Collegium Witelona Uczelnia Państwowa w Legnicy
Wydział Nauk Technicznych i Ekonomicznych
Kierunek: Informatyka

Projekt z przedmiotu "Projektowanie i programowanie systemów internetowych II"

Temat: FuelApp.

Autorzy
Mateusz Bogacz-Drewniak, nr. indeksu: 44491
Mateusz Chimkowski, nr. indeksu: 43831
Szymon Mikołajek, nr. indeksu: 41105
Paweł Kruk, nr. indeksu: 43854
Michał Nocuń, nr. indeksu: 40669

Prowadzący przedmiot
mgr inż. Krzysztof Rewak

Legnica, 2025

Spis treści

1 Podział obowiązków i odpowiedzialności w zespole

Poniższa tabela przedstawia przypisanie ról oraz zakres odpowiedzialności poszczególnych członków zespołu realizującego projekt FuelApp.

Imię i Nazwisko	**Pełniona rola i zakres obowiązków**
Mateusz Bogacz-Drewniak	**Team Leader / Backend / DevOps (wsparcie)** Zarządzanie zespołem, architektura systemu, implementacja API (.NET), konfiguracja Docker/CI, wsparcie merytoryczne przy konfiguracji środowiska uruchomieniowego.
Mateusz Chimkowski	**Frontend / UX/UI** Projektowanie interfejsu użytkownika, implementacja logiki po stronie klienta.
Szymon Mikołajek	**Project Manager / Tester** Nadzór nad harmonogramem, testy jednostkowe, weryfikacja zgodności z wymaganiami.
Paweł Kruk	**Frontend / DevOps** Implementacja logiki po stronie klienta, konfiguracja środowiska uruchomieniowego, wdrożenie i konfiguracja serwera.
Michał Nocuń	**Tester** Testy funkcjonalne, raportowanie błędów, weryfikacja scenariuszy użycia.

Tabela 1: Podział ról w zespole projektowym

2 Wprowadzenie

2.1 Cel systemu

Aplikacja webowa umożliwiająca użytkownikom przeglądanie, wyszukiwanie i aktualizowanie cen paliw na stacjach benzynowych w czasie rzeczywistym. System opiera się na modelu crowdsourcingu, gdzie społeczność zgłasza zmiany cen, które są weryfikowane przez administratorów na podstawie zdjęć pylonów cenowych.

2.2 Docelowi użytkownicy

- **Użytkownik** – zalogowana osoba, która bierze czynny udział w budowaniu bazy cen, zbiera punkty i zarządza swoim profilem.
- **Administrator** – osoba odpowiedzialna za weryfikację zgłoszeń, zarządzanie stacjami, markami paliw oraz moderację użytkowników.

2.3 Główne założenia biznesowe

- **Weryfikacja wizualna:** Każda propozycja ceny musi zawierać zdjęcie dowodowe (pylon stacji), co eliminuje fałszywe dane.
- **System grywalizacji:** Ranking najlepszych użytkowników oparty na liczbie zaakceptowanych zgłoszeń (punktów).
- **Geo-pozycjonowanie:** Wyszukiwanie stacji w oparciu o aktualną lokalizację użytkownika i promień wyszukiwania.
- **Bezpieczeństwo:** Ochrona przed botami i nadużyciami poprzez system raportowania i blokowania kont (banowania).

3 Aktorzy systemu

Rola	**Opis**
Gość	Niezalogowany użytkownik. Możliwość logowania i rejestracji (system i Facebook) oraz możliwość odzyskania hasła.
Użytkownik (User)	Zalogowany użytkownik. Może dodawać propozycje cen, zarządzać swoim kontem, zgłaszać innych użytkowników i przeglądać swoje statystyki.
Administrator (Admin)	Posiada pełne uprawnienia: edycja stacji, akceptacja cen, banowanie użytkowników, zarządzanie słownikami.

4 Funkcjonalności szczegółowe

4.1 MODUŁ AUTENTYKACJI I REJESTRACJI

4.1.1 Rejestracja klasyczna

Aktorzy: Gość
Opis: Proces zakładania konta przy użyciu adresu email i hasła.
Przebieg:

1. Użytkownik wchodzi na podstronę rejestracji (`/register`).
2. Wypełnia formularz danymi:
   - Nazwa użytkownika (unikalna).
   - Adres email.
   - Hasło (wymagane min. 6 znaków, duża litera, cyfra, znak specjalny).
   - Potwierdzenie hasła.
3. System weryfikuje unikalność emaila i loginu.
4. System tworzy konto i wysyła token weryfikacyjny na adres email.
5. Użytkownik klika w link z tokenem (`/confirm-email`).
6. System aktywuje konto i umożliwia logowanie.

4.1.2 Logowanie i Rejestracja przez Facebook

Aktorzy: Gość
Opis: Szybkie logowanie/rejestracja przy użyciu OAuth.
Przebieg:

1. Użytkownik wybiera opcję "Zaloguj przez Facebook".
2. System przekierowuje do dostawcy tożsamości (Facebook).
3. Po pomyślnej autoryzacji Facebook zwraca token dostępu.
4. System backendowy weryfikuje token w Graph API.
5. Scenariusz A (Konto nie istnieje): System automatycznie tworzy konto, pobierając email i nazwę z Facebooka.
6. Scenariusz B (Konto istnieje): System loguje użytkownika.
7. System generuje ciasteczko sesyjne JWT (HttpOnly).

4.1.3 Odzyskiwanie hasła

Aktorzy: Gość
Przebieg:

1. Użytkownik podaje email w formularzu "Zapomniałem hasła".
2. System sprawdza, czy konto istnieje i jest aktywne.
3. System wysyła email z tokenem resetującym (ważny 24h).
4. Użytkownik wchodzi w link, podaje nowe hasło i je potwierdza.
5. Hasło zostaje nadpisane w bazie danych.

4.2 MODUŁ STACJI PALIW (PRZEGLĄDANIE)

4.2.1 Wyszukiwanie stacji (Lista i Mapa)

Aktorzy: Użytkownik, Administrator
Opis: Przeglądanie dostępnych stacji z możliwością filtrowania.
Przebieg:

1. Użytkownik wchodzi na widok mapy (`/map`) lub listy (`/list`).
2. Użytkownik definiuje filtry (opcjonalnie):
   - Marka stacji (np. Orlen, BP).
   - Typ paliwa (np. LPG, ON).
   - Maksymalna cena.
   - Lokalizacja i promień wyszukiwania (w km).
3. System zwraca listę stacji pasujących do kryteriów.
4. Dla widoku listy: możliwe sortowanie po cenie (rosnąco/malejąco) lub dystansie.
5. Dla widoku mapy: wyświetlenie pinezek w odpowiednich koordynatach.

4.2.2 Szczegóły stacji (Profil stacji)

Aktorzy: Użytkownik, Administrator
Przebieg:

1. Użytkownik klika w wybraną stację.
2. System wyświetla profil stacji zawierający:
   - Dane adresowe i markę.
   - Listę dostępnych paliw wraz z aktualnymi cenami.
   - Datę ostatniej aktualizacji ceny.

4.3 MODUŁ PROPOZYCJI CEN (CROWDSOURCING)

4.3.1 Zgłaszanie nowej ceny

Aktorzy: Użytkownik, Administrator
Opis: Proces aktualizacji ceny paliwa, wymagający dowodu w postaci zdjęcia.
Przebieg:

1. Użytkownik będąc na profilu stacji wybiera opcję "Zgłoś cenę".
2. Wybiera typ paliwa (np. PB95) z listy dostępnych na tej stacji.
3. Wpisuje nową cenę (format liczbowy).
4. Wgrywa zdjęcie pylonu cenowego (wymagane, formaty: JPG, PNG, WEBP).
5. Klika "Wyślij".
6. System zapisuje propozycję ze statusem "Oczekująca"(Pending).
7. Cena na stacji **nie** zmienia się automatycznie do momentu weryfikacji.

4.3.2 Przeglądanie statystyk i historii

Aktorzy: Użytkownik, Administrator
Przebieg:

1. Użytkownik wchodzi w zakładkę statystyk (`/proposals`).
2. System wyświetla:
   - Całkowitą liczbę zgłoszeń.
   - Liczbę zgłoszeń zaakceptowanych i odrzuconych.
   - Wskaźnik skuteczności (Rate) i liczbę punktów rankingowych.

4.4 MODUŁ ADMINISTRACYJNY - ZARZĄDZANIE

4.4.1 Weryfikacja propozycji cen

Aktorzy: Administrator
Opis: Kluczowy proces zapewniający wiarygodność danych.
Przebieg:

1. Administrator wchodzi do panelu propozycji (`/proposals_admin`).
2. System wyświetla listę oczekujących zgłoszeń.
3. Administrator otwiera szczegóły zgłoszenia: widzi proponowaną cenę oraz zdjęcie dowodowe.
4. Administrator podejmuje decyzję:
   - **Akceptacja:** Cena na stacji zostaje zaktualizowana, status propozycji zmienia się na "Zaakceptowana", użytkownik otrzymuje punkty.
   - **Odrzucenie:** Cena na stacji pozostaje bez zmian, status propozycji "Odrzucona".
5. System uniemożliwia ponowną weryfikację tego samego zgłoszenia.

4.4.2 Zarządzanie stacjami (CRUD)

Aktorzy: Administrator
Przebieg (Dodawanie):

1. Administrator wypełnia formularz nowej stacji (Marka, Ulica, Miasto, Kod pocztowy, Koordynaty GPS).
2. Administrator definiuje początkowe paliwa i ich ceny.
3. System tworzy stację widoczną dla wszystkich użytkowników.

Przebieg (Edycja/Przypisanie paliwa):

1. Administrator może ręcznie zmienić cenę paliwa (z pominięciem kolejki propozycji).
2. Administrator może dodać nowy typ paliwa do istniejącej stacji (np. dodanie E85 do stacji, która go wcześniej nie miała).

4.4.3 Zarządzanie użytkownikami i Banowanie

Aktorzy: Administrator
Opis: Narzędzia do utrzymania porządku w systemie.
Przebieg (Blokada konta):

1. Administrator wybiera użytkownika z listy (`/user_admin`).
2. Klika opcję "Zablokuj"(Lock-out).
3. Podaje powód blokady oraz czas trwania (ilość dni lub blokada stała).
4. System blokuje dostęp użytkownikowi i wysyła mu email z powodem bana.
5. Wszystkie aktywne bany użytkownika są nadpisywane nowym.

4.4.4 Obsługa raportów (Zgłoszeń użytkowników)

Aktorzy: Użytkownik (zgłasza), Administrator (rozpatruje)
Przebieg:

1. Użytkownik A zgłasza Użytkownika B podając powód (np. spam, wulgarne nazwy).
2. Administrator widzi listę raportów w statusie "Pending".
3. Administrator może:
   - **Zaakceptować raport:** Użytkownik B zostaje zbanowany (zgodnie z ustawieniami bana), a status raportu zmienia się na "Zaakceptowany".
   - **Odrzucić raport:** Brak konsekwencji dla Użytkownika B, status raportu "Odrzucony".

5 Ograniczenia i wymagania techniczne

1. **Zdjęcia:** Maksymalny rozmiar pliku to 5MB. Obsługiwane formaty: JPEG, PNG, WEBP.
2. **Sesja:** Token JWT jest ważny przez 3 godziny. Po tym czasie wymagane jest odświeżenie (Refresh Token) lub ponowne logowanie.
3. **Unikalność danych:** W systemie nie mogą istnieć dwaj użytkownicy o tym samym emailu lub nazwie.
4. **Uprawnienia:** Zwykły użytkownik nie ma dostępu do endpointów o ścieżce `/api/admin/*`.

6 Struktura interfejsu (Widoki)

System składa się z następujących głównych ekranów:

- **Strefa Publiczna:**
  - `/home` - Strona startowa.
  - `/map` - Mapa stacji.
  - `/list` - Wyszukiwarka listowa stacji.
  - `/station/...` - Szczegóły konkretnej stacji.
  - `/login, /register` - Ekrany autoryzacji.
- **Strefa Użytkownika:**
  - `/dashboard` - Pulpit użytkownika.
  - `/proposals` - Historia własnych zgłoszeń.
  - `/settings` - Ustawienia konta.
- **Strefa Administratora:**
  - `/admin-dashboard` - Panel główny.
  - `/proposals_admin` - Panel weryfikacji cen (ze zdjęciami).
  - `/gas_station_admin` - Zarządzanie stacjami.
  - `/user_admin` - Zarządzanie użytkownikami (bany, role).
  - `/brand_admin` - Słownik marek paliw.

7 Architektura Systemu

System został zaprojektowany w architekturze mikroserwisowej/kontenerowej z wykorzystaniem Docker Compose. Warstwa prezentacji (Frontend) komunikuje się z warstwą logiki (Backend) poprzez protokół HTTP, a całość infrastruktury jest odseparowana w dedykowanych kontenerach.

7.1 Komponenty Infrastruktury (Docker Services)

System składa się z następujących serwisów:

- **Controllers (API):** Główny kontener backendowy (.NET), udostępniający REST API.
- **Client (Frontend):** Kontener serwujący aplikację React (Node.js/Vite). W środowisku deweloperskim działa w trybie HMR (Hot Module Replacement).
- **PostGIS (Database):** Baza danych PostgreSQL rozszerzona o moduł przestrzenny PostGIS do wydajnego przetwarzania danych geolokalizacyjnych.
- **Redis (Cache):** Szybka baza klucz-wartość wykorzystywana do cache'owania zapytań i przechowywania sesji rozproszonych.
- **Azurite (Blob Storage):** Emulator usługi Azure Blob Storage, służący do przechowywania plików binarnych (zdjęcia pylonów).
- **Nginx (Proxy & Web Server):** Serwer WWW działający jako Reverse Proxy, kierujący ruch do odpowiednich kontenerów oraz serwujący aplikację frontendową w środowisku produkcyjnym.
- **Certbot:** Kontener odpowiedzialny za automatyczne generowanie i odnawianie certyfikatów SSL (Let's Encrypt) w środowisku produkcyjnym.
- **Mailpit (SMTP):** Narzędzie do przechwytywania wiadomości email (wykorzystywane tylko w środowisku deweloperskim).

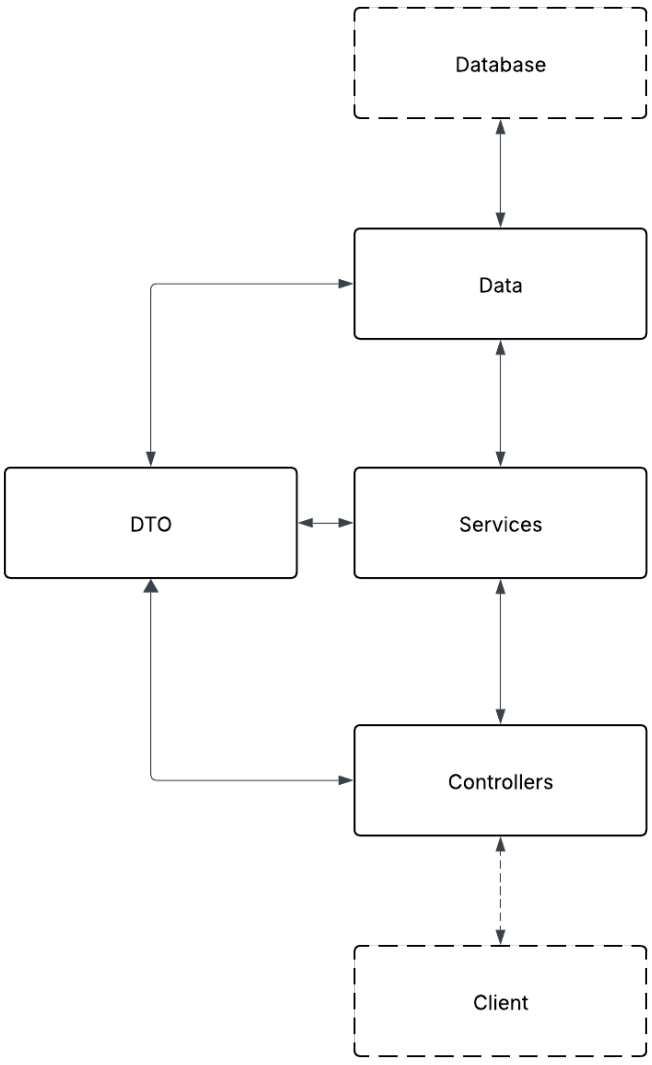
7.2 Architektura Wewnętrzna Backend (N-Layer)

Aplikacja backendowa (serwis `Controllers`) została zaimplementowana w architekturze wielowarstwowej (N-Layer), co zapewnia separację odpowiedzialności (SoC), testowalność i łatwość utrzymania.

1. **Warstwa Prezentacji (Presentation Layer / API):**
   - **Rola:** Obsługa żądań HTTP, walidacja wejściowa, autoryzacja (JWT).
   - **Komponenty:** `Controllers` (np. `StationController`), `Middleware`, `DTOs`.
2. **Warstwa Logiki Biznesowej (Service Layer / BLL):**
   - **Rola:** Przetwarzanie danych, realizacja procesów biznesowych (np. algorytmy punktacji), integracja z zewnętrznymi serwisami.

- **Komponenty:** `Services` (np. `StationService`), `Interfaces`.

3. **Warstwa Dostępu do Danych (Data Access Layer / DAL):**
   - **Rola:** Bezpośrednia komunikacja z bazą danych PostgreSQL, operacje CRUD.
   - **Komponenty:** `DbContext` (Entity Framework Core), `Repositories`.
4. **Warstwa Domeny (Domain Layer):**
   - **Rola:** Definicja kształtu danych w systemie.
   - **Komponenty:** `Entities` (`User`, `Station`), `Enums`.



Rysunek 1: Diagram architektury backendu (miejsce na obraz)

8 Stos Technologiczny (Tech Stack)

8.1 Backend (API)

- **Platforma:** .NET 8.0 (ASP.NET Core Web API).
- **ORM:** Entity Framework Core z providerem PostgreSQL (`Npgsql`).
- **Storage Client:** Azure SDK for .NET (obsługa Blob Storage/Azurite).
- **Cache Client:** StackExchange.Redis.
- **Email Service:** Brevo SMTP (produkcja) / Mailpit (dev).
- **Dokumentacja:** Swagger / OpenAPI 3.0.

8.2 Frontend (Klient)

- **Framework:** React + TypeScript.
- **Build Tool:** Vite.
- **Styling:** Tailwind CSS.
- **Mapy:** Leaflet + OpenStreetMap.

8.3 Infrastruktura i Wersje

- **OS Serwera:** Linux (Ubuntu Server).
- **Konteneryzacja:** Docker, Docker Compose V2.
- **Baza Danych:** PostgreSQL 17 + PostGIS 3.5.
- **Cache:** Redis 8.2.
- **Serwer WWW/Proxy:** Nginx.
- **SSL:** Certbot (Let's Encrypt).

9 Model Danych i Baza Danych

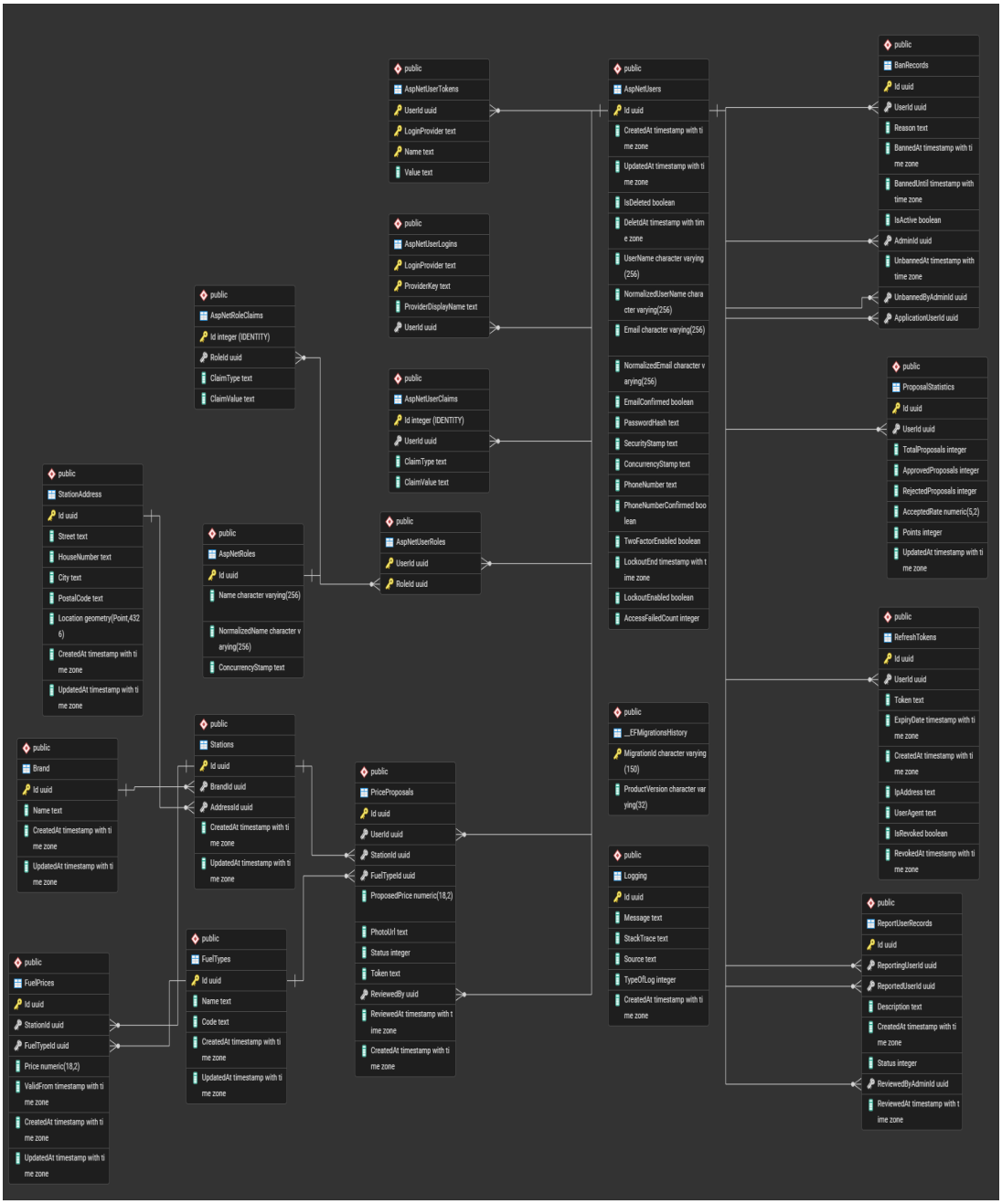
9.1 Geolokalizacja i PostGIS

Wykorzystanie rozszerzenia PostGIS jest kluczowe dla funkcjonalności mapy.

- **Typ danych:** Współrzędne stacji przechowywane są jako typy geometryczne/geograficzne PostGIS (np. `geometry(Point, 4326)`).
- **Indeksowanie:** Zastosowanie indeksów przestrzennych (GIST) dla szybkiego wyszukiwania "w pobliżu"(funkcja `ST_DWithin`).

9.2 Główne Encje

- **Station:** Zawiera atrybuty geograficzne oraz relacje do marki.
- **FuelPrice:** Przechowuje historię cen dla danej stacji.
- **PriceProposal:** Powiązana z systemem plików (Azurite) poprzez identyfikator zdjęcia.
- **User (Identity):** Standardowy model ASP.NET Identity przechowywany w PostgreSQL.



Rysunek 2: Diagram związków encji (ERD) bazy danych

10 Interfejs API i Komunikacja

10.1 Konfiguracja Połączeń

Kontener API komunikuje się z usługami zależnymi za pomocą zmiennych środowiskowych zdefiniowanych w pliku `docker-compose.yml`:

- **Baza danych:** `ConnectionStrings__DefaultConnection`.
- **Redis:** `Redis__Host` oraz `Redis__Port`.
- **Azure Blobs:** `Blob__ConnectionString` (wskazujący na kontener `azurite`).
- **SMTP:** Zmienne z sekcji `Mail__*` (Host, Port, Username, Password, SSL).

10.2 Obsługa plików (Blob Storage)

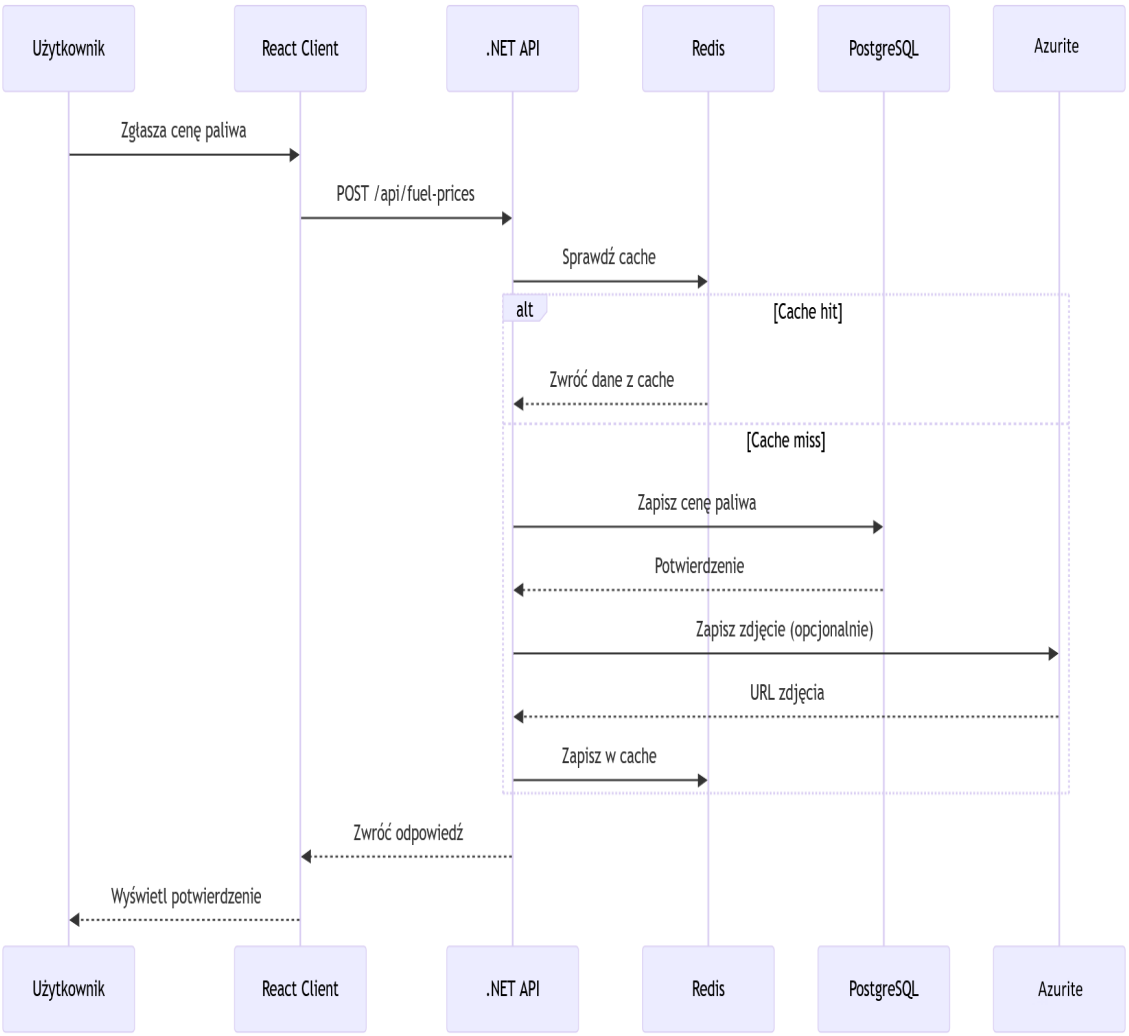
Endpoint `POST /api/station/price-proposal/add`:

- Plik graficzny jest przesyłany do API jako `multipart/form-data`.
- API łączy się z kontenerem Azurite (lub usługą Azure w chmurze) i zapisuje plik.
- System może generować publiczny URL lub SAS Token, aby umożliwić pobranie zdjęcia.

10.3 Cache (Redis)

Redis wykorzystywany jest do:

- Przechowywania wyników kosztownych zapytań (np. lista marek, konfiguracja).
- Przechowywania sesji rozproszonych (Distributed Cache), co pozwala na skalowanie aplikacji (stateless API).



Rysunek 3: Diagram przepływu danych (Dataflow) w systemie

11 Architektura i implementacja Backendu

11.1 Architektura i odpowiedzialność

Backend aplikacji FuelApp stanowi centralny punkt logiki biznesowej systemu. Został zrealizowany w oparciu o platformę .NET 8 w architekturze warstwowej (N-Tier). Jego głównymi zadaniami są przetwarzanie danych, zapewnienie bezpiecznej komunikacji z bazą danych, realizacja złożonych reguł biznesowych oraz udostępnianie interfejsu REST API, który jest konsumowany przez aplikację frontendową.

Architektura projektu promuje ścisłą separację odpowiedzialności (ang. *Separation of Concerns*), co zwiększa czytelność kodu, jego testowalność oraz ułatwia przyszły rozwój i utrzymanie. Projekt został podzielony na główne moduły logiczne (warstwy):

- **API (Warstwa Prezentacji):** Punkt wejścia dla żądań HTTP. Zawiera kontrolery REST, konfigurację potoku middleware (obsługa błędów, CORS, Rate Limiting), mechanizmy uwierzytelniania oraz konfigurację kontenera wstrzykiwania zależności (DI).
- **Services (Warstwa Logiki Biznesowej):** Serce aplikacji implementujące wszystkie reguły biznesowe. Warstwa ta odpowiada za walidację danych wejściowych, koordynację działań między repozytoriami a innymi komponentami infrastruktury (np. cache Redis, system zdarzeń, serwis wysyłki e-mail czy Azure Blob Storage).
- **Data (Warstwa Dostępu do Danych):** Abstrahuje bezpośredni dostęp do bazy danych. Zawiera definicje modeli domenowych (encji), kontekst bazy danych (Entity Framework Core) oraz repozytoria wykonujące bezpośrednie zapytania SQL/LINQ.
- **DTO (Data Transfer Objects):** Proste obiekty służące do przesyłania danych między warstwami oraz definiujące kontrakty żądań i odpowiedzi API, oddzielając modele domenowe od modeli widoku.

11.2 Użyte technologie

Poniżej zestawienie kluczowych technologii i bibliotek wykorzystanych w warstwie backendowej:

- **Platforma:** .NET 8.0 (ASP.NET Core Web API);
- **Baza danych:** PostgreSQL 16+ (z rozszerzeniem PostGIS dla danych geograficznych);
- **ORM (Object-Relational Mapping):** Microsoft.EntityFrameworkCore v. 8.0.20 (wraz z Npgsql.EntityFrameworkCore.PostgreSQL v. 8.0.11 i NetTopologySuite dla obsługi geometrii);
- **Tożsamość i autoryzacja:** ASP.NET Core Identity v. 8.0.20 oraz JWT Bearer Authentication;
- **Caching (rozproszony):** Redis (obsługiwany przez klienta NRedisStack v. 1.1.1);
- **Logowanie:** Serilog v. 4.3.0 (z zapisem strukturalnym do plików, konsoli oraz bazy PostgreSQL poprzez niestandardowy sink);

- **Storage plików:** Azure Blob Storage (biblioteka Azure.Storage.Blobs v. 12.26.0);
- **Zadania w tle:** Hosted Services (klasy dziedziczące po BackgroundService);
- **Dokumentacja API:** Swagger/OpenAPI (Swashbuckle v. 6.6.2).

11.3 Przepływ danych i kluczowe komponenty

Podstawowy przepływ żądania HTTP w aplikacji realizowany jest według schematu:

`Klient HTTP` → `Kontroler (API)` → `Serwis (Logika)` → `Repozytorium (Dane)` → `Baza Danych`

Kontrolery (np. `StationController`) są punktami wejścia API. Ich rola ogranicza się do przyjęcia żądania, wstępnej walidacji modelu (DTO) za pomocą atrybutów DataAnnotations i przekazania sterowania do odpowiedniego serwisu.

Serwisy zawierają właściwą logikę aplikacji. To tutaj podejmowane są decyzje, sprawdzane złożone warunki biznesowe, a także wykorzystywane mechanizmy optymalizacyjne, takie jak Caching (wzorzec *Cache-Aside* implementowany przez `CacheService`) czy Event Dispatcher do asynchronicznego powiadamiania innych części systemu o zmianach.

Repozytoria (np. `StationRepository`) stanowią warstwę abstrakcji nad Entity Framework Core. Dzięki nim logika biznesowa nie operuje bezpośrednio na `DbContext`, co ułatwia testowanie i centralizuje zapytania do bazy. Repozytoria odpowiadają za efektywne pobieranie danych (np. używając projekcji `.Select()`) i zarządzanie transakcjami.

11.3.1 Implementacja warstwy serwisu z wykorzystaniem Cache-Aside

Poniższy fragment kodu (`StationServices.cs`) ukazuje, jak serwis wykorzystuje repozytorium oraz mechanizm cache'owania do pobrania danych stacji. Zastosowano tu wzorzec *Cache-Aside*: najpierw sprawdzana jest pamięć Redis, a dopiero w przypadku braku danych (cache miss) następuje odpytanie bazy danych, a wynik jest zapisywany w cache na przyszłość.

```
public async Task<Result<GetStationInfoForEditResponse>> GetStationInfoForEdit(
    FindStationRequest request)
{
    try
    {
        // Generowanie unikalnego klucza dla cache na podstawie parametrow zadania
        var cacheKey = _cache.GenerateCacheKey(
            $"{CacheService.CacheKeys.StationPrefix}edit:{request.BrandName}:{request.
    City}:{request.Street}:{request.HouseNumber}"
        );

        // Proba pobrania z Redis. Jesli klucz nie istnieje, wykonaj podana funkcje (
    zapytanie do repozytorium),
        // zapisz jej wynik w cache i zwroc dane.
        var result = await _cache.GetOrSetAsync(
            cacheKey,
            async () => await _stationRepository.GetStationInfoForEdit(request), //
    Wywolanie repozytorium
            CacheService.CacheExpiry.Short
        );

        if (result == null)
        {
             // Obsluga sytuacji, gdy stacja nie zostala znaleziona
             _logger.LogWarning("Station not found for editing.");
             return Result<GetStationInfoForEditResponse>.Bad("Station not found.",
    StatusCodes.Status404NotFound, ...);
        }

        // Zwrocenie wyniku opakowanego w ustandaryzowany obiekt Result<T>
        return Result<GetStationInfoForEditResponse>.Good("Station info retrieved
    successfully.", StatusCodes.Status200OK, result);
    }
    catch (Exception ex)
    {
        // Globalna obsluga bledow w serwisie i logowanie wyjatku
        _logger.LogError(ex, $"An error occurred while retrieving station info for edit
    ...");
        return Result<GetStationInfoForEditResponse>.Bad("An error occurred...",
    StatusCodes.Status500InternalServerError, ...);
    }
}
```

Listing 1: Przykład użycia Cache-Aside w serwisie StationServices

11.3.2 Implementacja warstwy repozytorium z projekcją danych

Repozytorium wykonuje rzeczywiste zapytanie do bazy danych. W poniższym przykładzie (`StationRepository.cs`) użyto projekcji (metoda `.Select`), aby pobrać z bazy tylko te kolumny, które są wymagane przez DTO odpowiedzi, co znacząco zwiększa wydajność zapytania poprzez redukcję przesyłanych danych.

```
public async Task<GetStationInfoForEditResponse> GetStationInfoForEdit(FindStationRequest
     request)
    => await _context.Stations
        // Dolaczenie powiazanych encji (Eager Loading)
        .Include(s => s.Brand)
        .Include(s => s.Address)
        .Include(s => s.FuelPrice)
            .ThenInclude(fp => fp.FuelType)
        .Where(s =>
            // Filtrowanie po unikalnych cechach stacji (case-insensitive)
            s.Brand.Name.ToLower() == request.BrandName.ToLower() &&
            s.Address.Street.ToLower() == request.Street.ToLower() &&
            s.Address.HouseNumber.ToLower() == request.HouseNumber.ToLower() &&
            s.Address.City.ToLower() == request.City.ToLower()
        )
        // Projekcja wyniku bezposrednio do obiektu DTO
        .Select(s => new GetStationInfoForEditResponse
        {
            BrandName = s.Brand.Name,
            Street = s.Address.Street,
            HouseNumber = s.Address.HouseNumber,
            City = s.Address.City,
            Latitude = s.Address.Location.Y, // Mapowanie koordynatow z typu
    geometrycznego
            Longitude = s.Address.Location.X,
            FuelType = s.FuelPrice.Select(fp => new FindFuelRequest
            {
                Code = fp.FuelType.Code,
                Price = fp.Price
            }).ToList()
        })
        .FirstOrDefaultAsync();
```

Listing 2: Zapytanie EF Core z projekcją w StationRepository

11.4 Kluczowe mechanizmy backendu

11.4.1 Baza danych i dane geoprzestrzenne

Backend wykorzystuje PostgreSQL jako główny magazyn relacyjny. Kluczowym aspektem projektu jest użycie rozszerzenia **PostGIS** oraz biblioteki `NetTopologySuite` zintegrowanej z Entity Framework Core. Umożliwia to przechowywanie lokalizacji stacji jako natywnych typów geometrycznych (`geometry(Point, 4326)`) i wykonywanie wydajnych zapytań przestrzennych bezpośrednio na poziomie bazy danych – na przykład znajdowanie stacji w określonym promieniu od użytkownika lub sortowanie wyników po dystansie.

11.4.2 Uwierzytelnianie i Autoryzacja

System opiera się na bezstanowym mechanizmie tokenów JWT (JSON Web Token). Wykorzystano wbudowany w ASP.NET Core system Identity do zarządzania użytkownikami, rolami i claimami. Zaimplementowano również autoryzację przez zewnętrznych dostawców (Google, Facebook) oraz system Refresh Tokenów, pozwalający na bezpieczne odświeżanie sesji użytkownika bez konieczności ponownego wpisywania danych logowania.

11.4.3 Caching (Redis)

Aby odciążyć bazę danych przy częstych operacjach odczytu (np. wyświetlanie mapy stacji, list cen), zastosowano pamięć podręczną Redis. Zaimplementowano własny serwis pomocniczy `CacheService`, który ułatwia stosowanie wzorca "Cache-Aside", zarządzanie kluczami oraz inwalidację cache'u. Przykładowo, dodanie nowej stacji lub zmiana ceny paliwa powoduje automatyczne usunięcie powiązanych wpisów z cache (np. list stacji dla danego miasta), wymuszając ich odświeżenie przy kolejnym żądaniu.

11.4.4 Przetwarzanie w tle i architektura zdarzeń (Event-Driven)

Aplikacja wykorzystuje mechanizm `IHostedService` (Background Services) do wykonywania zadań cyklicznych w tle, niezależnie od żądań HTTP. Przykłady obejmują `BanExpirationService` (automatyczne odblokowywanie użytkowników po wygaśnięciu bana) czy `ProposalExpirationService` (odrzucanie starych propozycji cenowych).

Dodatkowo zastosowano wewnętrzny **Event Dispatcher** (implementacja wzorca Mediator). Pozwala on na asynchroniczną reakcję na zdarzenia w systemie w celu zachowania niskiego sprzężenia komponentów. Na przykład, zdarzenie rejestracji użytkownika (`UserRegisteredEvent`) jest publikowane przez serwis użytkowników, a niezależny handler (`SendRegistrationEmailHandler`) odbiera je i kolejkuje wiadomość powitalną, nie blokując głównego wątku obsługi rejestracji.

11.4.5 Integracje zewnętrzne i kolejkowanie

Backend integruje się z usługą Azure Blob Storage (w środowisku deweloperskim emulowaną przez Azurite) w celu przechowywania plików binarnych, takich jak zdjęcia weryfikacyjne stacji. Wysyłka wiadomości e-mail (SMTP) jest realizowana asynchronicznie poprzez wbudowaną kolejkę w pamięci (`InMemoryEmailQueue`) i przetwarzana przez dedykowany wątek roboczy w tle (`EmailBackgroundWorker`), co zapewnia wysoką responsywność API nawet przy problemach z zewnętrznym serwerem pocztowym.

12 Architektura i implementacja Frontendu

12.1 Architektura i odpowiedzialność

Warstwa kliencka aplikacji FuelApp została zrealizowana jako nowoczesna, dynamiczna aplikacja jednostronicowa (ang. *Single Page Application* - SPA). Działa ona całkowicie niezależnie od warstwy backendowej, komunikując się z nią wyłącznie za pośrednictwem bezstanowego interfejsu REST API.

Główną odpowiedzialnością frontendu jest zapewnienie responsywnego i intuicyjnego interfejsu użytkownika (UI/UX), wizualizacja danych geograficznych na interaktywnej mapie, obsługa formularzy (logowanie, dodawanie propozycji cen) oraz zarządzanie stanem aplikacji po stronie klienta. Architektura oparta jest na komponentach, co pozwala na wielokrotne użycie elementów interfejsu (np. nagłówek, stopka, komponent mapy) i ułatwia utrzymanie kodu.

12.2 Użyte technologie

Stos technologiczny warstwy frontendowej został dobrany pod kątem wydajności, bezpieczeństwa typowania oraz szybkości developmentu. Kluczowe zależności zdefiniowane w pliku `package.json`:

- **Framework i Routing:** React 19 wraz z React Router v7 (biblioteki `react`, `react-dom`, `@react-router/*`) - do budowania interfejsu użytkownika i zarządzania nawigacją w aplikacji SPA.
- **Build Tool:** Vite 7 (narzędzie do szybkiego budowania i serwowania aplikacji w środowisku deweloperskim z obsługą HMR - Hot Module Replacement).
- **Język:** TypeScript (nadzbiór JavaScript dodający statyczne typowanie, co znacząco redukuje ilość błędów).
- **Styling:** Tailwind CSS v4 wraz z pluginem daisyUI (framework CSS typu *utility-first* przyspieszający tworzenie responsywnych i estetycznych widoków).
- **Mapy:** Leaflet v1.9 (lekka biblioteka do obsługi interaktywnych map OpenStreetMap) zintegrowana z Reactem.
- **Internacjonalizacja (i18n):** `i18next` i `react-i18next` (do obsługi wielojęzyczności aplikacji - polski i angielski).
- **Wykresy:** Recharts (do wizualizacji danych statystycznych).
- **Komunikacja HTTP:** Natywny `fetch` API (do komunikacji z backendem, z obsługą ciasteczek `credentials: "include"`).

12.3 Kluczowe koncepcje i implementacja

12.3.1 Routing i struktura aplikacji

Aplikacja wykorzystuje najnowszą wersję React Router v7, gdzie definicja ścieżek znajduje się w pliku `routes.ts`. Struktura oparta jest na systemie plików, co upraszcza zarządzanie widokami. Główny komponent layoutu, `root.tsx`, definiuje wspólną strukturę HTML, nagłówki, style (w tym globalny `app.css` z dyrektywami Tailwind) oraz dostawcę kontekstu motywu (`ThemeProvider`).

```
import { type RouteConfig, index, route } from "@react-router/dev/routes";

export default [
  index("routes/home.tsx"), // Strona glowna
  route("login", "routes/login.tsx"),
  route("dashboard", "routes/dashboard.tsx"),
  // Sciezki administracyjne
  route("admin", "routes/admin-dashboard.tsx"),
  route("admin/stations", "routes/gas-station-admin.tsx"),
  // Sciezka z parametrami (szczegoly stacji)
  route("station/:brandName/:city/:street/:houseNumber", "routes/station.tsx"),
  // ... pozostale trasy
] satisfies RouteConfig;
```

Listing 3: Przykładowa definicja ścieżek w routes.ts

12.3.2 Zabezpieczanie widoków (Custom Hooks)

Dostęp do części aplikacji wymagających autoryzacji (np. pulpit użytkownika, panel administratora) jest kontrolowany za pomocą własnych hooków (Custom Hooks). Zamiast tradycyjnych komponentów opakowujących, zastosowano podejście oparte na hookach `useUserGuard` i `useAdminGuard`, które są wywoływane na początku komponentu strony.

Hooki te asynchronicznie odpytują endpoint `/api/me` w celu weryfikacji bieżącej sesji użytkownika (przesyłając ciasteczko `HttpOnly` z tokenem JWT). W zależności od odpowiedzi serwera i roli użytkownika, hook zwraca stan (`checking`, `allowed`, `redirected`) i w razie potrzeby automatycznie przekierowuje użytkownika (np. na stronę logowania).

```
export function useAdminGuard() {
  const [state, setState] = React.useState<AdminGuardState>("checking");
  // ...

  React.useEffect(() => {
    (async () => {
      try {
        const me = await fetchMe(); // Zapytanie do /api/me z credentials: "include"

        if (!me) {
          // Brak sesji -> przekierowanie do logowania
          window.location.href = "/login";
          setState("redirected");
          return;
        }

        // Normalizacja i sprawdzenie roli
        const role = extractRoleLoose(me);
        if (role !== "Admin") {
          // Zalogowany, ale brak uprawnien admina -> przekierowanie do dashboardu
          window.location.href = "/dashboard";
          setState("redirected");
          return;
        }

        setState("allowed"); // Dostep przyznany
      } catch (err) {
        // Obsluga bledow...
      }
    })();
  }, []);

  return { state, email };
}
```

Listing 4: Fragment hooka useAdminGuard.ts zabezpieczającego panel administratora

Użycie tego hooka w komponencie widoku (`admin-dashboard.tsx`) jest proste i deklaratywne:

```
export default function AdminDashboard() {
  // ...
  const { state, email } = useAdminGuard();

  if (state === "checking") {
    // Wyswietlenie spinnera podczas weryfikacji uprawnien
    return (/* ... */ <span className="loading loading-spinner loading-lg" /> /* ... */);
  }

  if (state !== "allowed") {
    // Jesli dostep nie jest przyznany, nie renderuj nic (hook juz przekierowal)
    return null;
  }

  // Renderowanie wlasciwego panelu administratora
  return ( /* ... */ );
}
```

12.3.3 Integracja z mapami (Leaflet)

Serce aplikacji stanowi interaktywna mapa, zaimplementowana w komponencie `GlobalMapContent.tsx`. Wykorzystuje ona bibliotekę Leaflet w sposób imperatywny, inicjalizując mapę i warstwę markerów wewnątrz hooka `useEffect`. Komponent ten odpowiada za:

- Inicjalizację mapy z kafelkami OpenStreetMap.
- Konfigurację domyślnych ikon markerów.
- Dynamiczne aktualizowanie markerów na podstawie przekazanej propsami listy stacji (`stations`).
- Kolorowanie markerów w zależności od marki stacji (np. Orlen na czerwono, BP na zielono).
- Tworzenie interaktywnych dymków (popupów) z informacjami o stacji i linkiem do jej szczegółów.

12.3.4 Zarządzanie stanem globalnym (Motyw i Język)

Aplikacja wykorzystuje React Context API do zarządzania globalnym stanem motywu (jasny/ciemny). Komponent `ThemeProvider` (w `ThemeContext.tsx`) przechowuje aktualny motyw w stanie lokalnym oraz w `localStorage`, a także aplikuje odpowiedni atrybut `data-theme` na elemencie `<html>`, co jest wykorzystywane przez Tailwind CSS i daisyUI.

Obsługa wielojęzyczności (i18n) jest realizowana za pomocą biblioteki `i18next`. Konfiguracja w pliku `i18n.ts` obejmuje wykrywanie języka przeglądarki, ładowanie tłumaczeń z plików JSON (`i18next-http-backend`) oraz integrację z Reactem. Zmiana języka w nagłówku aplikacji natychmiastowo aktualizuje wszystkie teksty w interfejsie.

13 Weryfikacja i Testowanie Systemu

Projekt zawiera kompleksowe testy weryfikujące poprawność działania logiki biznesowej aplikacji na najniższym poziomie (testy jednostkowe) oraz komunikację pomiędzy front-endem a back-endem (testy integracyjne API). Umożliwiają one wczesne wykrywanie błędów, ułatwiają rozwój projektu oraz zwiększają jego stabilność.

Testy zostały podzielone na:

- testy metod repozytoriów,
- testy metod serwisów,
- testy endpointów kontrolerów.

13.1 Technologie użyte w testach

- **Język testów:** C#
- **Framework testów back-endu:** xUnit v. 2.5.3
- **Mocking:** Moq v. 4.20.72
- **Dodatkowe biblioteki:**
  - Microsoft.EntityFrameworkCore.InMemory v. 8.0.20
  - Microsoft.AspNetCore.Mvc.Testing v. 8.0.22

13.2 Testy jednostkowe (Unit Tests)

Testy jednostkowe opierają się na trzech filarach:

1. **Framework xUnit** - udostępnia narzędzia do strukturyzacji testów i ich uruchamiania.
2. **Biblioteka InMemory** - pozwala na tworzenie bazy danych w pamięci RAM.
3. **Moq** - umożliwia tworzenie sztucznych obiektów (mocków), definiowanie ich zachowania oraz weryfikację interakcji.

W testach jednostkowych weryfikowane są pojedyncze metody repozytoriów i serwisów. Aby wykonać test, należy najpierw stworzyć odpowiednie środowisko testowe, używając wyżej wymienionych narzędzi. Tworzone jest repozytorium korzystające z bazy danych w pamięci, a w miejsca zależności, na których testowaniu nam nie zależy (np. Logger, UserManager), podstawiane są mocki. Daje to pełną kontrolę nad zawartością bazy danych i pozwala precyzyjnie sprawdzić, czy testowana metoda poprawnie wykonuje przewidziane zadanie.

13.2.1 Przykładowa konfiguracja środowiska testowego

```
public class UserRepositoryTest
{
    private readonly ApplicationDbContext _context; // baza danych in-memory
    private readonly Mock<ILogger<UserRepository>> _loggerMock; // mock loggera
    private readonly UserRepository _repository;
    private readonly ITestOutputHelper _output; // output do konsoli xUnit
    private readonly Mock<UserManager<ApplicationUser>> _userManagerMock; // mock UM

    public UserRepositoryTest(ITestOutputHelper output)
    {
        _output = output;
        var options = new DbContextOptionsBuilder<ApplicationDbContext>()
            .UseInMemoryDatabase(Guid.NewGuid().ToString())
            .ConfigureWarnings(w => w.Ignore(Microsoft.EntityFrameworkCore.Diagnostics.
    InMemoryEventId.TransactionIgnoredWarning)).Options;
        _context = new ApplicationDbContext(options);
        _loggerMock = new Mock<Ilogger<UserRepository>>();
        var _userMock = new Mock<IUserStore<ApplicationUser>>();
        _userManagerMock = new Mock<UserManager<ApplicationUser>>(_userMock.Object, null
    !, null!, null!, null!, null!, null!, null!, null!);
        _repository = new UserRepository(_context, _loggerMock.Object, _userManagerMock.
    Object);
    }
}
```

Listing 5: Konfiguracja UserRepositoryTest

13.2.2 Struktura testu (AAA)

Każda metoda testowa, oznaczona atrybutem `[Fact]`, zbudowana jest według wzorca Arrange-Act-Assert:

- **Arrange:** Operacje przygotowawcze, np. seedowanie bazy danych danymi testowymi.
- **Act:** Wywołanie testowanej metody.
- **Assert:** Weryfikacja wyników operacji.

```
[Fact]
public async Task DeleteUserAsyncTest_SuccessIfUserDeleted()
{
    //Arrange
    var user1 = new ApplicationUser
    {
        Id = Guid.NewGuid(),
        Email = "user1@test.com",
        UserName = "user1",
        CreatedAt = DateTime.UtcNow.AddDays(-1),
        IsDeleted = false
    };
    _context.Users.Add(user1);
    await _context.SaveChangesAsync();

    //Act
    var result = await _repository.DeleteUserAsync(user1);

    //Assert
    var userUpdate = await _context.Users.FirstAsync();
    Assert.True(result.Succeeded);
    Assert.Equal(user1.Id, userUpdate.Id);
    Assert.True(userUpdate.IsDeleted);
    Assert.NotNull(userUpdate.DeletdAt);
    _output.WriteLine("Success, DeleteUserAsync changes flags in a deleted user correctly
    ");
}
```

Listing 6: Przykładowy test jednostkowy metody DeleteUserAsync

Powyższy test weryfikuje, czy metoda `DeleteUserAsync` poprawnie oznacza użytkownika jako usuniętego (soft delete) zamiast fizycznie usuwać go z bazy. Metody są testowane pod kątem wielu przypadków brzegowych (np. dla istniejącego i nieistniejącego użytkownika).

13.3 Testy integracyjne API

Testy API polegają na weryfikacji działania pojedynczych endpointów kontrolerów. Aby uniknąć korzystania z prawdziwego, zewnętrznego serwera, wykorzystana została biblioteka `Microsoft.AspNetCore.Mvc.Testing`.

Pozwala ona na stworzenie serwera oraz klienta HTTP w pamięci, dzięki utworzeniu fabryki niestandardowej `WebApplicationFactory`. Umożliwia to wysyłanie żądań na endpointy aplikacji. Następnie weryfikowana jest odpowiedź serwera poprzez sprawdzenie zwróconego kodu statusu HTTP oraz ewentualne sprawdzenie, czy zadanie zostało wykonane (poprzez dostęp do bazy danych in-memory lub odczytanie danych zwrotnych w formacie JSON).

Podejście to pozwala na tworzenie nowej, odizolowanej instancji serwera i zbioru danych dla każdego testu osobno, dzięki czemu unikamy problemów takich jak przypadkowe usunięcie lub modyfikacja danych przez inne testy.

13.3.1 Przykładowa konfiguracja klienta testowego

```
public class ProposalStatisticControllerTest : IAsyncLifetime
{
    private HttpClient _client;
    private CustomAppFact _factory;

    public async Task InitializeAsync()
    {
        _factory = new CustomAppFact();
        _client = _factory.CreateClient();
        // Symulacja uwierzytelnionego użytkownika
        _client.DefaultRequestHeaders.Authorization =
            new AuthenticationHeaderValue("Bearer", "test-user-token");
        await Task.CompletedTask;
    }

    public async Task DisposeAsync()
    {
        _client?.Dispose();
        await _factory.DisposeAsync();
    }
}
```

Listing 7: Konfiguracja ProposalStatisticControllerTest

13.3.2 Przykładowy test endpointu

```
[Fact]
public async Task GetTopUserListAsyncTest_400()
{
    //Arrange
    // Nieprawidłowe parametry paginacji
    var requestUri = "/api/proposal-statistic/top-users?PageNumber=0&PageSize=0";

    // Act
    var response = await _client.GetAsync(requestUri);

    //Assert
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}
```

Listing 8: Test weryfikujący błędne żądanie (Bad Request)

Każdy z endpointów testowany jest w wielu ścieżkach wykonania, w zależności od ilości przewidzianych odpowiedzi. Oznacza to, że weryfikowane są ścieżki zwracające statusy takie jak 200 OK, 404 Not Found, 400 Bad Request, 401 Unauthorized itd.

14 Bezpieczeństwo

- **Uwierzytelnianie:** JWT (Json Web Token) + Refresh Token (przechowywany w bezpiecznym ciasteczku HTTP-only).
- **OAuth:** Integracja z Facebookiem i Google (sekrety aplikacji przekazywane bezpiecznie w zmiennych środowiskowych).
- **Zarządzanie sekretami:** W środowisku produkcyjnym sekrety (hasła DB, klucze API, SMTP) są przechowywane w pliku `.env` na serwerze i wstrzykiwane do kontenerów. Plik `.env` jest wykluczony z repozytorium.
- **Izolacja sieci:** Wszystkie serwisy backendowe komunikują się wewnątrz prywatnej sieci Dockerowej `dev` (driver: bridge). Jedynym punktem wejścia z zewnątrz są porty serwera Nginx (80/443).
- **SSL/TLS:** Cały ruch produkcyjny jest szyfrowany za pomocą certyfikatów Let's Encrypt, zarządzanych automatycznie przez kontener Certbot.
- **Rate Limiting:** API posiada wbudowane mechanizmy ograniczania liczby żądań (np. dla endpointów logowania), chroniące przed atakami typu brute-force.

15 Instrukcja lokalnego uruchomienia systemu (Środowisko deweloperskie)

Poniższa instrukcja opisuje proces uruchomienia pełnego środowiska deweloperskiego aplikacji FuelApp na lokalnej maszynie przy użyciu technologii Docker.

Wymagania wstępne:

- Zainstalowany i działający Docker oraz Docker Compose.
- Zainstalowany klient Git.
- Dostęp do internetu w celu pobrania obrazów kontenerów.

15.1 Krok 1: Pobranie repozytorium

Rozpocznij od sklonowania repozytorium projektu na swój lokalny komputer. Otwórz terminal i wykonaj następujące polecenia:

```
# Inicjalizacja pustego repozytorium git (opcjonalnie, jesli nie klonujesz bezposrednio)
git init

# Dodanie zdalnego repozytorium
git remote add origin https://github.com/mateusz-bogacz-collegiumwitelona/fuel

# Pobranie metadanych ze zdalnego repozytorium
git fetch

# Przelaczenie na galaz glowna (main), ktora zawiera stabilna wersje deweloperska
git switch main
```

15.2 Krok 2: Konfiguracja zmiennych środowiskowych (Główny katalog)

W głównym katalogu projektu znajduje się plik wzorcowy `.env.example`. Należy stworzyć na jego podstawie plik `.env`, który będzie zawierał lokalną konfigurację dla kontenerów Dockera.

```
# Kopiowanie pliku przykladowego do wlasciwego pliku konfiguracyjnego
cp .env.example .env
```

Następnie otwórz nowo utworzony plik `.env` w dowolnym edytorze tekstu. Poniżej znajduje się domyślna konfiguracja dla środowiska deweloperskiego. Upewnij się, że porty nie są zajęte przez inne usługi na Twoim komputerze.

Aby uruchomić logowanie przez Facebooka i Google, musisz uzupełnić sekcje `#facebook oauth` oraz `#google oauth` swoimi sekretami aplikacji (uzyskanymi odpowiednio z Meta for Developers i Google Cloud Console). Jeśli nie posiadasz tych danych, funkcjonalność logowania społecznościowego nie będzie działać, ale reszta aplikacji uruchomi się poprawnie.

```
# .net API
API_PORT=5111

# postgres/postgis configuration
POSTGRES_USER=user
POSTGRES_PASSWORD=pass
POSTGRES_DB=database
POSTGRES_PORT=5432
POSTGRES_HOST=postgis

# mailpit (SMTP testing tool)
MAILPIT_PORT=63854

# redis cache
REDIS_HOST=redis
REDIS_PORT=6379

# nginx proxy
NGINX_HTTP_PORT=8080
NGINX_HTTPS_PORT=443

# client frontend
CLIENT_PORT=4000

# azurite (Azure Blob Storage emulator)
AZURITE_BLOB_PORT=10000
AZURITE_QUEUE_PORT=10001
AZURITE_TABLE_PORT=10002

# facebook oauth (UZUPELNIJ WLASNYMI DANYMI)
FACEBOOK_OAUTH_CLIENT_ID=TwojFacebookClientId
FACEBOOK_OAUTH_CLIENT_SECRET=TwojFacebookClientSecret

# google oauth (UZUPELNIJ WLASNYMI DANYMI)
GOOGLE_CLIENT_ID=TwojGoogleClientId
GOOGLE_CLIENT_SECRET=TwojGoogleClientSecret
```

Listing 9: Zawartość głównego pliku .env

15.3 Krok 3: Konfiguracja zmiennych środowiskowych (Frontend)

Aplikacja frontendowa (React) również wymaga osobnej konfiguracji dla mechanizmów OAuth. Przejdź do katalogu `Client` i utwórz tam plik `.env` na podstawie dostarczonego przykładu.

```
cd Client
cp .env.example .env
```

Edytuj plik `Client/.env` i wpisz te same identyfikatory klientów (Client ID), których użyłeś w głównym pliku `.env`.

```
VITE_FACEBOOK_CLIENT_ID=TwojFacebookClientId
VITE_GOOGLE_CLIENT_ID=TwojGoogleClientId
```

Listing 10: Zawartość pliku Client/.env

15.4 Krok 4: Budowanie i uruchomienie aplikacji

Po poprawnej konfiguracji zmiennych środowiskowych wróć do głównego katalogu projektu i uruchom środowisko za pomocą Docker Compose. Proces ten może potrwać kilka minut, ponieważ system musi pobrać obrazy bazowe, zbudować aplikację backendową i frontendową oraz przeprowadzić inicjalne migracje bazy danych.

```
# Powrot do glownego katalogu
cd ..

# Budowanie obrazow kontenerow
docker compose build

# Uruchomienie srodowiska w trybie detached (w tle)
docker compose up -d
```

Po zakończeniu procesu, aplikacja będzie dostępna pod adresem:

`http://localhost:8080`

Dodatkowo, narzędzie do podglądu wiadomości e-mail (Mailpit) będzie dostępne pod adresem `http://localhost:63854`.

16 Instrukcja zdalnego wdrożenia systemu na serwerze VPS

Poniższa instrukcja zakłada, że serwer VPS spełnia wymagania wstępne (OS Linux, Docker, domena wskazująca na IP serwera, dostęp do Brevo SMTP).

16.1 Krok 1: Przygotowanie repozytorium na serwerze

Zaloguj się na serwer VPS i sklonuj repozytorium projektu, przełączając się na gałąź produkcyjną.

```
git clone git@github.com:mateusz-bogacz-collegiumwitelona/fuel.git project
cd project
git switch production
# W przypadku aktualizacji istniejącego wdrożenia:
# git pull
```

16.2 Krok 2: Konfiguracja pliku .env

Utwórz plik `.env` w głównym katalogu projektu na serwerze. Skopiuj do niego zawartość szablonu i uzupełnij własnymi danymi (hasła, klucze SMTP, domena).

```
nano .env
```

Przykładowy szablon pliku .env:

```
#.net
API_PORT=5111
ASPNETCORE_ENVIRONMENT=Production
ADMIN_EMAIL=ADMIN_EMAIL
ADMIN_PASSWORD=ADMIN_PASSWORD

#postgres/postgis and redis
POSTGRES_USER=user
POSTGRES_PASSWORD=pass
POSTGRES_DB=database
POSTGRES_PORT=5432
POSTGRES_HOST=postgis
REDIS_HOST=redis
REDIS_PORT=6379

#frontend
NGINX_HTTP_PORT=80
NGINX_HTTPS_PORT=443
CLIENT_PORT=4000
FRONTEND_PUBLIC_URL=URL_FORM_CLIENT_APP

#azurite
AZURITE_BLOB_PORT=10000
AZURITE_QUEUE_PORT=10001
AZURITE_TABLE_PORT=10002

#social media
FACEBOOK_OAUTH_CLIENT_ID=FACEBOOK_OAUTH_CLIENT_ID
FACEBOOK_OAUTH_CLIENT_SECRET=FACEBOOK_OAUTH_CLIENT_SECRET
GOOGLE_CLIENT_ID=GOOGLE_CLIENT_ID
GOOGLE_CLIENT_SECRET=GOOGLE_CLIENT_SECRET

#Brevo
MAIL_HOST=smtp-relay.brevo.com
MAIL_PORT=587
MAIL_USERNAME=USERNAME_BREVO
MAIL_PASSWORD=PASSWORD_BREVO
MAIL_FROM=EMAIL_FROM_BREVO
MAIL_DISPLAY_NAME=DISPLAY_NAME_BREVO
MAIL_ENABLE_SSL=true
```

W folderze Client też należy utworzyć plik .env i skopjować do niego zawartość szablonu i uzupełnić własnymi danymi

```
nano .env
```

Przykładowy szablon pliku .env:

```
VITE_FACEBOOK_CLIENT_ID=FACEBOOK_OAUTH_CLIENT_ID
VITE_GOOGLE_CLIENT_ID=GOOGLE_CLIENT_ID
```

16.3 Krok 3: Konfiguracja domeny w Nginx

Edytuj plik konfiguracyjny Nginx, aby wpisać swoją domenę.

```
cd templates
nano default.conf
```

Zamień w pliku wszystkie wystąpienia `TWOJA_DOMENA.PL` (oraz `www.TWOJA_DOMENA.PL`) na swoją rzeczywistą domenę.

16.4 Krok 4: Uruchomienie aplikacji (HTTP)

Zbuduj i uruchom kontenery. Na tym etapie Nginx wystartuje na porcie 80.

```
# Wróć do głównego katalogu
cd ~/project
docker compose build
docker compose up -d
```

16.5 Krok 5: Generowanie certyfikatów SSL

Użyj jednorazowego kontenera Certbot do wygenerowania certyfikatów Let's Encrypt.

Ważne: Zamień `TWOJA_DOMENA.PL` na swoją domenę, a `EMAIL@ADMIN.PL` na swój adres email.

```
docker compose run --rm certbot certonly \
  --webroot \
  --webroot-path /var/www/certbot \
  -d TWOJA_DOMENA.PL \
  -d www.TWOJA_DOMENA.PL \
  --email EMAIL@ADMIN.PL \
  --agree-tos \
  --no-eff-email \
  --force-renewal
```

16.6 Krok 6: Uruchomienie HTTPS

Zrestartuj kontener Nginx, aby załadować nowe certyfikaty.

```
docker compose restart web
```

Aplikacja jest teraz dostępna pod bezpiecznym adresem `https://TWOJA_DOMENA.PL`.

17 Wnioski projektowe

17.1 Wnioski członka zespołu: Michał Nocuń (Tester)

Projekt nauczył mnie działania w zespole. Bardzo dobrze współpracowało mi się z moją grupą, każdy był komunikatywny, sumienny, gotowy na dialog, otwarty na zmiany czy nowe sugestie oraz pomocny. Jako tester nabyłem wiele umiejętności twardych. Zaznajomiłem się z typami testów oprogramowania, poznałem frameworki do testów jednostkowych oraz integracyjnych. Praca testera uświadomiła mi, jak ważną rolę pełnią testy oprogramowania w procesie deweloperskim. Dzięki testom jesteśmy w stanie wcześnie wyłapać błędy i poprawić je, zanim kod zostanie bardziej rozbudowany lub co gorsza trafi do produkcji. Uważam, że projekt, nad którym pracowaliśmy, jest bardzo ciekawy. Nasza aplikacja internetowa jest bardzo użyteczna dla wszystkich kierowców. Istnieje też wiele opcji rozbudowy w przyszłości – na przykład dodanie do mapy lokalizacji mechaników, których użytkownicy mogą oceniać i komentować, numery kontaktowe do ubezpieczycieli czy szablony dokumentów, jak na przykład protokół szkody.

17.2 Wnioski członka zespołu: Szymon Mikołajek (Tester / Project Manager)

Podczas realizacji projektu zapoznałem się z podstawami tworzenia testów jednostkowych z wykorzystaniem frameworka xUnit oraz zrozumiałem, jak istotną rolę odgrywają one w zapewnieniu poprawnej funkcjonalności aplikacji. Nabyłem również podstawową wiedzę z zakresu pisania testów frontendowych przy użyciu narzędzia Selenium, które pozwalają automatycznie weryfikować poprawność interakcji użytkownika z aplikacją. Testy te umożliwiają wykrywanie błędów w UI, co bezpośrednio przekłada się na poprawę doświadczenia użytkownika.

17.3 Wnioski członka zespołu: Paweł Kruk (Frontend / DevOps)

Realizacja projektu pozwoliła na praktyczne opanowanie podstaw biblioteki React. Zastosowanie modelu Single Page Application zapewniło płynność działania interfejsu bez konieczności przeładowywania strony. Problemy z wydajnością gotowych wrapperów (np. react-leaflet) rozwiązano poprzez natywną implementację biblioteki Leaflet. Kluczowe okazały się hooki: useRef umożliwił stabilny dostęp do elementu DOM mapy bez zbędnych przerenderowań.

17.4 Wnioski członka zespołu: Mateusz Chimkowski (Frontend / UX / UI Designer)

Projekt pozwolił mi znacząco rozwinąć umiejętności pracy zespołowej i techniczne związane z tworzeniem frontendu. W praktyce pracowałem z Vite + React, poznałem podstawy Tailwind/DaisyUI, poprawiłem umiejętność pisania zapytań do API oraz ulepszyłem strukturę i optymalizację plików projektu. Zespół był dobrze zorganizowany - spotkania były regularne, komunikacja sprawna, a nowy członek szybko został włączony i wsparty przez resztę grupy. Projekt poszedł lepiej niż w zeszłym roku, nauczyłem się kilku nowych rzeczy i jestem zadowolony z efektu.

17.5 Wnioski członka zespołu: Mateusz Bogacz-Drewniak (Team Leader / Backend / DevOps wsparcie)

Pełnienie roli Team Leadera oraz głównego architekta backendu wymagało ode mnie spojrzenia na projekt w sposób holistyczny. Moim priorytetem było zaprojektowanie skalowalnej i łatwej w utrzymaniu architektury (N-Tier w .NET 8), która umożliwiłaby bezkonfliktową, równoległą pracę zespołów frontendowego i backendowego. Decyzja o ścisłej separacji warstw i wykorzystaniu DTO okazała się kluczowa dla stabilności kontraktów API.

Od strony technicznej największym wyzwaniem, a zarazem sukcesem, było wdrożenie zaawansowanych mechanizmów infrastrukturalnych. Zintegrowałem bazę PostgreSQL z rozszerzeniem PostGIS, co umożliwiło wydajne wykonywanie zapytań geoprzestrzennych – serca naszej aplikacji. Zaimplementowałem również caching rozproszony w oparciu o Redis (wzorzec Cache-Aside) oraz wprowadziłem architekturę sterowaną zdarzeniami (Event-Driven z wykorzystaniem wzorca Mediator) do obsługi procesów w tle, co znacząco odciążyło główny wątek aplikacji.

W obszarze DevOps przygotowałem kompleksową konfigurację Docker Compose, która zautomatyzowała uruchamianie całego stosu technologicznego (w tym emulatora Azure Blob Storage, bazy danych i serwera Nginx z obsługą SSL), co drastycznie przyspieszyło proces onboardingu nowych członków zespołu i ujednoliciło środowiska deweloperskie. Projekt był dla mnie cennym doświadczeniem w godzeniu zaawansowanych zadań technicznych z odpowiedzialnością za koordynację pracy zespołu.