Create, Update i Delete. Poniżej przykład operacji GET dla klientów

```
@client_routes.route('/clients', methods=['GET'])
def get_clients():
    clients = db.session.query(Client).all()
    return jsonify([client.to_dict() for client in clients]), 200
```

    - **Rowery:** Trasa /api/bikes obsługuje operacje GET, POST, PUT i DELETE. W przeciwieństwie do tabel która nie używa danych z kluczami obcymi, tutaj wykorzystano procedury dla GET, aby zwracać przykładowe dane klienta

```
@bike_routes.route('/bikes', methods=['GET'])
def get_bikes():
    result = execute_procedure('lista_rowerow_z_klientem', ())
    bikes_data = [{'id_rowera': row[0], 'typ_roweru': row[1], 'marka': row[2],
'model': row[3], 'klient': row[4], 'klient_data': row[5]} for row in result]
    return jsonify(bikes_data), 200
```

Poniżej przedstawiono przykładowy skład procedury zwracającej nietypowe dane do wyświetlenia we frontendzie:

```
CREATE OR REPLACE FUNCTION lista_rowerow_z_klientem()
RETURNS TABLE (
    id_rowera INT,
    typ_roweru VARCHAR(50),
    marka VARCHAR(50),
    model VARCHAR(50),
    klient_info INT,
    klient_data_info VARCHAR(100)
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        r.id_rowera,
        r.typ_roweru,
        r.marka,
        r.model,
        k.id_klienta as klient_info,
        CONCAT(k.imie, ' ', k.nazwisko, ' (', k.numer_telefonu, ')')::VAR-
CHAR(150) AS klient_data_info
    FROM rowery r
    JOIN klienci k ON r.klient_id = k.id_klienta;
END;
$$ LANGUAGE plpgsql;
```

Wiązanie frontendu z backendem

Inicjalizacja aplikacji frontendowej

1. **Inicjalizacja aplikacji React:** W pliku main.jsx aplikacja React jest inicjalizowana i renderowana do elementu DOM o identyfikatorze root:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

2. **Konfiguracja routingu:** W pliku App.jsx konfiguracja routingu jest realizowana za pomocą react-router-dom. Aplikacja definiuje różne trasy, które odpowiadają różnym komponentom, takim jak Clients, Bikes, Orders, Services i Employees:

```
import { BrowserRouter as Router, Route, Routes, Link } from 'react-router-
dom';
import Clients from './components/Clients';
import Bikes from './components/Bikes';
import Orders from './components/Orders';
import Services from './components/Services';
import Employees from './components/Employees';
import { useState, useEffect, useRef } from 'react';

function App() {
  // ... (pozostała część kodu)
  return (
    <Router>
      <div className="container mx-auto p-4 md:px-10">
        <header className="fixed top-0 left-0 right-0 z-50 flex justify-
between items-center py-4 bg-gray-700 text-white shadow-md">
          <div className="flex items-center">
            <Link to="/" className="px-4 py-2 text-white">
              <svg xmlns="http://www.w3.org/2000/svg" className="h-8 w-8"
fill="none" viewBox="0 0 24 24" stroke="currentColor">
                <path strokeLinecap="round" strokeLinejoin="round"
strokeWidth="2" d="M3 12l2-2m0 0l7-7 7 7m-9 2v8m-4-4h8" />
              </svg>
            </Link>
            <h1 className="text-lg md:text-2xl">System zarządzania sklepem ro-
werowym</h1>
          </div>
          <div className="relative flex items-center">
            <button
              className="md:hidden text-black-white"
              onClick={() => setMenuOpen(!menuOpen)}
            >
              ☰
            </button>
            <nav
              ref={menuRef}
              className={`fixed top-0 right-0 h-full bg-gray-800 text-white
transform ${
                menuOpen ? 'translate-x-0' : 'translate-x-full'
              } transition-transform duration-300 ease-out md:relative
md:transform-none md:bg-transparent md:text-white md:shadow-none`}
            >
              <div className="flex justify-end p-4 md:hidden">
```

```
                <button onClick={closeMenu} className="text-white text-xl
font-bold">×</button>
              </div>
              <ul className="flex flex-col md:flex-row md:items-center space-
y-4 md:space-y-0 md:space-x-4">
                <li className="group relative">
                  <span
                    className="block px-2 py-1 md:inline-block"
                    onClick={() => handleMenuClick('orders-menu')}
                  >
                    Zamówienia <span className={`arrow ${activeMenu ===
'orders-menu' ? 'rotate-180' : ''}`}>▼</span>
                  </span>
                  <ul
                    id="orders-menu"
                    className={`absolute left-0 mt-2 bg-white text-black
rounded-lg shadow-lg ${
                      activeMenu === 'orders-menu' ? 'block' : 'hidden'
                    }`}
                  >
                    <li>
                      <Link to="/clients" className="block px-4 py-2"
onClick={closeMenu}>
                        Klienci
                      </Link>
                    </li>
                    <li>
                      <Link to="/bikes" className="block px-4 py-2"
onClick={closeMenu}>
                        Rowery
                      </Link>
                    </li>
                    <li>
                      <Link to="/services" className="block px-4 py-2"
onClick={closeMenu}>
                        Usługi
                      </Link>
                    </li>
                  </ul>
                </li>
                <li className="group relative">
                  <span
                    className="block px-2 py-1 md:inline-block"
                    onClick={() => handleMenuClick('employees-menu')}
                  >
                    Pracownicy <span className={`arrow ${activeMenu ===
'employees-menu' ? 'rotate-180' : ''}`}>▼</span>
                  </span>
```

```
                <ul
                  id="employees-menu"
                  className={`absolute left-0 mt-2 bg-white text-black
rounded-lg shadow-lg ${
                    activeMenu === 'employees-menu' ? 'block' : 'hidden'
                  }`}
                >
                  <li>
                    <Link to="/employees" className="block px-4 py-2"
onClick={closeMenu}>
                      Lista pracowników
                    </Link>
                  </li>
                  <li>
                    <Link to="/orders" className="block px-4 py-2"
onClick={closeMenu}>
                      Zlecenia
                    </Link>
                  </li>
                </ul>
              </li>
              <li className="group relative">
                <span
                  className="block px-2 py-1 md:inline-block"
                  onClick={() => handleMenuClick('options-menu')}
                >
                  Opcje <span className={`arrow ${activeMenu === 'options-
menu' ? 'rotate-180' : ''}`}>▼</span>
                </span>
                <ul
                  id="options-menu"
                  className={`absolute left-0 mt-2 bg-white text-black
rounded-lg shadow-lg ${
                    activeMenu === 'options-menu' ? 'block' : 'hidden'
                  }`}
                >
                  <li>
                    <Link to="#" onClick={exportData} className="block px-4
py-2 text-blue-600">
                      Eksportuj bazę do pliku JSON
                    </Link>
                  </li>
                </ul>
              </li>
            </ul>
          </nav>
        </div>
      </header>
```

```
        <div className="pt-20" onClick={closeMenu}>
          <Routes>
            <Route path="/" element={<h1 className="text-3xl font-bold">Witamy
w systemie zarządzania sklepem rowerowym</h1>} />
            <Route path="/clients" element={<Clients />} />
            <Route path="/bikes" element={<Bikes />} />
            <Route path="/orders" element={<Orders />} />
            <Route path="/services" element={<Services />} />
            <Route path="/employees" element={<Employees />} />
          </Routes>
        </div>
      </div>
    </Router>
  );
}

export default App;
```

3. **Komunikacja z backendem – przykład z Clients:** W pliku Clients.jsx komponent Clients komunikuje się z backendem za pomocą fetch API, aby pobrać, dodać, zaktualizować i usunąć dane klientów. Ten komponent również jest dzieckiem innego komponentu obsługującego inputy i wyświetlanie tabeli (CRUD_tabview.jsx), a także weryfikację danych, zostaje to jednak wyjaśnione w następnej sekcji:

```
import React from 'react';
import CRUDTabView from './CRUD_tabview';

function Clients() {
  const validateInput = (selectedItem) => {
// (…) };

  const updateItemFields = ['imie', 'nazwisko', 'numer_telefonu', 'email'];
  const addItemFields = ['imie', 'nazwisko', 'numer_telefonu', 'email'];

  return (
    <CRUDTabView
      apiUrl="http://localhost:5000/api/clients"
      itemFields={['imie', 'nazwisko', 'numer_telefonu', 'email']}
      itemIdentifier="id_klienta"
      fieldLabels={{
        imie: 'Imię',
        nazwisko: 'Nazwisko',
        numer_telefonu: 'Numer telefonu',
        email: 'Email'
      }}
      validateInput={validateInput}
```

```
      title="Klienci"
      enableButtons={[true, true]}
      updateItemFields={updateItemFields}
      addItemFields={addItemFields}
      fieldOptions={{}} // No specific field options for clients
    />
  );
}

export default Clients;
```

Działanie frontendowej części strony z tabelą oraz danymi

Widok i obsługa dividera

1. **Widok:** Widok aplikacji jest zbudowany przy użyciu komponentu CRUDTabView, który jest odpowiedzialny za wyświetlanie tabeli z danymi oraz formularzy do edycji i dodawania nowych rekordów. Komponent ten jest używany w różnych sekcjach aplikacji, takich jak Clients, Bikes, Orders, Services, i Employees.

2. **Obsługa dividera:** Divider jest elementem, który umożliwia użytkownikowi zmianę szerokości paneli w widoku desktopowym. Divider jest obsługiwany przez funkcję handleMouseDown, która rejestruje zdarzenia mousemove i mouseup w celu dynamicznej zmiany szerokości paneli:

```
const handleMouseDown = (e) => {
  if (isMobileView) return; // Disable divider in mobile view

  const startX = e.clientX;

  const handleMouseMove = (e) => {
    const newDividerPosition = Math.min(90, Math.max(10, (e.clientX /
window.innerWidth) * 100));
    setDividerPosition(newDividerPosition);
  };

  const handleMouseUp = () => {
    document.removeEventListener('mousemove', handleMouseMove);
    document.removeEventListener('mouseup', handleMouseUp);
    document.body.style.userSelect = ''; // Re-enable text selection
  };

  document.addEventListener('mousemove', handleMouseMove);
  document.addEventListener('mouseup', handleMouseUp);
  document.body.style.userSelect = 'none'; // Disable text selection
};
```

3. **Widok mobilny:** W widoku mobilnym, formularze do edycji i dodawania nowych rekordów są wyświetlane w popupie, który zajmuje całą szerokość ekranu. Popup jest wyświetlany po kliknięciu na wiersz tabeli lub przycisk "Dodaj nowy":

```
{isMobileView && showPopup && (
  <div className="popup-container Mobile">
    <div className="popup-content">
      <button
        onClick={handleClosePopupWrapper}
        className="absolute top-2 right-2 text-gray-500 hover:text-gray-700"
      >
        X
      </button>
      {selectedItem ? (
        <div>
          <h3 className="text-xl font-bold" style={{ paddingBottom: '10px' }}>
            {selectedItem && selectedItem[itemIdentifier] ? 'Edytuj element' :
'Dodaj element'}
          </h3>
          {/* Formularz */}
        </div>
      ) : (
        <p>Wybierz element z tabeli lub utwórz nowy...</p>
      )}
    </div>
  </div>
)}
```

Działanie elementów strony

1. **Tabela:** Tabela jest renderowana przy użyciu biblioteki react-table, która umożliwia sortowanie, filtrowanie i zmianę rozmiaru kolumn. Tabela jest zdefiniowana w komponencie CRUDTabView:

```
const { getTableProps, getTableBodyProps, headerGroups, rows, prepareRow } =
useTable(
  { columns, data },
  useFilters,
  useSortBy,
  useResizeColumns,
  useFlexLayout
);

return (
  <table {...getTableProps()} className="min-w-full bg-white dark:bg-gray-
800">
    <thead>
      {headerGroups.map((headerGroup) => (
```

```
        <tr {...headerGroup.getHeaderGroupProps()}>
          {headerGroup.headers.map((column) => (
            <th
              {...column.getHeaderProps(column.getSortByToggleProps())}
              className="py-2 align-middle"
            >
              {column.render('Header')}
              {column.isSorted ? (column.isSortedDesc ? ' 🔽' : ' 🔼') : ''}
              {column.canResize && <div {...column.getResizerProps()}
className="resizer" />}
            </th>
          ))}
        </tr>
      ))}
    </thead>
    <tbody {...getTableBodyProps()}>
      {rows.map((row) => {
        prepareRow(row);
        return (
          <tr
            {...row.getRowProps()}
            className={`border-t ${highlightedRow === row.id ? 'highlighted' :
''}`}
            onClick={() => handleItemClick(row.original)}
          >
            {row.cells.map((cell) => (
              <td {...cell.getCellProps()} className="py-2 cell-content">
                {cell.render('Cell')}
              </td>
            ))}
          </tr>
        );
      })}
    </tbody>
  </table>
);
```

2. **Zwykły input:** Zwykły input jest używany do wprowadzania tekstu. Jest renderowany jako pole tekstowe:

```
<input
  type="text"
  name={field}
  value={selectedItem[field] || ''}
  onChange={handleInputChange}
  className="mt-1 p-2 border rounded w-full"
/>
```

3. **Combo:** Combo jest używane do wyboru jednej opcji z rozwijanej listy. Jest renderowane jako pole select:

```
<select
  name={field}
  value={selectedItem[field] || ''}
  onChange={handleInputChange}
  className="mt-1 p-2 border rounded w-full"
>
  <option value="">{selectedItem[field] || 'Wybierz...'}</option>
  {fieldOptions[field].options.map((option) => (
    <option key={option.value} value={option.value}>
      {option.label}
    </option>
  ))}
</select>
```

4. **SearchCombo:** SearchCombo jest używane do wyszukiwania i wyboru jednej opcji z rozwijanej listy. Jest renderowane jako kombinacja pola tekstowego i pola select:

```
<div className="flex justify-between space-x-2">
  <input
    type="text"
    name={field}
    placeholder="Wpisz znak, aby wyszukać..."
    value={selectedItem[field] || ''}
    onChange={(e) => {
      handleInputChange(e);
      onSearchTermChange(e.target.value);
    }}
    className="mt-1 p-2 border rounded w-1/2"
  />
  <select
    name={field}
    value={selectedItem[field] || ''}
    onChange={handleInputChange}
    className="mt-1 p-2 border rounded w-1/2"
  >
    <option value="">Wybierz...</option>
    {fieldOptions[field].options
      .filter(option =>

option.searchTerm.toLowerCase().includes((selectedItem[field]?.toString().toLo
werCase() || '')))
      .map((option) => (
        <option key={option.value} value={option.value}>
          {option.label}
        </option>
```

```
      ))}
  </select>
</div>
```

5. **Date:** Date jest używane do wyboru daty za pomocą kalendarza. Jest renderowane przy użyciu komponentu DatePicker:

```
<DatePicker
  selected={selectedItem[field] ? new Date(selectedItem[field]) : null}
  onChange={(date) => handleInputChange({ target: { name: field, value:
date.toISOString() } })}
  dateFormat="dd/MM/yyyy"
  className="mt-1 p-2 border rounded w-full"
/>
```

Weryfikacja danych

Weryfikacja danych jest realizowana przez funkcję validateInput, która sprawdza, czy wszystkie wymagane pola są wypełnione i czy wartości są poprawne. Funkcja ta zwraca obiekt z błędami, który jest następnie używany do wyświetlania komunikatów o błędach:

```
const validateInput = (selectedItem) => {
  const newErrors = {};
  if (!selectedItem.imie) newErrors.imie = 'First name is required';
  if (!selectedItem.nazwisko) newErrors.nazwisko = 'Last name is required';
  if (!selectedItem.numer_telefonu ||
!/^\d{9}$/.test(selectedItem.numer_telefonu)) {
    newErrors.numer_telefonu = 'Phone number must be 9 digits';
  }
  if (!selectedItem.email || !/\S+@\S+\.\S+/.test(selectedItem.email)) {
    newErrors.email = 'Invalid email address';
  }
  return newErrors;
};
```

6. Przekształcenie bazy z relatywnej na nierelatywną

Jako część projektu została wykonana opcja konwersji bazy danej relacyjnej na nierelacyjną, z możliwością jej pobrania z poziomu aplikacji w formacie JSON.

Cel funkcji

Celem funkcji export_data_to_json jest konwersja danych z relacyjnej bazy danych PostgreSQL na format JSON. Funkcja ta umożliwia eksportowanie danych z różnych tabel w bazie danych oraz ich relacji do struktury JSON, co ułatwia ich dalsze przetwarzanie i analizę w aplikacjach nierelacyjnych.

Proces konwersji

1. **Nawiązanie połączenia z bazą danych:** Funkcja rozpoczyna od nawiązania połączenia z bazą danych PostgreSQL przy użyciu SQLAlchemy. Użycie konstrukcji with zapewnia, że połączenie zostanie automatycznie zamknięte po zakończeniu bloku kodu, co zapobiega wyciekom zasobów.
2. **Ustawienie schematu:** Po nawiązaniu połączenia, funkcja ustawia schemat bazy danych na sklep. Dzięki temu wszystkie kolejne zapytania SQL będą wykonywane w kontekście tego schematu, co zapewnia, że dane będą pobierane z odpowiednich tabel.
3. **Pobieranie danych z tabel:** Funkcja wykonuje szereg zapytań SQL, aby pobrać dane z różnych tabel w schemacie sklep. Każde zapytanie pobiera wszystkie wiersze z odpowiedniej tabeli (klienci, rowery, uslugi, pracownicy, zlecenia). Wyniki tych zapytań są następnie konwertowane na listy słowników, gdzie każdy słownik reprezentuje jeden wiersz z tabeli. Dzięki temu dane są łatwe do przetwarzania i mogą być bezpośrednio użyte w strukturze JSON.
4. **Pobieranie relacji kluczy obcych:** Funkcja wykonuje zapytanie SQL, aby pobrać informacje o relacjach kluczy obcych w bazie danych. Zapytanie to korzysta z information_schema, które zawiera metadane o strukturze bazy danych. Wyniki tego zapytania są również konwertowane na listę słowników, gdzie każdy słownik reprezentuje jedną relację klucza obcego. Informacje te są kluczowe dla zrozumienia powiązań między różnymi tabelami w bazie danych.
5. **Tworzenie struktury JSON:** Po pobraniu wszystkich danych i relacji, funkcja tworzy strukturę JSON, łącząc wszystkie pobrane dane w jeden słownik. Każdy klucz w słowniku (clients, bikes, services, employees, orders, relationships) odpowiada jednej tabeli lub zestawowi relacji kluczy obcych, a jego wartość to lista słowników reprezentujących wiersze z tabeli. Dzięki temu wszystkie dane są zorganizowane w przejrzystą i łatwą do przetwarzania strukturę JSON.
6. **Zwracanie danych:** Na końcu funkcja zwraca utworzony słownik data, który zawiera wszystkie dane w formacie JSON. Jeśli podczas wykonywania zapytań SQL wystąpi błąd, funkcja przechwytuje wyjątek SQLAlchemyError, drukuje komunikat o błędzie i zwraca None. Dzięki temu użytkownik jest informowany o problemach z eksportem danych, a aplikacja może odpowiednio zareagować na błędy

7. Podsumowanie

Projekt "System zarządzania sklepem rowerowym" spełnia założone cele, dostarczając narzędzie do efektywnego zarządzania danymi w sklepie rowerowym. Dzięki zastosowaniu nowoczesnych technologii, aplikacja jest skalowalna, wydajna i łatwa w utrzymaniu. Responsywny design zapewnia wygodę użytkowania zarówno na urządzeniach desktopowych, jak i mobilnych.

Aplikacja może być dalej rozwijana o dodatkowe funkcjonalności, takie jak integracja z systemami płatności, zaawansowane raportowanie czy automatyzacja procesów serwisowych. Dzięki modularnej architekturze, rozszerzanie aplikacji o nowe funkcje będzie stosunkowo proste i efektywne.

8. Użyte materiały

- https://react.dev/
- https://vite.dev/
- https://flask.palletsprojects.com/en/stable/#api-reference
- https://vite.dev/guide/
- https://nodejs.org/en/learn/getting-started/introduction-to-nodejs
- https://docs.docker.com/get-started/