Gdańsk, czerwiec 2025

PromoComparer

Jakub Kapica

Julia Kusy

Wiktoria Wiśniewska

Spis treści

1. Architektura systemu 5
2. Funkcjonalności 6
2.1. Scrapowanie gazetek promocyjnych 6
2.2. Konwersja PDF do formatu graficznego 6
2.3. Analiza treści obrazu z wykorzystaniem OpenAI 7
2.4. Zarządzanie promocjami i ich filtrowanie 7
2.5. Zarządzanie strukturą sklepów i kategorii 8
2.6. Personalizacja użytkownika – ulubione promocje 8
2.7. Automatyzacja zadań cyklicznych 8
3. Konfiguracja i środowisko uruchomieniowe 8
3.1. Inicjalizacja hosta i wczytanie konfiguracji 9
3.2. Rejestracja usług i zależności (Dependency Injection) 9
3.3. Konfiguracja CORS, autoryzacji i tożsamości 10
3.3.1. CORS 10
3.3.2. Uwierzytelnianie i autoryzacja 11
3.3.3. Rejestracja ról i użytkowników 12
3.4. Budowanie aplikacji i konfiguracja pipeline'u HTTP 12
3.5. Tworzenie ról przy starcie aplikacji 13
4. Przepływ danych 14
4.1. Pozyskanie danych źródłowych 14
4.2. Konwersja plików PDF do obrazów 14
4.3. Analiza semantyczna obrazu (OpenAI) 15
4.4. Transformacja danych do modeli i zapis w bazie 17
4.5. Udostępnianie danych przez API 17
4.6. Personalizacja i interakcje użytkownika 17
5. Struktura bazy danych 18
5.1. Tabela Store – Sklep 18
5.2. Tabela Leaflet – Gazetka promocyjna 18
5.3. Tabela Promotion – Promocja 19
5.4. Tabela Category – Kategoria 19
5.5. Tabela Favourite – Ulubiona promocja 20

5.6. Tabela User – Użytkownik systemu 20
6. Procedury przetwarzania danych 21
6.1. GetPromotionsByStore 21
6.2. GetTop10LargestPromotions 22
6.3. GetPromotionsByCategory 22
7. Kontrolery 23
7.1. CategoryController 23
7.2. LeafletController 24
7.3. PromotionController 25
7.4. StoreController 26
7.5. OpenAIController 27
7.6. ScrapingController 27
7.7. UserPanelController 28
8. Usługi 30
8.1. CategoryService 30
8.2. LeafletService 31
8.3. PromotionService 31
8.4. StoreService 33
8.5. PdfHandlerService 34
8.6. OpenAIService 35
8.7. Scheduler 35
8.8. UserPanelService 36
9. Wprowadzenie do aplikacji klienckiej PromoComparer 38
9.1. Opis aplikacji 38
9.2. Technologie wykorzystane w projekcie 38
9.3. Podstawowe funkcjonalności 38
9.4. Integracja z backendem 39
9.5. Struktura projektu 39
10. Architektura aplikacji frontendowej 40
10.1. Podział na warstwy 40
10.2. Przepływ danych 41
10.3. Zasady projektowe 41

11. Warstwa domenowa 42
11.1. Struktura modeli domenowych 42
11.2. Rola modeli domenowych 42
12. Warstwa aplikacyjna 44
12.1. Serwisy aplikacyjne 44
12.2. Hooki aplikacyjne 45
12.3. Współpraca serwisów i hooków 46
13. Warstwa infrastrukturalna 47
13.1. Klient HTTP 47
13.2. Interceptor autoryzacji 47
13.3. Adapter do pamięci lokalnej 48
14. Warstwa prezentacyjna 50
14.1. Struktura katalogów 50
14.2. Komponent prezentacyjny pojedynczej promocji 52
14.3. Konteksty aplikacyjne 56
15. Zarządzanie stanem aplikacji 57
15.1. AuthContext — mechanizmy i implementacja 57
15.2. FavouritesContext – zarządzanie ulubionymi promocjami 58
15.3. Zarządzanie stanem lokalnym 59
16. Zarządzanie stanem aplikacji 61
16.1. Globalny stan aplikacji (React Context) 61
16.2. Zarządzanie stanem lokalnym 62
16.3. Synchronizacja i aktualizacja stanu 62
16.4. Wzorce użycia i korzyści 62
17. Komunikacja z backendem 63
17.1. Obsługa zapytań HTTP 63
17.2. Struktura zapytań HTTP 63
17.3. Obsługa błędów 64
17.4. Format danych 65
18. Narzędzia i funkcje użytkowe (src/utilities) 66
18.1. Formatowanie dat 66
18.2. Inne funkcje użytkowe 67

19. Proces wdrożenia aplikacji 68
19.1. Budowanie i uruchamianie aplikacji 68
19.2. CI/CD i środowisko produkcyjne 68
19.3. Konfiguracja zmiennych środowiskowych 69

1. Architektura systemu

Aplikacja **PromoComparer** została zaprojektowana z zachowaniem bieżących zasada inżynierii oprogramowania, wykorzystując architekturę trójwarstwową, która wspiera modularność, łatwość utrzymania oraz rozszerzalność systemu. Całość oparta jest na platformie ASP.NET Core, co zapewnia wysoką wydajność, wsparcie dla RESTful API oraz szeroką integrację z ekosystemem .NET.

Pierwszą warstwę systemu stanowi **interfejs API**, czyli zestaw kontrolerów odpowiedzialnych za odbieranie i przetwarzanie żądań HTTP. Kontrolery odpowiadają za mapowanie przychodzących zapytań do odpowiednich metod logiki biznesowej oraz za serializację wyników do formatu JSON, który jest standardowym sposobem komunikacji z frontendem aplikacji. Każdy z kontrolerów pełni określoną rolę tematyczną – przykładowo, osobne kontrolery odpowiadają za operacje związane z promocjami, gazetkami, sklepami, kategoriami czy interakcją z systemem OpenAI. Dzięki temu separacja odpowiedzialności w tej warstwie jest zachowana, co ułatwia zarówno testowanie, jak i dalszy rozwój aplikacji.

Kolejnym filarem architektury jest **warstwa logiki biznesowej**, która obejmuje zestaw serwisów implementujących kluczowe funkcjonalności aplikacji. Serwisy odpowiadają za realizację procesów takich jak analiza obrazów przy użyciu modeli językowych OpenAI, harmonogramowanie pobierania gazetek promocyjnych, transformację danych z formatu JSON do struktur domenowych, zarządzanie danymi o promocjach oraz logikę personalizacji użytkownika. Każdy serwis jest projektowany

jako samodzielna jednostka funkcjonalna, co pozwala na niezależne testowanie, reużywalność kodu oraz łatwe wprowadzanie modyfikacji bez wpływu na inne obszary systemu.

Trzecią warstwą systemu jest **warstwa dostępu do danych**, która obejmuje zarówno definicje modeli domenowych (encje), jak i klasę kontekstu bazy danych ApplicationDbContext. Modele reprezentują poszczególne jednostki danych, takie jak promocje, kategorie, sklepy czy gazetki, i zawierają informacje o relacjach między nimi. Kontekst bazy danych jest odpowiedzialny za konfigurację dostępu do SQL Server oraz zarządzanie cyklem życia danych z wykorzystaniem technologii Entity Framework Core. Migracje definiowane w tej warstwie umożliwiają ewolucję schematu bazy danych w sposób kontrolowany i spójny z logiką aplikacyjną.

Cała struktura aplikacji została zorganizowana w sposób wspierający zasadę pojedynczej odpowiedzialności (Single Responsibility Principle) oraz rozdzielenie zależności (Separation of Concerns). Dzięki temu system charakteryzuje się wysokim poziomem czytelności, łatwością testowania jednostkowego, a także odpornością na błędy wynikające ze sprzężenia między komponentami.

2. Funkcjonalności

2.1. Scrapowanie gazetek promocyjnych

Jednym z głównych zadań systemu PromoComparer jest automatyczne pozyskiwanie treści promocyjnych w postaci gazetek reklamowych publikowanych na stronach internetowych sieci handlowych. Proces ten jest inicjowany przez ScrapingController i realizowany w PdfHandlerService. Scrapowanie obejmuje lokalizację plików PDF na stronach docelowych, pobieranie ich do systemu plików aplikacji oraz przygotowanie do dalszego przetwarzania.

Pliki zapisywane są w katalogach przypisanych do poszczególnych sklepów (Stem jako identyfikator). Operacja ta jest zautomatyzowana i może być wyzwalana ręcznie lub cyklicznie przy pomocy harmonogramu.

2.2. Konwersja PDF do formatu graficznego

Po pobraniu plików PDF, kolejnym krokiem przetwarzania jest ich konwersja do formatu rastrowego (JPEG, PNG). Operacja ta realizowana jest przez metodę ConvertAllPdfsToImagesAndDelete() w serwisie PdfHandlerService. Proces konwersji jest niezbędny, ponieważ analiza treści odbywa się na poziomie obrazu, a nie pliku PDF.

Konwersja realizowana jest z wykorzystaniem narzędzi zewnętrznych, które transformują każdą stronę PDF na osobny plik graficzny. Pliki te są następnie

przetwarzane przez usługę OpenAI. Oryginalne pliki PDF są usuwane po zakończeniu konwersji, co pozwala na optymalizację przestrzeni dyskowej.

2.3. Analiza treści obrazu z wykorzystaniem OpenAI

Najbardziej zaawansowaną funkcjonalnością systemu jest semantyczna analiza obrazów gazetek promocyjnych przy użyciu modeli językowych OpenAI. Proces ten realizowany jest w serwisie OpenAIService, który implementuje interfejs IOpenAIService.

Metoda ParseImagesToFunction() iteruje po folderach zawierających obrazy gazet promocyjnych, a następnie dla każdego obrazu wywołuje metodę GetJsonPromotions(). Ta z kolei konstruuje odpowiedni prompt, wysyła obraz do API OpenAI i oczekuje odpowiedzi w formacie JSON zawierającej szczegóły oferty promocyjnej. Przykładowy zestaw danych pozyskanych z modelu może obejmować:

- nazwę produktu,
- cenę pierwotną i promocyjną,
- typ promocji (np. rabat procentowy, 2+1),
- jednostkę sprzedaży (np. kg, szt.),
- daty obowiązywania promocji.

Uzyskane dane są następnie przekształcane do modelu Promotion i zapisywane w bazie danych przy użyciu serwisu PromotionService.

2.4. Zarządzanie promocjami i ich filtrowanie

System pozwala na zarządzanie strukturą danych dotyczących promocji, w tym ich tworzenie, pobieranie, filtrowanie i udostępnianie przez API. Funkcje te obsługiwane są przez PromotionController oraz IPromotionService.

Użytkownik może pobierać:

- wszystkie promocje (/api/promotions),
- tylko aktywne promocje (/api/promotions/active),
- promocje w konkretnym sklepie (/api/promotions/store/{storeId}),
- promocje z wybranej kategorii (/api/promotions/category/{categoryId}),
- 10 największych promocji według wartości rabatu (/api/promotions/top).

Wszystkie operacje są optymalizowane poprzez DTO, które dostarczają jedynie istotne informacje z punktu widzenia frontendu. Ponadto system oblicza rabaty (kwotowo i procentowo) na etapie ekstrakcji danych.

2.5. Zarządzanie strukturą sklepów i kategorii

System umożliwia rejestrowanie sklepów (Store) oraz tworzenie i utrzymywanie listy kategorii produktów (Category). Operacje na tych jednostkach realizowane są przez kontrolery StoreController oraz CategoryController, a logikę obsługują odpowiednie serwisy implementujące interfejsy IStoreService i ICategoryService.

Sklepy identyfikowane są przez unikalną nazwę oraz atrybut Stem, który służy do identyfikacji URL źródłowych gazetek. Kategorie natomiast wykorzystywane są do klasyfikacji produktów promocyjnych w systemie.

Zarówno sklepy, jak i kategorie mogą być tworzone ręcznie (np. z listy w konfiguracji), jak również automatycznie, jeśli dane pojawią się w trakcie analizy obrazu przez OpenAI.

2.6. Personalizacja użytkownika – ulubione promocje

Aplikacja obsługuje funkcję dodawania promocji do listy ulubionych, co pozwala użytkownikowi na personalizację interfejsu oraz szybki dostęp do interesujących go ofert. Dane te są przechowywane w tabeli Favourites i powiązane bezpośrednio z użytkownikiem systemu (User).

Interakcja z listą ulubionych odbywa się przez kontroler UserPanelController, który obsługuje dodawanie i usuwanie pozycji oraz pobieranie listy ulubionych ofert użytkownika. Funkcja ta wymaga uwierzytelnienia — aplikacja wykorzystuje tożsamość użytkownika na podstawie ClaimsPrincipal.

2.7. Automatyzacja zadań cyklicznych

System wyposażony jest w harmonogram zadań realizowany przy pomocy biblioteki **Coravel**, która umożliwia cykliczne uruchamianie określonych procesów — bez konieczności korzystania z zewnętrznego systemu CRON.

Harmonogram automatycznie uruchamia proces: pobierania gazetek, konwersji PDF oraz analizy przez OpenAI. Dzięki temu baza danych jest stale aktualizowana, a system może działać w trybie niemal bezobsługowym.

3. Konfiguracja i środowisko uruchomieniowe

Konfiguracja środowiska aplikacji PromoComparer odbywa się w pliku Program.cs, który wykorzystuje skrócony model hostowania dostępny w ASP.NET Core 6+. Wspomniany plik zawiera wszystkie niezbędne kroki uruchomienia aplikacji -począwszy od wczytania konfiguracji, przez rejestrację zależności i usług, aż po zdefiniowanie ścieżek HTTP i uruchomienie serwera.

3.1. Inicjalizacja hosta i wczytanie konfiguracji

Proces konfiguracji rozpoczyna się od utworzenia instancji WebApplicationBuilder

```
var builder = WebApplication.CreateBuilder(args);
```

Rys. 3. 1 Inicjalizacja aplikacji ASP .NET Core w stylu "minimal hosting"

ASP.NET Core automatycznie ładuje:

- appsettings.json – bazowy plik konfiguracyjny,
- appsettings.{Environment}.json – plik środowiskowy, np. appsettings.Development.json,
- zmienne środowiskowe – np. zdefiniowane w systemie

3.2. Rejestracja usług i zależności (Dependency Injection)

System korzysta z wbudowanego kontenera DI (Dependency Injection), do którego rejestrowane są wszystkie potrzebne komponenty: kontekst bazy danych, serwisy aplikacyjne oraz planista Coravel.

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(conString));
builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
```

Rys. 3.2. Konfiguracja usług w ASP .NET Core

Następnie rejestrowane są konkretne serwisy, zgodnie z ich interfejsami.

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddTransient<IPdfHandlerService, PdfHandlerService>();
builder.Services.AddScoped<IPromotionService, PromotionService>();
builder.Services.AddScoped<IStoreService, StoreService>();
builder.Services.AddScoped<ILeafletService, LeafletService>();
builder.Services.AddScoped<ICategoryService, CategoryService>();
builder.Services.AddScoped<IUserPanelService, UserPanelService>();
builder.Services.AddTransient<IOpenAIService, OpenAIService>(sp =>
{
    var apiKey = builder.Configuration["OPENAI_API_KEY"];
    var promotionService = sp.GetRequiredService<IPromotionService>();
    var categoryService = sp.GetRequiredService<ICategoryService>();

    return new OpenAIService(apiKey, promotionService, categoryService);
});
builder.Services.AddTransient<Scheduler>();

// Add Coravel's scheduling services
builder.Services.AddScheduler();
```

Rys. 3. 3 Rejestracja usług w kontenerze DI wraz z harmonogramowaniem

3.3. Konfiguracja CORS, autoryzacji i tożsamości

3.3.1. CORS

System umożliwia dostęp do API z dowolnej domeny (np. dla frontendowej części aplikacji)

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAll", builder =>
    {
        builder.AllowAnyOrigin()
               .AllowAnyMethod()
               .AllowAnyHeader();
    });
});
```

Rys. 3. 4 Konfiguracja polityki CORS

Powyższy fragment rejestruje w kontenerze usług prostą, otwartą politykę CORS o nazwie AllowAll. Dzięki niej każda domena (origin) może wysyłać żądania do Twojego API, używając dowolnych metod HTTP (GET, POST, PUT, DELETE itp.) oraz przesyłając dowolne nagłówki.

3.3.2. Uwierzytelnianie i autoryzacja

Aplikacja korzysta z ASP.NET Core Identity z obsługą ciasteczek i tokenów JWT

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme = IdentityConstants.ApplicationScheme;
    options.DefaultChallengeScheme = IdentityConstants.ApplicationScheme;
})
.AddCookie(IdentityConstants.ApplicationScheme)
.AddBearerToken(IdentityConstants.BearerScheme);
```

Rys. 3.5 Konfiguracja uwierzytelniania w ASP .NET Core

Powyższy fragment kodu rejestruje w kontenerze usług mechanizmy uwierzytelniania aplikacji. Najpierw za pomocą AddAuthentication ustalane są domyślne schematy: DefaultScheme oraz DefaultChallengeScheme wskazują na własny, zdefiniowany w IdentityConstants.ApplicationScheme. Następnie metoda AddCookie wprowadza uwierzytelnianie oparte na ciasteczkach dla mechanizmu „Application", co pozwala na sesyjną obsługę logowania użytkowników. Dzięki wywołaniu AddBearerToken (najczęściej mapowanemu na JwtBearer) do aplikacji trafiają także żądania z nagłówkiem Authorization: Bearer <token>, umożliwiając bezstanowe uwierzytelnianie tokenami JWT.

Dodatkowo zdefiniowane jest zachowanie w przypadku nieautoryzowanego żądania

```
builder.Services.ConfigureApplicationCookie(options =>
{
    options.Events.OnRedirectToLogin = context =>
    {
        context.Response.StatusCode = StatusCodes.Status401Unauthorized;
        return Task.CompletedTask;
    };
});
```

Rys. 3. 6 Dostosowanie zachowania przekierowania do logowania

Powyższy fragment kodu konfiguruje opcje cookie-based authentication, aby w przypadku braku uwierzytelnienia nie wykonywać domyślnego przekierowania na stronę logowania, lecz zwrócić od razu status HTTP 401 Unauthorized. Dzięki podpięciu zdarzenia OnRedirectToLogin i ustawieniu context.Response.StatusCode = StatusCodes.Status401Unauthorized, klienci API otrzymują jednoznaczną odpowiedź o braku uprawnień zamiast HTML-owego przekierowania

3.3.3. Rejestracja ról i użytkowników

```
builder.Services.AddIdentityCore<User>(options => { /* Identity options if any */ })
    .AddRoles<IdentityRole>() // Add this line to support roles
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddApiEndpoints();
```

Rys. 3. 7 Konfiguracja ASP .NET Core Identity z obsługą ról

W powyższym kodzie:

- User to klasa definiująca model użytkownika aplikacji,
- AddRoles<IdentityRole>() aktywuje obsługę ról,
- AddEntityFrameworkStores<ApplicationDbContext>() wskazuje, że dane o użytkownikach będą przechowywane w bazie danych za pomocą EF Core,
- AddApiEndpoints() dodaje domyślne endpointy API dla zarządzania kontami.

3.4. Budowanie aplikacji i konfiguracja pipeline'u HTTP

Po skonfigurowaniu usług, tworzona jest aplikacja webowa

```
var app = builder.Build();
```

Rys. 3. 8 Utworzenie aplikacji webowej

W środowisku deweloperskim aplikacja uruchamia Swagger oraz migracje.

Realizowana zostaje ścieżka przetwarzania przychodzących żądań w zależności od środowiska uruchomieniowego aplikacji. W trybie deweloperskim (*IsDevelopment())* aktywuje się dokumentacja **OpenAPI** (metody *UseSwagger() i UseSwaggerUI())* oraz automatyczne zastosowanie migracji bazy danych (*ApplyMigrations()),* co przyspiesza iterację nad zmianami w modelu. W środowisku produkcyjnym natomiast włącza się przekierowanie ruchu HTTP na HTTPS *(UseHttpsRedirection())*

```
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
    app.ApplyMigratons();
}
else
{
    app.UseHttpsRedirection();
}
```

Rys. 3. 9 Konfiguracja potoku żądań HTTP

Uruchamiany jest harmonogram

```
app.Services.UseScheduler(scheduler =>
{
    scheduler.Schedule<Scheduler>()
        .DailyAt(0, 0);
});
```

Rys. 3. 10 Uruchomienie harmonogramu

3.5. Tworzenie ról przy starcie aplikacji

Po uruchomieniu aplikacji, role **Admin** i **User** są tworzone automatycznie w bazie danych w ramach procedury inicjalizacyjnej. Proces ten jest realizowany w bloku inicjalizacji, który wykonuje się natychmiast po zbudowaniu aplikacji, zanim zostanie uruchomiony serwer HTTP.

Kod odpowiedzialny za to znajduje się bezpośrednio po wywołaniu builder.Build()

```
using (var scope = app.Services.CreateScope())
{
    var roleManager = scope.ServiceProvider.GetRequiredService<RoleManager<IdentityRole>>();
    EnsureRoles(roleManager); // call the synchronous version
}
```

4. Przepływ danych

System PromoComparerAPI realizuje kompletny cykl przetwarzania danych promocyjnych — począwszy od pozyskiwania gazet promocyjnych z zewnętrznych źródeł, przez analizę treści z użyciem sztucznej inteligencji, aż po zapis danych w relacyjnej bazie danych i udostępnienie ich przez REST API.

4.1. Pozyskanie danych źródłowych

Proces pozyskiwania danych promocyjnych rozpoczyna się od **scrapowania** — czyli automatycznego pobierania gazet promocyjnych (plików PDF) z witryn internetowych.

Mechanizm ten może zostać uruchomiony na dwa sposoby:

- bezpośrednie wywołanie endpointu API -POST /api/scraping/parse
- automatyczny **harmonogram Coravel**, który wyzwala scrapowanie codziennie o północy

```
app.Services.UseScheduler(scheduler =>
{
    scheduler.Schedule<Scheduler>()
        .DailyAt(0, 0);
});
```

Rys. 4. 1 Harmonogramowanie zadania

Logikę biznesową obsługuje klasa *PdfHandlerService*

Wewnątrz tej metody system:

- iteruje po zdefiniowanych sklepach (Store),
- sprawdza, czy dany sklep ma przypisany adres źródłowy (Stem),
- wykonuje żądania HTTP do pobrania dostępnych plików PDF,
- zapisuje je do lokalnego katalogu roboczego.

Każdy plik trafia do podfolderu odpowiadającego nazwie sklepu, co pozwala łatwo kontrolować ich źródło i identyfikację podczas dalszego przetwarzania.

4.2. Konwersja plików PDF do obrazów

Aby umożliwić analizę treści zawartej w gazetkach, pliki PDF muszą zostać przekształcone. W tym celu aplikacja uruchamia metodę:

```
public void ConvertAllPdfsToImagesAndDelete()
{
    var pdfFiles = Directory.GetFiles(_pdfDirectory, "*.pdf", SearchOption.AllDirectories);

    Parallel.ForEach(pdfFiles, pdfFile =>
    {
        try
        {
            Console.WriteLine($"Converting {pdfFile}");
            ConvertPdfToImages(pdfFile);


            File.Delete(pdfFile);
            Console.WriteLine($"Deleted {pdfFile}");
        }

        catch (Exception ex)
        {
            Console.WriteLine($"Error processing PDF: {ex.Message} with {pdfFile}");
        }
    });
}
```

Rys. 4. 2 Metoda ConverAllPdfsToImageAndDelete

Proces działa w następujących krokach:

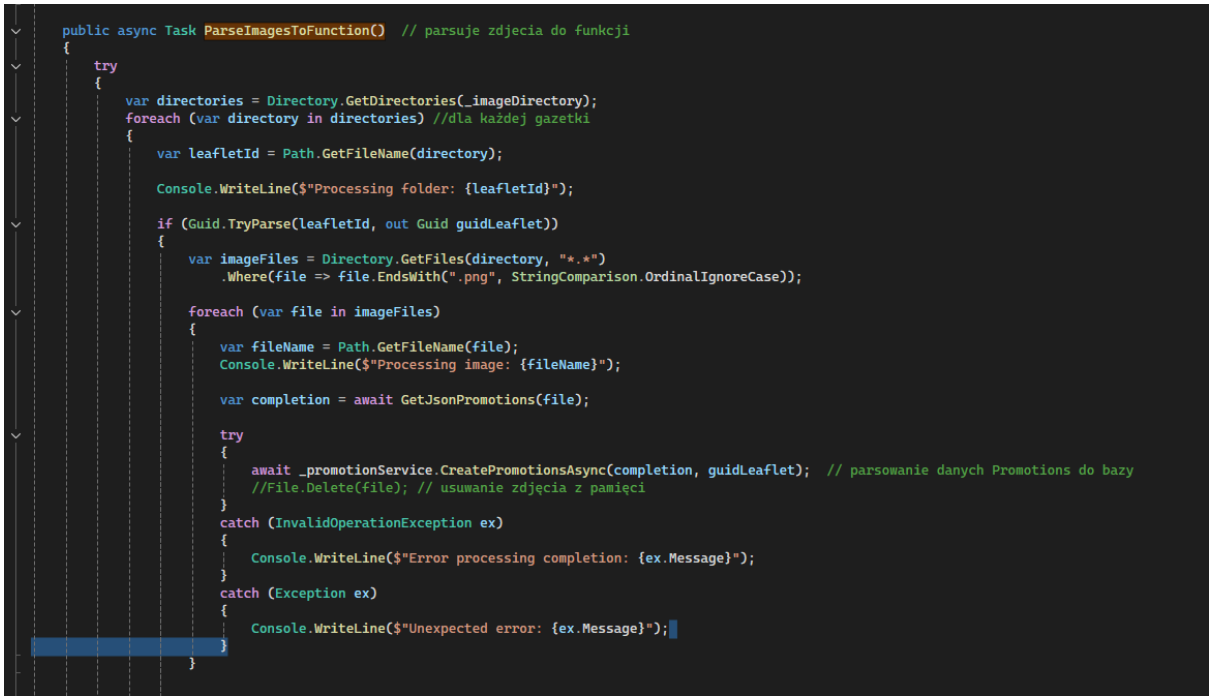
- dla każdego pliku PDF odczytywana jest liczba stron,
- każda strona jest konwertowana na plik JPEG/PNG,
- pliki są zapisywane do katalogu roboczego, a oryginalne PDF-y usuwane.

Do konwersji najprawdopodobniej wykorzystywana jest biblioteka zewnętrzna. Pliki graficzne są przygotowywane do przetworzenia przez AI i zapisywane z identyfikowalnymi nazwami.

4.3. Analiza semantyczna obrazu (OpenAI)

Najbardziej zaawansowany etap przepływu danych to **ekstrakcja informacji semantycznej** z obrazów za pomocą modeli językowych OpenAI.

Klasa *ParseImagesToFunction*, która realizuje zaawansowany etap przepływu danych – wywołanie modelu GPT-4O w celu wykrycia i sformatowania informacji o promocjach. Metoda rozpoczyna się od pobrania listy katalogów w folderze Assets, z których każdy nazywa się identyfikatorem ulotki (GUID). Dla każdego folderu następuje weryfikacja poprawności GUID-a, a następnie przetworzenie wszystkich plików PNG.



```
public async Task ParseImagesToFunction()  // parsuje zdjecia do funkcji
{
    try
    {
        var directories = Directory.GetDirectories(_imageDirectory);
        foreach (var directory in directories) //dla każdej gazetki
        {
            var leafletId = Path.GetFileName(directory);

            Console.WriteLine($"Processing folder: {leafletId}");

            if (Guid.TryParse(leafletId, out Guid guidLeaflet))
            {
                var imageFiles = Directory.GetFiles(directory, "*.*")
                    .Where(file => file.EndsWith(".png", StringComparison.OrdinalIgnoreCase));

                foreach (var file in imageFiles)
                {
                    var fileName = Path.GetFileName(file);
                    Console.WriteLine($"Processing image: {fileName}");

                    var completion = await GetJsonPromotions(file);

                    try
                    {
                        await _promotionService.CreatePromotionsAsync(completion, guidLeaflet);  // parsowanie danych Promotions do bazy
                        //File.Delete(file); // usuwanie zdjęcia z pamięci
                    }
                    catch (InvalidOperationException ex)
                    {
                        Console.WriteLine($"Error processing completion: {ex.Message}");
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine($"Unexpected error: {ex.Message}");
                    }
                }
```

Rys. 4.3 Funkcja ParseImagesToFunction

Kolejny kluczowy element to sposób, w jaki przygotowywany jest prompt i dane wejściowe do modelu. Metoda GetJsonPromotions wczytuje zawartość pliku jako strumień bajtów, załącza go wraz z opisem w formie tekstowej i prosi GPT-4O o zwrócenie danych wyłącznie w formacie JSON

Dla każdego folderu sklepu i każdego obrazu system:

- tworzy prompt tekstowy zawierający instrukcje dla modelu OpenAI (np. "Wypisz wszystkie promocje na stronie..."),
- wywołuje model OpenAI z obrazem jako inputem (model typu GPT-4 z obsługą obrazu lub inna kombinacja OCR + NLP),
- otrzymuje odpowiedź w formacie JSON z polami:
    - productName
    - originalPrice, discountPrice
    - promotionType
    - unit
    - validFrom, validTo
    - categoryName

Otrzymany wynik jest parsowany do obiektów .NET i przekazywany do kolejnego etapu.

4.4. Transformacja danych do modeli i zapis w bazie

Po uzyskaniu danych z modelu AI, system przystępuje do ich transformacji do struktur wewnętrznych i zapisuje je w bazie danych. Obsługuje to głównie PromotionService, ale również inne serwisy wspomagające:

- CategoryService – kategorie produktów,
- StoreService – identyfikacja sklepu,
- LeafletService – przypisanie do gazetki.

Promocje są zapisywane do tabeli Promotions z pełnym zestawem informacji, w tym:

- Id sklepu i kategorii (relacje obce),
- Opis, ceny, typ promocji, jednostka,
- Daty obowiązywania,
- Ewentualny identyfikator zewnętrzny (do deduplikacji).

System zapewnia spójność danych — kategorie i sklepy są tworzone dynamicznie, jeśli nie istnieją.

4.5. Udostępnianie danych przez API

Zgromadzone dane są udostępniane klientom (np. aplikacji frontendowej) przez *RESTful API*. Interfejsy kontrolerów takie jak *PromotionController* pozwalają na różnorodne operacje filtrowania i wyszukiwania:

Dane są serializowane do formatu JSON za pomocą DTO, co pozwala na:

- uproszczenie modelu danych dla frontendu,
- zmniejszenie rozmiaru odpowiedzi HTTP,
- ukrycie pól wewnętrznych, np. StoreId, EF navigation properties.

Wszystkie żądania są przetwarzane przez *IPromotionService* oraz *ApplicationDbContext*.

4.6. Personalizacja i interakcje użytkownika

Użytkownicy zarejestrowani w systemie mogą korzystać z funkcji personalizacji. Główna funkcja to możliwość dodania promocji do ulubionych. Dane te są przechowywane w relacji wiele-do-wielu między User i Promotion:

- Tabela pośrednia: FavouritePromotion

- Obsługa: UserPanelController i UserPanelService

Dostęp do tych zasobów wymaga autoryzacji (token *JWT* lub ciasteczko *auth cookie*). Tożsamość użytkownika odczytywana jest z *ClaimsPrincipal*.

5. Struktura bazy danych

Aplikacja PromoComparer wykorzystuje relacyjną bazę danych SQL Server, zarządzaną za pomocą Entity Framework Core. Schemat danych został zaprojektowany w sposób modularny, z uwzględnieniem zasad normalizacji i separacji odpowiedzialności.

5.1. Tabela Store – Sklep

Tabela Store reprezentuje sieci handlowe, których gazetki są pobierane, analizowane i prezentowane w systemie. Każdy sklep może mieć przypisane wiele gazetek (Leaflet), co odzwierciedla naturalną relację między siecią a jej kampaniami promocyjnymi.

Atrybuty:

- Id (Guid) – klucz główny, jednoznacznie identyfikuje sklep.
- Leaflets (ICollection<Leaflet>) – lista gazetek przypisanych do sklepu.

Relacje:

- **1 : N** z tabelą Leaflet – jeden sklep może mieć wiele gazet.

5.2. Tabela Leaflet – Gazetka promocyjna

Encja Leaflet odwzorowuje konkretną gazetkę promocyjną, która obowiązuje przez określony czas i zawiera zestaw promocji (Promotion). Powiązana jest jednoznacznie z danym sklepem (Store), a jej treść może być analizowana automatycznie lub ręcznie.

Atrybuty:

- Id (Guid) – identyfikator gazetki.
- StartDate (DateTime) – data rozpoczęcia promocji.
- EndDate (DateTime) – data zakończenia promocji.

- PdfLink (string) – link URL do pobrania gazetki w formacie PDF.
- StoreId (Guid) – klucz obcy do tabeli Store.
- Store (Store) – relacja nawigacyjna do sklepu.
- Promotions (ICollection<Promotion>) – zbiór promocji przypisanych do gazetki.

Relacje:

- **N : 1** z Store – gazetka należy do jednego sklepu.
- **1 : N** z Promotion – gazetka zawiera wiele promocji.

5.3. Tabela Promotion – Promocja

Tabela Promotion zawiera szczegółowe dane dotyczące pojedynczych ofert promocyjnych, które zostały zidentyfikowane w gazetkach. Zawiera informacje o zakresie obowiązywania, produktach, cenach oraz przynależności do kategorii. Jest centralną encją systemu.

Atrybuty:

- Id (Guid) – identyfikator promocji.
- UntilOutOfStock (bool) – flaga określająca, czy promocja trwa do wyczerpania zapasów.
- LeafletId (Guid) – odwołanie do gazetki, w której występuje promocja.
- Leaflet (Leaflet) – relacja nawigacyjna do gazetki.
- CategoryId (Guid) – odniesienie do kategorii produktu.
- Category (Category) – relacja nawigacyjna do encji Category.
- Favourites (ICollection<Favourite>) – relacja do użytkowników, którzy dodali promocję do ulubionych.

Relacje:

- **N : 1** z Leaflet – promocja należy do jednej gazetki.
- **N : 1** z Category – przypisana do jednej kategorii.
- **1 : N** z Favourite – może być ulubioną wielu użytkowników.

5.4. Tabela Category – Kategoria

Encja Category reprezentuje klasyfikację produktów promocyjnych. Umożliwia filtrowanie i grupowanie ofert według typu produktów, co ułatwia użytkownikom przeglądanie interesujących ich promocji.

Atrybuty:

- Id (Guid) – identyfikator kategorii.
- Promotions (ICollection<Promotion>) – lista przypisanych promocji.

Relacje:

- **1 : N** z Promotion – każda kategoria może mieć wiele przypisanych promocji.

5.5. Tabela Favourite – Ulubiona promocja

Tabela Favourite to encja pośrednia, implementująca relację wiele-do-wielu między użytkownikami a promocjami. Umożliwia zapisywanie przez użytkowników swoich ulubionych promocji, co jest funkcją personalizacji.

Atrybuty:

- UserId (string) – identyfikator użytkownika (część klucza głównego).
- PromotionId (Guid) – identyfikator promocji (część klucza głównego).
- User (User) – relacja nawigacyjna do encji użytkownika.
- Promotion (Promotion) – relacja nawigacyjna do encji promocji.

Relacje:

- **N : 1** z User – wielu użytkowników może oznaczyć tę samą promocję.
- **N : 1** z Promotion – jedna promocja może być ulubiona przez wielu użytkowników.

5.6. Tabela User – Użytkownik systemu

Encja użytkownika (User) jest rozszerzeniem modelu ASP.NET Core Identity. Umożliwia autoryzację, zarządzanie rolami (Admin, User) oraz integrację z funkcją ulubionych promocji.

Atrybuty :

- Id (string) – unikalny identyfikator użytkownika.
- Favourites – kolekcja ulubionych promocji.

Relacje:

- **1 : N** z Favourite – użytkownik może mieć wiele ulubionych promocji.

6. Procedury przetwarzania danych

W systemie **PromoComparer** zdefiniowane są procedury aplikacyjne (metody kontrolerów i serwisów), które agregują dane z wielu tabel i wykonują obliczenia domenowe, takie jak analiza rabatów, filtrowanie po sklepie czy kategorii. Poniżej znajduje się zestawienie kluczowych procedur wykorzystywanych do obsługi zapytań użytkowników.

6.1. GetPromotionsByStore

Procedura służy do pobierania wszystkich promocji przypisanych do sklepu o podanym identyfikatorze. Wyniki zawierają szczegóły dotyczące produktu, rabatu i przynależności do sklepu i kategorii.

Wykorzystywane encje:

- Stores
- Leaflets
- Promotions
- Categories

Logika działania:

1. Na podstawie StoreId, pobierane są wszystkie gazetki (Leaflets) powiązane z danym sklepem.
2. Z tych gazetek wyciągane są wszystkie przypisane promocje (Promotions).
3. Dla każdej promocji:
    a. Obliczana jest wartość rabatu (różnica między ceną oryginalną a promocyjną).
    b. Obliczany jest procentowy rabat:
4. Formatowane są daty rozpoczęcia i zakończenia promocji.
5. Zwracana jest lista obiektów zawierających:
    a. Id promocji
    b. ProductName
    c. OriginalPrice i PriceAfterPromotion
    d. DiscountAmount i DiscountPercent
    e. CategoryName
    f. StoreName

6.2. GetTop10LargestPromotions

Procedura odpowiada za identyfikację 10 najbardziej atrakcyjnych promocji — pod względem wysokości rabatu w złotówkach, a w przypadku remisu również procentowego.

Wykorzystywane encje:

- Promotions
- Leaflets
- Categories
- Stores

Logika działania:

1. Pobierane są wszystkie **aktywne** promocje (te, których EndDate jest większy niż data bieżąca).
2. Dla każdej promocji obliczane są:
    a. DiscountAmount = OriginalPrice - PriceAfterPromotion
    b. DiscountPercent = analogicznie jak wyżej
3. Promocje są sortowane:
    a. malejąco według DiscountAmount
    b. następnie malejąco według DiscountPercent
4. Zwracane jest **top 10** promocji z polami:
    a. Id, ProductName, OriginalPrice, PriceAfterPromotion
    b. DiscountAmount, DiscountPercent
    c. CategoryName, StoreName, StoreId

6.3. GetPromotionsByCategory

Procedura umożliwia przegląd wszystkich promocji przypisanych do konkretnej kategorii (np. „nabiał”, „napoje gazowane”).

Wykorzystywane encje:

- Categories
- Promotions
- Leaflets
- Stores

Logika działania:

1. Na podstawie CategoryId system pobiera wszystkie promocje przypisane do tej kategorii.
2. Promocje są filtrowane – pozostają tylko te, których data zakończenia (EndDate) jest przyszła.
3. Dla każdej promocji obliczany jest:
    a. DiscountAmount i DiscountPercent (analogicznie jak wyżej).
4. Wyniki są formatowane i zawierają:
    a. Id promocji
    b. ProductName
    c. OriginalPrice, PriceAfterPromotion
    d. DiscountAmount, DiscountPercent
    e. CategoryName, StoreName

7. Kontrolery

Warstwa kontrolerów w aplikacji PromoComparerpełni rolę bramy wejściowej do systemu. Każdy kontroler odpowiada za określoną część logiki aplikacyjnej i pośredniczy między żądaniami HTTP a warstwą serwisów.

7.1. CategoryController

CategoryController pełni rolę warstwy pośredniczącej pomiędzy klientem API a serwisem zarządzającym kategoriami produktów. Został oznaczony atrybutem [ApiController] oraz zmapowany na ścieżkę api/Categories. W konstruktorze przyjmuje implementację interfejsu ICategoryService oraz logger, co umożliwia delegowanie logiki biznesowej do warstwy serwisów i rejestrowanie zdarzeń lub błędów.

```
[ApiController]
[Route("api/Categories")]
public class CategoryController : ControllerBase
{
    private readonly ICategoryService _categoryService;
    private readonly ILogger<CategoryController> _logger;

    public CategoryController(ICategoryService categoryService, ILogger<CategoryController> logger)
    {
        _categoryService = categoryService;
        _logger = logger;
    }

    [HttpGet]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<IEnumerable<CategoryDto>>> GetAllCategories()
    {
        try
        {
            var categoryDtos = await _categoryService.GetAllCategoriesAsync();
            return Ok(categoryDtos);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An error occurred while retrieving categories.");
            return StatusCode(StatusCodes.Status500InternalServerError, "An error occurred while retrieving categories.");
        }
    }
```

Rys.7.1 Kod kontrolera CategoryController

Metoda *GetAllCategories* odpowiada na żądania HTTP GET bez dodatkowych parametrów pod adresem /api/Categories.

Wewnątrz tej akcji wywoływana jest asynchronicznie metoda serwisu, zwracająca kolekcję DTO reprezentujących kategorie. W przypadku powodzenia controller zwraca kod 200 ze zserializowaną listą, natomiast wszelkie wyjątki są przechwytywane, logowane jako błąd i przekierowywane do klienta z kodem 500. Dodatkowo, *CreateCategoryFromList*, dostępna pod ścieżką POST /api/Categories/from-list, inicjuje jednokrotne utworzenie kategorii na podstawie wewnętrznie zdefiniowanej listy.

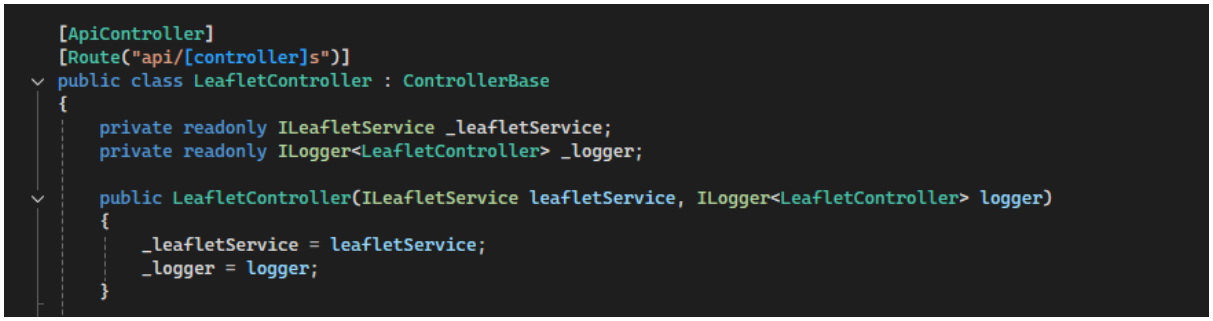
```
    //}

    [HttpPost("from-list")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status400BadRequest)]
    [ProducesResponseType(StatusCodes.Status500InternalServerError)]
    public async Task<IActionResult> CreateCategoryFromList()   // jednorazowe - wprowadza dane z wypisanej listy
    {
        try
        {
            await _categoryService.CreateCategoryFromListAsync();
            _logger.LogInformation("Categories created from the list successfully.");
            return Ok();
        }
        catch (InvalidOperationException ex)
        {
            _logger.LogWarning(ex, "Invalid operation: {ErrorMessage}", ex.Message);
            ModelState.AddModelError("", ex.Message);
            return BadRequest(ModelState);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An error occurred while creating categories from the list.");
            return StatusCode(StatusCodes.Status500InternalServerError, "An error occurred while creating categories from the list.");
        }
    }
}
```

Rys.7.2 Metoda GetAllCategories

W przypadku nieprawidłowej operacji serwer reaguje kodem 400 z informacją o błędzie, a przy nieprzewidzianych wyjątkach – kodem 500 .

7.2. LeafletController

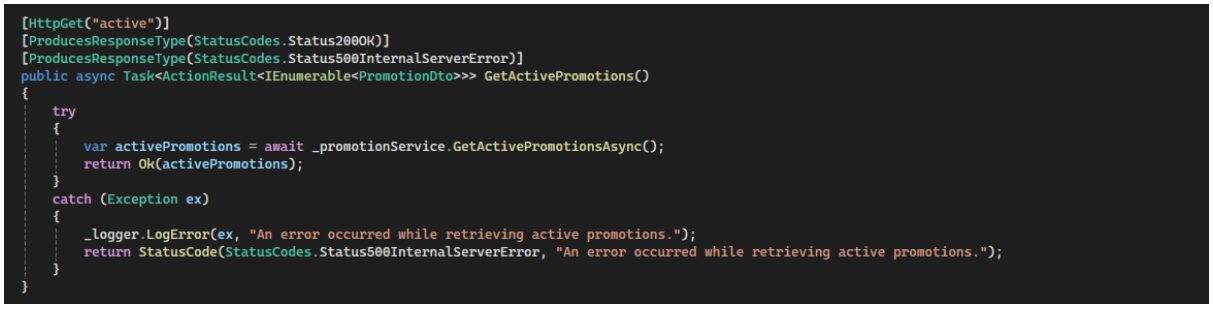
Kontroler LeafletController odpowiada za obsługę „ulotkowych” danych w systemie. Jego ścieżka bazowa to api/Leaflets, co realizowane jest przez [Route("api/[controller]s")]. Pomimo że większość metod została zakomentowana, widać intencję wsparcia pełnego CRUD—pobierania wszystkich rekordów, pojedynczego wpisu czy tworzenia nowych – każda z nich wyposażona w odpowiednie atrybuty HTTP, typy odpowiedzi i logikę przechwytywania wyjątków. Komentarze kodu sugerują, że w przyszłości kontroler ten będzie w pełni operacyjny, z użyciem ILeafletService do wykonywania operacji na obiektach DTO modelu ulotki.



Rys.7.3 Kod kontrolera LeafletController

7.3. PromotionController

W *PromotionController* skoncentrowano się na możliwościach filtrowania promocji według różnych kryteriów. Bazowa ścieżka to api/Promotions, a konstruktor przyjmuje serwis *IPromotionService* i *logger*. Metoda *GetActivePromotions*, wywoływana pod adresem GET /api/Promotions/active, zwraca aktualnie obowiązujące promocje, natomiast GetTopPromotions pod /api/Promotions/top przekazuje listę wyróżnionych ofert, wykorzystując typ DTO z dodatkowymi informacjami o topowych promocjach. Dla każdej z tych akcji przewidziano obsługę wyjątków z logowaniem błędów i zwracaniem statusu 500.



Rys. 7.4 Metoda GetActivePromotions

Dodatkowo kontroler udostępnia metodę GetAllActivePromotionsByStore, dostępną pod GET /api/Promotions/store/{storeId}, która najpierw pobiera listę promocji dla wskazanego sklepu, a w przypadku braku wyników odpowiada kodem 404. Analogicznie działa *GetAllActivePromotionsByCategory* pod /api/Promotions/category/{categoryId}, również zwracając status 404, gdy nie odnajdzie żadnej promocji dla danej kategorii. Dzięki tej strukturze klient może elastycznie pozyskiwać zarówno wszystkie aktywne promocje, jak i zawężać wyniki do konkretnego sklepu lub kategorii.

```
// GET: api/Promotions/category/{categoryId}
[HttpGet("category/{categoryId}")]
public async Task<ActionResult<IEnumerable<TopActivePromotionDto>>> GetAllActivePromotionsByCategory(Guid categoryId)
{
    try
    {
        var promotions = await _promotionService.GetAllActivePromotionsByCategory(categoryId);

        if (promotions == null || !promotions.Any())
        {
            _logger.LogInformation("No promotions found for category {CategoryId}.", categoryId);
            return NotFound($"No promotions found for category with ID {categoryId}.");
        }

        return Ok(promotions);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "An error occurred while fetching promotions for category {CategoryId}.", categoryId);
        return StatusCode(500, "Internal server error");
    }
}
```

Rys.7.5 Metoda GetAllActivePromotionsByCategory

7.4. StoreController

StoreController odpowiada za zarządzanie zasobami sklepów. Dzięki oznaczeniu [Route("api/[controller]s")] wszystkie akcje trafiają pod ścieżkę api/Stores. Metoda *GetAllStores* realizuje zapytanie GET, które przy pomocy serwisu IStoreService pobiera całą listę sklepów. Jeżeli lista jest pusta, zwracany jest kod 404, natomiast przy błędach wewnętrznych – kod 500.

```
[HttpGet]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<IEnumerable<StoreDto>>> GetAllStores()
{
    try
    {
        var storeDtos = await _storeService.GetAllStoresAsync();
        return storeDtos != null ? Ok(storeDtos) : NotFound();
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "An error occurred while retrieving stores.");
        return StatusCode(StatusCodes.Status500InternalServerError, "An error occurred while retrieving stores.");
    }
}
```

Rys.7.6 Metoda GetAllStores

Specjalna metoda *CreateStoresFromConf*, dostępna pod POST /api/Stores/all, służy do jednorazowego załadowania konfiguracji sklepów, co może być wykorzystywane przy początkowym inicjalizowaniu bazy danych. Po pomyślnym wywołaniu klient otrzymuje status 201, a wszelkie błędy są logowane i przekazywane z kodem 500 .

```
    [HttpPost("all")]
    [ProducesResponseType(StatusCodes.Status201Created)]
    [ProducesResponseType(StatusCodes.Status500InternalServerError)]
    public async Task<IActionResult> CreateStoresFromConf() //jednorazowe wywołanie
    {
        try
        {
            await _storeService.CreateStoresFromConfAsync();
            return Created();
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An error occurred while creating stores from configuration.");
            return StatusCode(StatusCodes.Status500InternalServerError, "An error occurred while creating stores from configuration.");
        }
    }
}
```

Rys.7.7 Metoda CreateStoresFromConf

7.5. OpenAIController

Kontroler OpenAIController skupia się na integracji z zewnętrzną usługą OpenAI. Mapowany pod ścieżkę [controller], czyli domyślnie /OpenAI, zdefiniował jedną akcję *ParseImagesToFunction*, dostępną pod POST /OpenAI/analize-images. Jej zadaniem jest wywołanie metody serwisu IOpenAIService, która odpowiada za przetwarzanie obrazów za pomocą API OpenAI. Dzięki loggerowi użytkownik otrzymuje informację zwrotną o sukcesie lub szczegóły błędu w przypadku niepowodzenia.

```
using PromoComparerAPI.Interfaces;

namespace PromoComparerAPI.Controllers;

[ApiController]
[Route("[controller]")]
public class OpenAIController : ControllerBase
{
    private readonly IOpenAIService _openAIService;
    private readonly ILogger<OpenAIController> _logger;

    public OpenAIController(IOpenAIService openAIService, ILogger<OpenAIController> logger)
    {
        _openAIService = openAIService;
        _logger = logger;
    }

    [HttpPost("analize-images")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status500InternalServerError)]
    public async Task<IActionResult> ParseImagesToFunction()
    {
        try
        {
            await _openAIService.ParseImagesToFunction();
            _logger.LogInformation("OpenAI successfully integrated.");
            return Ok("OpenAI successfully integrated.");
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An error occurred while integrating OpenAI.");
            return StatusCode(StatusCodes.Status500InternalServerError, "An error occurred while integrating OpenAI.");
        }
    }
}
```

Rys. 7.8 Kod kontrolera OpenAIController

7.6. ScrapingController

ScrapingController pełni funkcję automatycznego pobierania i konwersji plików promocyjnych. Ścieżka bazowa to /Scraping. Metoda *DownloadPdfs*, dostępna pod *GET /Scraping/download-pdfs*, wykorzystuje serwis *IStoreService* do pozyskania listy

sklepów, a następnie dla każdego wywołuje *ScrappPromotionData* serwisu *IPdfHandlerService*.

```
public async Task<IActionResult> DownloadPdfs()
{
    try
    {
        var shopsList = await _storeService.GetAllStemsAsync();

        if (shopsList == null || shopsList.Count == 0)
        {
            _logger.LogWarning("No shops available to download PDFs.");
            return NotFound("No shops available to download PDFs.");
        }

        foreach (var shop in shopsList)
        {
            await _pdfHandlerService.ScrappPromotionData(shop);
        }

        _logger.LogInformation("PDF download process initiated.");
        return Ok("PDF download process initiated.");
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "An error occurred while downloading PDFs.");
        return StatusCode(StatusCodes.Status500InternalServerError, "An error occurred while downloading PDFs.");
    }
}
```

Rys. 7.9 Metoda DownloadPdfs

Dzięki logowaniu możliwe jest śledzenie braku dostępnych sklepów (404) czy błędów (500). W kolejnej akcji DownloadImages pod GET /Scraping/convert-to-images uruchamiana jest konwersja wszystkich pobranych PDF-ów na obrazy z użyciem metody ConvertAllPdfsToImagesAndDelete, a wynik operacji komunikowany jako tekstowy komunikat w odpowiedzi 200 lub – w razie problemów – 500 .

7.7. UserPanelController

Ostatni z omawianych kontrolerów, *UserPanelController*, służy do obsługi działań użytkownika po uwierzytelnieniu. Oznaczony atrybutem *[Authorize]* i zmapowany pod

ścieżką *api/UserPanel*, udostępnia trzy akcje: *GetFavouritePromotions, AddFavourite* oraz *RemoveFavourite*.

```
public async Task<IActionResult> GetFavouritePromotions()
{
    // Pobranie identyfikatora użytkownika z tokena (ClaimTypes.NameIdentifier)
    var userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
    if (string.IsNullOrEmpty(userId))
    {
        return Unauthorized();
    }

    var promotions = await _userPanelService.GetFavouritePromotionsAsync(userId);
    return Ok(promotions);
}

[HttpPost]
Odwołania: 0
public async Task<IActionResult> AddFavourite([FromBody] FavouriteRequest request)
{
    var userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
    if (string.IsNullOrEmpty(userId))
    {
        return Unauthorized();
    }

    await _userPanelService.AddFavouriteAsync(userId, request.PromotionId);
    return Ok(new { Message = "Promocja została dodana do ulubionych." });
}

[HttpDelete("favourite-promotions/{promotionId:guid}")]
Odwołania: 0
public async Task<IActionResult> RemoveFavourite(Guid promotionId)
{
    var userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
    if (string.IsNullOrEmpty(userId))
    {
        return Unauthorized();
    }

    try
    {
        await _userPanelService.RemoveFavouriteAsync(userId, promotionId);
    }
    catch (Exception ex)
    {
        return NotFound(new { Message = ex.Message });
    }

    return Ok(new { Message = "Promocja została usunięta z ulubionych." });
}
```

Rys. 7.10 Metody Metody GetFavouritePromotions, AddFavourite oraz RemoveFavourite

Wszystkie trzy metody pobierają identyfikator użytkownika z tokena JWT (klaima *ClaimTypes.NameIdentifier*). Metoda *GetFavouritePromotions* zwraca listę ulubionych promocji użytkownika w formie DTO. Metoda *AddFavourite* umożliwia dodanie nowej promocji do ulubionych, zwracając komunikat potwierdzający powodzenie operacji, natomiast metoda *RemoveFavourite* służy do usuwania promocji z ulubionych, również zwracając stosowny komunikat potwierdzający. W przypadku braku identyfikatora użytkownika lub jego niepoprawności wszystkie

metody zwracają odpowiedź *HTTP 401 Unauthorized*. Dodatkowo, metoda *RemoveFavourite* zwraca komunikat HTTP *404 Not Found* w przypadku niepowodzenia operacji.

8. Usługi

8.1. CategoryService

W serwisie *CategoryService* wstrzykiwany jest kontekst bazy danych, co pozwala na bezpośrednie operacje CRUD na encjach kategorii. W konstruktorze przypisywana jest referencja do *ApplicationDbContext*, dzięki czemu wszystkie metody pracują na wspólnym obiekcie kontekstu.

Kluczową rolę pełni metoda *GetAllCategoriesListAsync*, wykorzystywana później przez OpenAIService do dynamicznego budowania promptów. Używa ona wyłącznie projekcji nazw kategorii, co minimalizuje transfer danych.

```
public async Task<List<string>> GetAllCategoriesListAsync() // wykorzystane w OpenAIService
{
    return await _context.Categories
        .Select(category => category.Name)
        .ToListAsync();
}
```

Rys. 8.1 Metoda GetAllCategoriesListAsync

Przedstawiony sposób pobierania danych zapewnia efektywne wykorzystanie EF Core i SQL Server, ograniczając do minimum ilość przesyłanych kolumn .

Metoda *GetCategoryIdFromCategoryNameAsync* została zoptymalizowana przez dodanie *AsNoTracking*(), co wyłącza śledzenie encji przez kontekst i zmniejsza narzut pamięciowy przy odczytach, jednocześnie rzucając *KeyNotFoundException* w przypadku braku kategorii:

```
public async Task<CategoryDto> GetCategoryByIdAsync(Guid id)
{
    var category = await _context.Categories.FindAsync(id);
    if (category == null)
    {
        throw new KeyNotFoundException("Category not found.");
    }

    return new CategoryDto { Id = category.Id, Name = category.Name };
}
```

Rys. 8.2 Metoda GetCategoryIdFromCategoryNameAsync

Dzięki temu, w przypadku nieznalezienia rekordu, wyżej położona warstwa może precyzyjnie obsłużyć wyjątek, np. zwracając kod 404 do klienta .

8.2. LeafletService

LeafletService odpowiada za parsowanie dat i zarządzanie encjami ulotek w bazie. Konstruktor wstrzykuje *IStoreService*, by na podstawie aliasu sklepu uzyskać jego identyfikator.

Metoda *CreateLeafletAsync* rozdziela łańcuch zawierający zakres dat, waliduje format, a następnie konwertuje początku i końce dni na precyzyjne znaczniki czasowe („00:00:00” i „23:59:59.9999999”). Całość zapisywana jest jako nowy rekord, a wygenerowany Guid zwracany kontrolerowi

```
public async Task<LeafletDto> CreateLeafletAsync(LeafletDto leafletDto)
{
    var leaflet = new Leaflet
    {
        StartDate = leafletDto.StartDate,
        EndDate = leafletDto.EndDate,
        PdfLink = leafletDto.PdfLink,
        StoreId = leafletDto.StoreId
    };

    _context.Leaflets.Add(leaflet);
    await _context.SaveChangesAsync();

    leafletDto.Id = leaflet.Id;
    return leafletDto;
}
```

Rys. 8.3 Metoda CreateLeafletAsync

Dzięki temu kontroler nie musi zajmować się logiką parsowania dat ani walidacji formatu wejściowego .

8.3. PromotionService

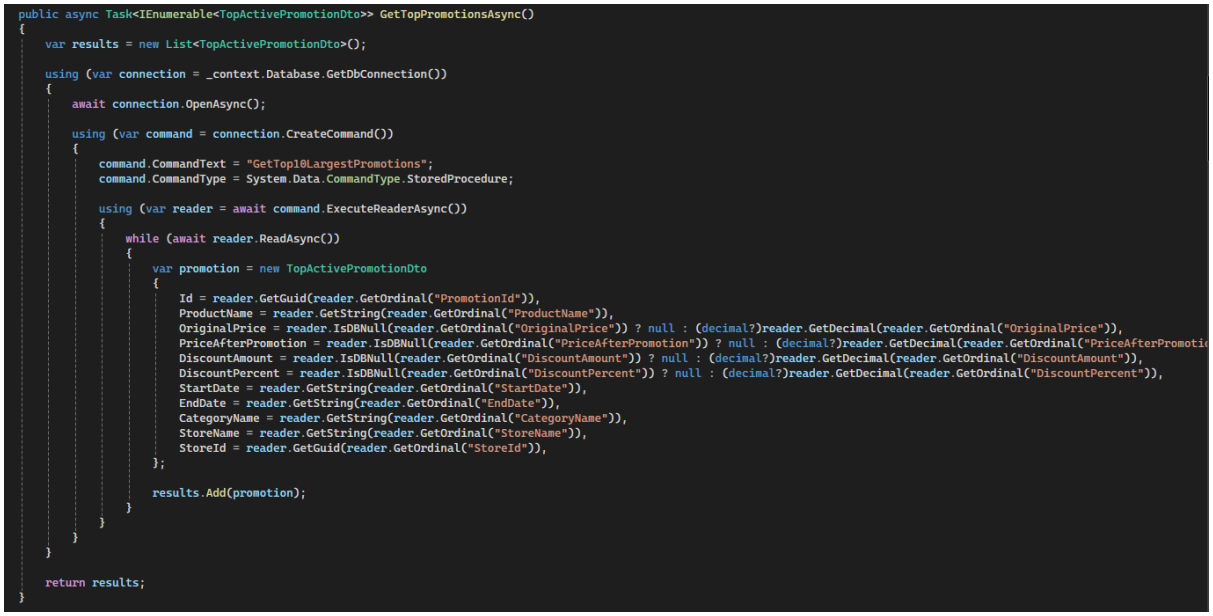
PromotionService agreguje promocje z kilku źródeł. Metody oparte na LINQ (*GetAllPromotionsAsync*, *GetActivePromotionsAsync*) projektują wyniki bezpośrednio do DTO, co eliminuje potrzeby mapowania w kontrolerze i zapobiega ładowaniu niepotrzebnych właściwości:

```
}
public async Task<IEnumerable<PromotionDto>> GetAllPromotionsAsync()
{
    return await _context.Promotions
        .Select(promotion => new PromotionDto
        {
            Id = promotion.Id,
            ProductName = promotion.ProductName,
            UnitType = promotion.UnitType,
            OriginalPrice = promotion.OriginalPrice,
            PriceAfterPromotion = promotion.PriceAfterPromotion,
            PromotionType = promotion.PromotionType,
            StartDate = promotion.StartDate,
            EndDate = promotion.EndDate,
            UntilOutOfStock = promotion.UntilOutOfStock,
            RequiredApp = promotion.RequiredApp,
            CategoryId = promotion.CategoryId,
            LeafletId = promotion.LeafletId
        })
        .ToListAsync();
}

public async Task<PromotionDto> GetPromotionByIdAsync(Guid id)
{
    var promotion = await _context.Promotions.FindAsync(id);
    if (promotion == null)
    {
        throw new KeyNotFoundException("Promotion not found.");
    }

    return new PromotionDto
    {
        Id = promotion.Id,
        ProductName = promotion.ProductName,
        UnitType = promotion.UnitType,
        OriginalPrice = promotion.OriginalPrice,
        PriceAfterPromotion = promotion.PriceAfterPromotion,
        PromotionType = promotion.PromotionType,
        StartDate = promotion.StartDate,
        EndDate = promotion.EndDate,
        UntilOutOfStock = promotion.UntilOutOfStock,
        RequiredApp = promotion.RequiredApp,
        CategoryId = promotion.CategoryId,
        LeafletId = promotion.LeafletId
    };
}
```

Rys. 8.4 Metody GetAllPromotionsAsync, GetActivePromotionsAsync

Takie rozwiązanie pozwala na jednoprzebiegowe zapytanie, łączące trzy tabele w SQL i przekazujące gotowe obiekty do klienta .

Dla bardziej zaawansowanych filtrów, jak pobranie top 10 promocji lub promocji według sklepu/kategorii, używane są procedury składowane wraz z DbConnection i SqlCommand.



```
public async Task<IEnumerable<TopActivePromotionDto>> GetTopPromotionsAsync()
{
    var results = new List<TopActivePromotionDto>();

    using (var connection = _context.Database.GetDbConnection())
    {
        await connection.OpenAsync();

        using (var command = connection.CreateCommand())
        {
            command.CommandText = "GetTop10LargestPromotions";
            command.CommandType = System.Data.CommandType.StoredProcedure;

            using (var reader = await command.ExecuteReaderAsync())
            {
                while (await reader.ReadAsync())
                {
                    var promotion = new TopActivePromotionDto
                    {
                        Id = reader.GetGuid(reader.GetOrdinal("PromotionId")),
                        ProductName = reader.GetString(reader.GetOrdinal("ProductName")),
                        OriginalPrice = reader.IsDBNull(reader.GetOrdinal("OriginalPrice")) ? null : (decimal?)reader.GetDecimal(reader.GetOrdinal("OriginalPrice")),
                        PriceAfterPromotion = reader.IsDBNull(reader.GetOrdinal("PriceAfterPromotion")) ? null : (decimal?)reader.GetDecimal(reader.GetOrdinal("PriceAfterPromoti
                        DiscountAmount = reader.IsDBNull(reader.GetOrdinal("DiscountAmount")) ? null : (decimal?)reader.GetDecimal(reader.GetOrdinal("DiscountAmount")),
                        DiscountPercent = reader.IsDBNull(reader.GetOrdinal("DiscountPercent")) ? null : (decimal?)reader.GetDecimal(reader.GetOrdinal("DiscountPercent")),
                        StartDate = reader.GetString(reader.GetOrdinal("StartDate")),
                        EndDate = reader.GetString(reader.GetOrdinal("EndDate")),
                        CategoryName = reader.GetString(reader.GetOrdinal("CategoryName")),
                        StoreName = reader.GetString(reader.GetOrdinal("StoreName")),
                        StoreId = reader.GetGuid(reader.GetOrdinal("StoreId")),
                    };

                    results.Add(promotion);
                }
            }
        }
    }

    return results;
}
```

Rys. 8.5 Wyznaczenie Top 10 aktywnych promocji

8.4. StoreService

W *StoreService* wstrzykiwany jest *IConfiguration*, z którego pobierana jest lista aliasów sklepów. Dzięki temu początkowa konfiguracja nie jest na stałe zakodowana w aplikacji, a w pliku *appsettings.json*. Metoda *CreateStoresFromConfAsync* formatuje aliasy na czytelne nazwy, sprawdza ich unikalność i następnie zapisuje.

```
public async Task CreateStoresFromConfAsync() //jednorazowe wprowadzenie sklepów do bazy
{
    foreach (var shop_stem in _shopsList)
    {
        try
        {
            var shopParts = shop_stem.Split('-');

            for (int i = 0; i < shopParts.Length; i++)
            {
                if (shopParts[i].Length > 0)
                {
                    var firstChar = char.ToUpper(shopParts[i][0]);
                    var rest = shopParts[i].Substring(1).ToLower();
                    shopParts[i] = firstChar + rest;
                }
            }

            var shop_name = string.Join(" ", shopParts);

            if (await _context.Stores.AnyAsync(s => s.Name.ToLower() == shop_name.ToLower()))
            {
                throw new InvalidOperationException($"Store '{shop_name}' already exists!");
            }

            var store = new Store
            {
                Name = shop_name,
                Stem = shop_stem
            };

            _context.Stores.Add(store);
            await _context.SaveChangesAsync();

            Console.WriteLine($"Added store: {shop_name}");
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine(ex.Message);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Unexpected error for store '{shop_stem}': {ex.Message}");
        }
    }
```

Rys. 8.6 Metoda CreateStoresFromConfAsyn

8.5. PdfHandlerService

PdfHandlerService łączy w sobie pobieranie, przetwarzanie i konwersję plików PDF. W statycznym konstruktorze ustawiana jest ścieżka do *Ghostscript*, co jest wymagane przez Magick.NET. Obiekt *HttpClient* tworzony jest raz na cały serwis, by unikać wyczerpania gniazd TCP.

Metoda *ScrappPromotionData* korzysta z *HtmlAgilityPack* do znalezienia przycisków pobierania PDF oraz odpowiadających im tekstów z datami, po czym deleguje zapis do *DownloadAndSavePdf*:

Następnie *ConvertAllPdfsToImagesAndDelete* wykonuje równoległą konwersję wszystkich PDF-ów na obrazy z wysoką rozdzielczością (300 DPI) oraz natychmiastowym usunięciem oryginałów:

```
public void ConvertAllPdfsToImagesAndDelete()
{
    var pdfFiles = Directory.GetFiles(_pdfDirectory, "*.pdf", SearchOption.AllDirectories);

    Parallel.ForEach(pdfFiles, pdfFile =>
    {
        try
        {
            Console.WriteLine($"Converting {pdfFile}");
            ConvertPdfToImages(pdfFile);


            File.Delete(pdfFile);
            Console.WriteLine($"Deleted {pdfFile}");
        }

        catch (Exception ex)
        {
            Console.WriteLine($"Error processing PDF: {ex.Message} with {pdfFile}");
        }
    });
}
```

Rys. 8.7 Metoda ConvertAllPdfsToImagesAndDelete

8.6. OpenAIService

W konstruktorze *OpenAIService* inicjalizowany jest klient ChatClient z kluczem API i modelem gpt-4o. Bezpośrednio sprawdzana jest obecność katalogu ze zdjęciami, co zapobiega późniejszym błędom.

Metoda *ParseImagesToFunction* iteruje po folderach nazwanych GUID, wczytuje każdą grafikę PNG i wywołuje prywatną metodę *GetJsonPromotions*, w której obraz jest przesyłany jako *BinaryData*. Prompt zawiera dynamicznie wstawioną listę kategorii pobraną z *CategoryService*,.

8.7. Scheduler

Klasa *Scheduler*, implementująca *IInvocable* z pakietu *Coravel*, orkiestruje cały cykl aktualizacji danych. W konstruktorze wstrzykiwane są niezbędne serwisy, a metodzie *Start* sprawdzana jest lista sklepów, po czym kolejno wywoływane są: pobranie PDF-ów, konwersja do obrazów i analiza przez OpenAI

```
public class Scheduler : IInvocable
{
    private readonly IPdfHandlerService _pdfHandlerService;
    private readonly IOpenAIService _openAIService;
    private readonly IStoreService _storeService;

    public Scheduler(IConfiguration configuration, IPdfHandlerService pdfHandlerService, IOpenAIService openAIService, IStoreService storeService)
    {
        _pdfHandlerService = pdfHandlerService ?? throw new ArgumentNullException(nameof(pdfHandlerService));
        _openAIService = openAIService ?? throw new ArgumentNullException(nameof(openAIService));
        _storeService = storeService ?? throw new ArgumentNullException(nameof(storeService));
    }

    public async Task Invoke()
    {
        await Start();
    }

    // stores i categories jednorazowo wprowadzić do bazy

    public async Task Start()
    {
        var shopsList = await _storeService.GetAllStemsAsync();

        if (shopsList == null || shopsList.Count == 0)
        {
            Console.WriteLine("No shops found. Aborting the task.");
            return;
        }
        foreach (var shop in shopsList)
        {
            await _pdfHandlerService.ScrappPromotionData(shop);  // ściąga pdfy gazetek i parsuje Leaflets do bazy
        }
        _pdfHandlerService.ConvertAllPdfsToImagesAndDelete(); // konwertuje pdfy na zdjęcia i usuwa pdfy
        await _openAIService.ParseImagesToFunction(); //parsuje zdjęcia do promptów openai i wywołuje funkcje parsującą Promotions do bazy i usuwa przeanalizowane zdjęcia
    }
}
```

Rys. 8.8 Kod Scheduler

Dzięki temu wystarczy zaplanować wywołanie *Scheduler.Invoke* w harmonogramie, by regularnie odświeżać bazę promocji bez ręcznej ingerencji.

8.8. UserPanelService

W UserPanelService wykorzystywany jest wzorzec „Include–ThenInclude" EF Core, by uniknąć problemu N+1 zapytań i zwrócić pełne obiekty promocji wraz z kategorią, ulotką oraz powiązanym sklepem w jednym zapytaniu. Metoda GetFavouritePromotionsAsync zwraca dane w formie DTO ActivePromotionDto, zawierające szczegóły takie jak nazwa produktu, typ jednostki, oryginalna cena, cena po promocji, rodzaj promocji, daty ważności oraz informacje o wymaganej aplikacji, nazwie kategorii i sklepu.

```
public async Task<IEnumerable<ActivePromotionDto>> GetFavouritePromotionsAsync(string userId)
{
    var favouritePromotions = await _context.Favourites
        .Where(f => f.UserId == userId)
        .Include(f => f.Promotion)
            .ThenInclude(p => p.Category)
        .Include(f => f.Promotion)
            .ThenInclude(p => p.Leaflet)
                .ThenInclude(l => l.Store)
        .Select(f => new ActivePromotionDto
        {
            Id = f.Promotion.Id,
            ProductName = f.Promotion.ProductName,
            UnitType = f.Promotion.UnitType,
            OriginalPrice = f.Promotion.OriginalPrice,
            PriceAfterPromotion = f.Promotion.PriceAfterPromotion,
            PromotionType = f.Promotion.PromotionType,
            StartDate = f.Promotion.StartDate ?? f.Promotion.Leaflet.StartDate,
            EndDate = f.Promotion.EndDate ?? f.Promotion.Leaflet.EndDate,
            UntilOutOfStock = f.Promotion.UntilOutOfStock,
            RequiredApp = f.Promotion.RequiredApp,
            CategoryName = f.Promotion.Category.Name,
            StoreName = f.Promotion.Leaflet.Store.Name
        })
        .ToListAsync();

    return favouritePromotions;
}
```

Rys. 8.9 Metoda GetPavouritePromotionAsync

Metoda *AddFavouriteAsync* najpierw weryfikuje istnienie rekordu, a w przypadku próby dodania duplikatu po prostu zwraca, zachowując idempotentność działania. Dodatkowo, metoda ta sprawdza istnienie promocji o podanym identyfikatorze, a brak takiego rekordu skutkuje wyjątkiem z odpowiednim komunikatem. Metoda *RemoveFavouriteAsync* usuwa wybraną promocję z ulubionych, a jeśli dana promocja nie istnieje w ulubionych użytkownika, również rzuca wyjątek z odpowiednim komunikatem.

```
    public async Task AddFavouriteAsync(string userId, Guid promotionId)
    {
        var exists = await _context.Favourites.AnyAsync(f => f.UserId == userId && f.PromotionId == promotionId);
        if (exists)
        {
            return;
        }

        var promotionExists = await _context.Promotions.AnyAsync(p => p.Id == promotionId);
        if (!promotionExists)
        {
            throw new Exception("Nie znaleziono promocji o podanym Id.");
        }

        var favourite = new Favourite
        {
            UserId = userId,
            PromotionId = promotionId
        };

        _context.Favourites.Add(favourite);
        await _context.SaveChangesAsync();
    }

    Odwołania: 0
    public async Task RemoveFavouriteAsync(string userId, Guid promotionId)

var favourite = await _context.Favourites
    .FirstOrDefaultAsync(f => f.UserId == userId && f.PromotionId == promotionId);

if (favourite == null)
{
    throw new Exception("Nie znaleziono takiej promocji w ulubionych.");
}

_context.Favourites.Remove(favourite);
await _context.SaveChangesAsync();

}
```

Rys. 8.10 Metody AddFavouriteAsync oraz RemoveFavouriteAsync

9. Wprowadzenie do aplikacji klienckiej PromoComparer

9.1. Opis aplikacji

Aplikacja kliencka PromoComparer stanowi interaktywny interfejs użytkownika dla systemu PromoComparer, umożliwiającego efektywne przeglądanie aktualnych ofert promocyjnych pochodzących z różnych sieci handlowych. Głównym zadaniem aplikacji frontendowej jest prezentacja danych promocyjnych w atrakcyjnej i intuicyjnej formie, zapewniającej użytkownikom możliwość szybkiego filtrowania ofert, wyszukiwania interesujących promocji oraz personalizacji poprzez zarządzanie ulubionymi promocjami.

9.2. Technologie wykorzystane w projekcie

Do realizacji aplikacji frontendowej PromoComparer wykorzystano następujące technologie:

- **React.js** – biblioteka JavaScript do budowania dynamicznych i responsywnych interfejsów użytkownika.
- **JavaScript (ES6+)** – podstawowy język programowania służący do implementacji logiki aplikacyjnej.
- **Material UI (@mui)** – framework komponentów React, zapewniający spójny design, szybkość wdrożenia oraz gotowe do użycia komponenty interfejsu.
- **Axios** – biblioteka wykorzystywana do realizacji komunikacji z backendem za pomocą HTTP, obsługująca RESTful API.
- **React Context oraz React Hooks** – mechanizmy pozwalające na zarządzanie stanem globalnym oraz lokalnym w aplikacji.

9.3. Podstawowe funkcjonalności

Najważniejsze funkcjonalności aplikacji PromoComparer obejmują:

- Wyświetlanie szczegółowych danych o aktualnych promocjach (nazwa produktu, ceny, typy rabatów, terminy obowiązywania).
- Możliwość filtrowania ofert według sklepów oraz kategorii produktowych.
- Wyszukiwanie promocji na podstawie słów kluczowych wpisywanych przez użytkownika.
- Autoryzacja użytkowników – rejestracja, logowanie oraz zarządzanie sesją użytkownika.

- Personalizacja interfejsu użytkownika poprzez zarządzanie ulubionymi promocjami.
- Adaptacyjność interfejsu do różnych rozmiarów ekranów – pełna responsywność zarówno na urządzeniach desktopowych, jak i mobilnych.

9.4. Integracja z backendem

Komunikacja z warstwą backendową, bazującą na technologii ASP.NET Core, realizowana jest za pomocą RESTful API. Każda akcja wykonywana przez użytkownika aplikacji frontendowej (np. logowanie, filtrowanie ofert czy zarządzanie ulubionymi promocjami) powoduje wysłanie odpowiedniego żądania HTTP do serwera backendowego, który zwraca dane w formacie JSON.

9.5. Struktura projektu

Struktura aplikacji frontendowej została zaprojektowana zgodnie z zasadą Clean Architecture, dzięki czemu zapewniona jest modularność, łatwość utrzymania kodu oraz skalowalność aplikacji. Wyróżniono następujące warstwy logiczne:

- **Domain** – definicja modeli danych używanych w aplikacji, takich jak promocje, sklepy czy kategorie.
- **Application** – warstwa implementująca logikę biznesową aplikacji oraz interakcję z API.
- **Infrastructure** – warstwa techniczna odpowiedzialna za komunikację HTTP, zarządzanie sesjami użytkowników oraz integrację z lokalną pamięcią przeglądarki.
- **Presentation** – komponenty interfejsu użytkownika, strony aplikacji oraz obsługa zdarzeń interakcji użytkowników.
- **Utilities** – dodatkowe narzędzia wspomagające działanie aplikacji, np. formatowanie danych czy operacje na datach.

10. Architektura aplikacji frontendowej

Aplikacja PromoComparer, realizująca funkcjonalność klienta webowego, została zaprojektowana w oparciu o **architekturę warstwową**, zgodną z paradygmatem Clean Architecture. Struktura projektu gwarantuje rozdział odpowiedzialności, modularność i wysoką czytelność kodu, co przekłada się na łatwość utrzymania i możliwość dalszego rozwoju.

10.1. Podział na warstwy

Projekt podzielono na pięć głównych obszarów logicznych:

- **Domain**
  Warstwa domenowa obejmuje definicje modeli danych używanych w całej aplikacji. Modele te są strukturami typu „czyste dane” (DTO), bez logiki biznesowej, wykorzystywanymi do transferu informacji pomiędzy komponentami oraz przy wymianie danych z backendem. Przykładami są: model promocji (`promotion.js`), sklepu (`shop.js`), kategorii (`category.js`) czy użytkownika (`user.js`).
- **Application**
  Warstwa aplikacyjna odpowiedzialna jest za implementację logiki biznesowej klienta – w tym komunikację z API, agregację danych oraz ich wstępne przetwarzanie przed przekazaniem do warstwy prezentacyjnej. Obejmuje serwisy odpowiadające za poszczególne obszary funkcjonalne (np. obsługę promocji, kategorii, sklepów, użytkownika), a także dedykowane hooki React realizujące pobieranie oraz synchronizację danych.
- **Infrastructure**
  W warstwie infrastrukturalnej znajdują się moduły odpowiedzialne za niskopoziomowe aspekty techniczne, takie jak wykonywanie żądań HTTP (`HttpClient.js`), automatyczna obsługa tokenów autoryzacyjnych (`AuthInterceptor.js`), czy integracja z lokalną pamięcią przeglądarki (`LocalStorageAdapter.js`). Warstwa ta odseparowuje szczegóły techniczne od wyższych poziomów aplikacji.
- **Presentation**
  Warstwa prezentacyjna zawiera komponenty interfejsu użytkownika – zarówno proste elementy (przyciski, karty), jak i złożone sekcje oraz całe strony aplikacji. Odpowiada również za obsługę zdarzeń użytkownika, zarządzanie nawigacją oraz prezentację danych pozyskanych z niższych warstw. Kluczowe są tu komponenty typu „container”, które integrują logikę biznesową z UI.
- **Utilities**
  Warstwa narzędziowa skupia uniwersalne funkcje pomocnicze, wykorzystywane w całej aplikacji – na przykład do formatowania dat, walidacji czy konwersji danych.

10.2. Przepływ danych

Proces przepływu danych w aplikacji opiera się na czytelnym łańcuchu zależności:

1. **Komponent prezentacyjny** zgłasza zapotrzebowanie na dane (np. lista promocji).
2. **Dedykowany hook** wywołuje odpowiedni serwis z warstwy aplikacyjnej.
3. **Serwis** realizuje żądanie HTTP przez klienta z warstwy infrastrukturalnej.
4. **Odpowiedź** z backendu trafia z powrotem do hooka, gdzie następuje wstępne przetworzenie danych i przekazanie ich do komponentu UI.
5. **Komponent prezentacyjny** aktualizuje stan interfejsu i wyświetla dane użytkownikowi.

Każda warstwa spełnia jasno określoną funkcję, eliminując zjawisko „przeciekającej logiki" i gwarantując możliwość testowania oraz łatwego refaktoryzowania kodu.

10.3. Zasady projektowe

- **Separation of Concerns** – każda warstwa posiada własny zakres odpowiedzialności i nie narusza domeny pozostałych warstw.
- **Reużywalność** – komponenty i serwisy są zaprojektowane jako uniwersalne, łatwe do ponownego wykorzystania w innych częściach systemu.
- **Skalowalność** – modularna struktura umożliwia łatwe dodawanie nowych funkcjonalności bez ryzyka regresji.

11. Warstwa domenowa

Warstwa domenowa aplikacji PromoComparer zawiera modele danych, które są podstawą wymiany informacji pomiędzy warstwami aplikacji frontendowej oraz komunikacji z backendem. Modele te zostały zaprojektowane jako proste struktury danych, odwzorowujące najważniejsze encje systemowe, bez nadmiarowej logiki biznesowej.

11.1. Struktura modeli domenowych

W katalogu `src/domain` znajdują się następujące definicje modeli:

- **promotion.js**
  Definiuje strukturę danych reprezentującą pojedynczą promocję. Kluczowe pola obejmują: identyfikator promocji, nazwę produktu, cenę oryginalną i promocyjną, typ promocji, jednostkę sprzedaży, daty obowiązywania, powiązanie z gazetką, sklepem oraz kategorią.
- **shop.js**
  Model sklepu wykorzystywany do prezentacji dostępnych sieci handlowych w aplikacji. Zawiera takie atrybuty jak: identyfikator sklepu, nazwę oraz powiązaną listę gazetek promocyjnych.
- **category.js**
  Model kategorii produktowej pozwala klasyfikować oferty promocyjne. Obejmuje identyfikator, nazwę oraz listę powiązanych promocji.
- **user.js**
  Struktura użytkownika systemu, wykorzystywana przy obsłudze funkcji personalizacji oraz autoryzacji. Kluczowe pola obejmują: identyfikator użytkownika, nazwę (login), listę ulubionych promocji i pozostałe dane powiązane z kontem.

11.2. Rola modeli domenowych

Modele te wykorzystywane są:

- Do przesyłania danych pomiędzy warstwami aplikacji frontendowej (np. z serwisów do komponentów prezentacyjnych).
- W komunikacji z backendem podczas realizacji żądań HTTP (przy serializacji/deserializacji danych w formacie JSON).
- Jako podstawa do mapowania danych otrzymanych z API na wewnętrzne struktury aplikacji.

Każdy z modeli został zaprojektowany w sposób umożliwiający łatwą rozbudowę o dodatkowe pola lub powiązania – bez konieczności przebudowy pozostałych warstw aplikacji.

12. Warstwa aplikacyjna

Warstwa aplikacyjna odpowiada za obsługę logiki biznesowej po stronie klienta oraz za pośrednictwo w wymianie danych między komponentami prezentacyjnymi a backendem. Podstawę tej warstwy stanowią serwisy oraz dedykowane hooki React, które zapewniają spójność i aktualność prezentowanych informacji.

12.1. Serwisy aplikacyjne

Serwisy znajdujące się w katalogu `src/application/services` zostały zaprojektowane zgodnie z zasadą pojedynczej odpowiedzialności. Każdy serwis odpowiada za komunikację z wybranym zakresem API i udostępnia metody umożliwiające wykonywanie operacji na określonym zasobie.

Przykład implementacji metody pobierającej aktywne promocje:

```
1  // src/application/services/PromotionService.js
2  import HttpClient from '../../infrastructure/HttpClient';
3
4  const PromotionService = {
5    getActive() {
6      return HttpClient.get('/api/Promotions/active');
7    },
```

Rys. 12.1. Fragment kodu metody pobierającej wszystkie aktywne promocje w serwisie PromotionService.js.

Serwisy pełnią rolę pośrednika – nie przechowują stanu, a jedynie realizują żądania do backendu oraz przekazują odpowiedzi do warstwy wyżej.

12.2. Hooki aplikacyjne

Dedykowane hooki React, zlokalizowane w katalogu `src/application/hooks`, ułatwiają zarządzanie cyklem życia danych, obsługę stanów ładowania i błędów oraz integrację logiki biznesowej z komponentami prezentacyjnymi.

Przykład użycia hooka do pobierania promocji:

```
// src/application/hooks/useData.js
import { useState, useEffect } from 'react';

/**
 * Custom hook for data fetching.
 * @param {Function} fetchFn - The function that returns a promise resolving to data.
 * @param {Array} deps - Dependency array controlling when to re-run fetch.
 */
export default function useData(fetchFn, deps = []) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    let mounted = true;
    setLoading(true);
    fetchFn()
      .then(result => {
        if (mounted) setData(result);
      })
      .catch(err => {
        if (mounted) setError(err);
      })
      .finally(() => {
        if (mounted) setLoading(false);
      });
    return () => { mounted = false; };
  }, deps);

  return { data, loading, error };
}

```

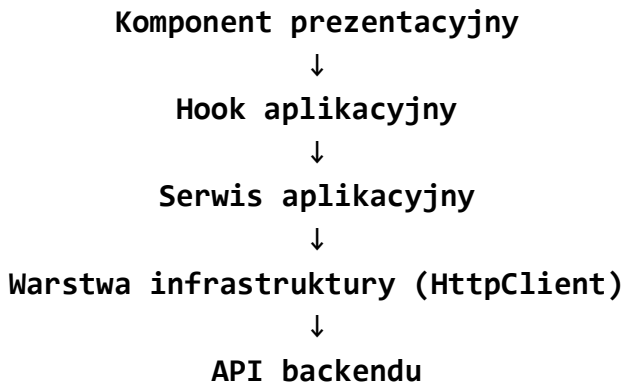
Rys. 12.2. Fragment hooka usePromotionData odpowiedzialnego za pobieranie i udostępnianie listy promocji.

Hooki zapewniają komponentom UI łatwy dostęp do danych oraz obsługę typowych przypadków, takich jak ładowanie, obsługa błędów czy odświeżanie stanu po zmianie kryteriów filtrowania.

12.3. Współpraca serwisów i hooków

Struktura warstwy aplikacyjnej opiera się na ścisłej współpracy serwisów i hooków. Każdy hook wywołuje konkretne metody serwisu, dzięki czemu warstwa prezentacyjna pozostaje odseparowana od detali dotyczących komunikacji z API i obsługi żądań HTTP.

Schemat zależności przedstawiono na poniższym diagramie:



Rys. 12.3. Schemat współpracy pomiędzy warstwami aplikacji frontendowej.

13. Warstwa infrastrukturalna

Warstwa infrastrukturalna odpowiada za realizację zaplecza technicznego aplikacji frontendowej. W jej skład wchodzą rozwiązania umożliwiające komunikację HTTP z backendem, obsługę autoryzacji użytkownika oraz integrację z lokalną pamięcią przeglądarki.

13.1. Klient HTTP

Podstawę komunikacji aplikacji z backendem stanowi własna implementacja klienta HTTP, znajdująca się w pliku `src/infrastructure/HttpClient.js`. Klient ten opiera się na mechanizmie fetch i jest rozszerzony o obsługę interceptorów (np. automatyczne dołączanie tokenu JWT oraz obsługa błędów autoryzacji).

```
5   class HttpClient {
6     constructor() {
7       this.baseUrl = process.env.REACT_APP_API_URL || 'http://localhost:5068';
8       this.interceptor = new AuthInterceptor(this);
9     }
```

Rys. 13.1. Konstruktor klasy HttpClient

Opis:
Klient inicjalizowany jest z domyślnym adresem backendu oraz instancją AuthInterceptora, co pozwala na automatyczną obsługę autoryzacji każdego zapytania HTTP.

```
29    get(p)    { return this.request(p, { method:'GET' }); }
30    post(p,b) { return this.request(p, { method:'POST', body:JSON.stringify(b)}); }
31    put(p,b)  { return this.request(p, { method:'PUT',  body:JSON.stringify(b)}); }
32    delete(p) { return this.request(p, { method:'DELETE' }); }
```

Rys. 13.2. Przykładowe metody klasy HttpClient

Opis:
Klasa HttpClient udostępnia standardowe metody REST do obsługi komunikacji z API, ukrywając szczegóły realizacji żądania oraz automatycznie wywołując interceptor.

13.2. Interceptor autoryzacji

Klasa AuthInterceptor (`src/infrastructure/AuthInterceptor.js`) odpowiada za automatyczne dołączanie tokenu autoryzacyjnego do każdego zapytania oraz obsługę błędów autoryzacji (status HTTP 401).

```
10    async intercept(options) {
11      const token = AuthService.getAccessToken();
12      const headers = { ...(options.headers||{}), 'Content-Type':'application/json' };
13      if (token) headers.Authorization = `Bearer ${token}`;
14      return { ...options, headers };
15    }
```

Rys. 13.3. Mechanizm dołączania tokenu JWT

Opis:
Przed wykonaniem żądania HTTP interceptor sprawdza, czy użytkownik jest zalogowany i w razie potrzeby dodaje nagłówek Authorization z aktualnym tokenem JWT.

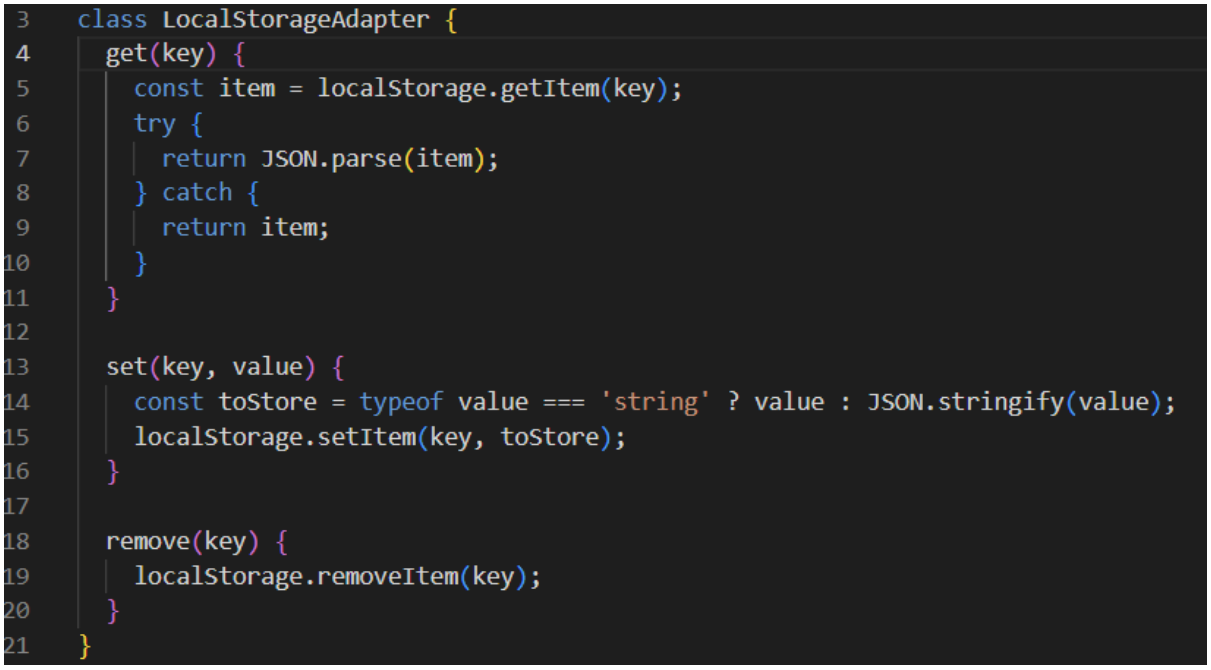
```
17    async handleResponseError(err, originalRequest) {
18      if (err.response?.status === 401) {
19        await AuthService.refresh();
20        const opts = await this.intercept(originalRequest);
21        return this.client.request(originalRequest.url, opts);
22      }
23      throw err;
24    }
```

Rys. 13.4. Automatyczne odświeżanie tokenu

Opis:
W przypadku napotkania błędu autoryzacji (401) interceptor wywołuje metodę odświeżania tokenu (`AuthService.refresh()`), a następnie powtarza żądanie z nowym tokenem.

13.3. Adapter do pamięci lokalnej

Klasa `LocalStorageAdapter` (`src/infrastructure/LocalStorageAdapter.js`) dostarcza zunifikowany interfejs do przechowywania, pobierania i usuwania danych w pamięci przeglądarki.



```
class LocalStorageAdapter {
  get(key) {
    const item = localStorage.getItem(key);
    try {
      return JSON.parse(item);
    } catch {
      return item;
    }
  }

  set(key, value) {
    const toStore = typeof value === 'string' ? value : JSON.stringify(value);
    localStorage.setItem(key, toStore);
  }

  remove(key) {
    localStorage.removeItem(key);
  }
}
```

Rys. 13.5. Przykład obsługi localStorage

Opis:
Adapter automatycznie serializuje i deserializuje dane do/z formatu JSON, dzięki czemu jest uniwersalny i wygodny w użyciu zarówno przez serwisy aplikacyjne, jak i komponenty frontendowe.

Warstwa infrastrukturalna została zaprojektowana tak, by była niezależna od pozostałych elementów systemu i mogła być łatwo wymieniana lub rozszerzana, bez wpływu na resztę kodu aplikacji.

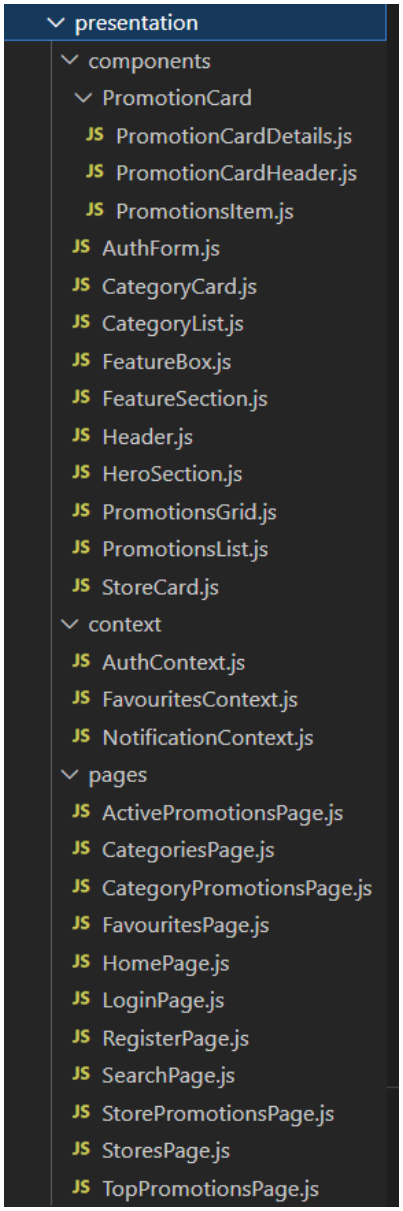
14. Warstwa prezentacyjna

Warstwa prezentacyjna obejmuje wszystkie komponenty odpowiedzialne za wizualizację danych, obsługę interakcji użytkownika oraz prezentację funkcjonalności systemu w formie graficznej. W projekcie PromoComparer strukturę tę tworzą zarówno drobne, wielorazowe komponenty, jak i kompletne strony, współpracujące ze sobą w ramach spójnej nawigacji i jednolitego stylu interfejsu.

14.1. Struktura katalogów

Komponenty warstwy prezentacyjnej zostały pogrupowane w podfoldery odpowiadające ich funkcji w aplikacji:

- `src/presentation/components` – zawiera podstawowe, wielorazowe komponenty interfejsu (np. karty promocji, listy, formularze).
- `src/presentation/components/PromotionCard` – rozwinięcie głównego komponentu wizualizującego pojedynczą promocję.
- `src/presentation/context` – konteksty React do globalnego zarządzania stanem aplikacji (autoryzacja, ulubione, powiadomienia).
- `src/presentation/pages` – kompletne strony aplikacji odpowiadające poszczególnym ścieżkom nawigacji.



Rys. 14.1. Struktura katalogów warstwy prezentacyjnej.

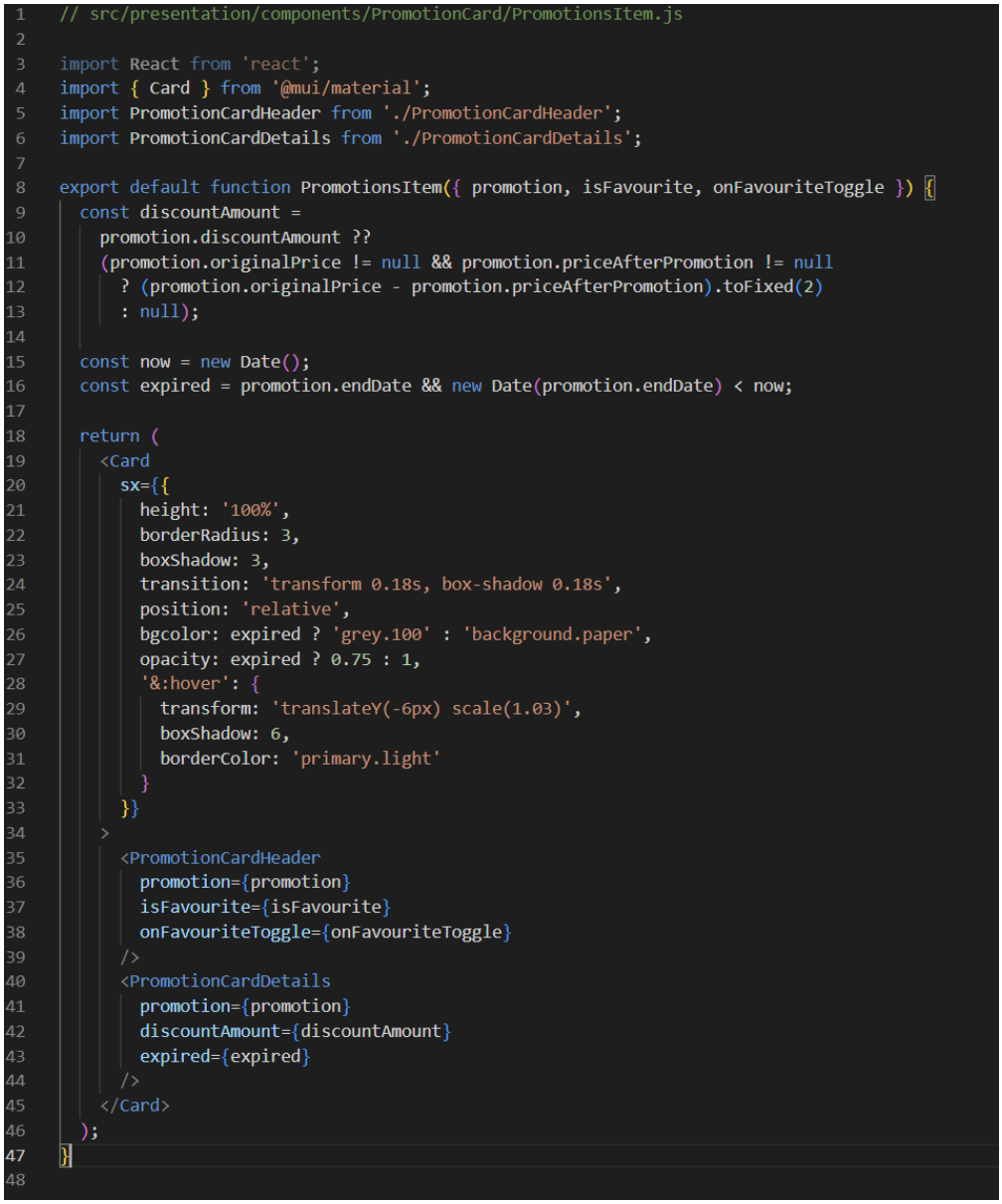
14.2. Komponent prezentacyjny pojedynczej promocji

Jednym z kluczowych komponentów warstwy prezentacyjnej jest **PromotionsItem** – odpowiadający za prezentację pojedynczej oferty promocyjnej w postaci karty. Komponent został zbudowany modułowo, co ułatwia jego rozbudowę oraz utrzymanie.

Struktura komponentu

PromotionsItem korzysta z dwóch wewnętrznych komponentów:

- **PromotionCardHeader** – odpowiada za nagłówek karty (nazwa produktu, sklep, status ulubionych, oznaczenia).
- **PromotionCardDetails** – prezentuje szczegóły oferty (ceny, rabaty, okres obowiązywania, status wyprzedaży).



```jsx
// src/presentation/components/PromotionCard/PromotionsItem.js

import React from 'react';
import { Card } from '@mui/material';
import PromotionCardHeader from './PromotionCardHeader';
import PromotionCardDetails from './PromotionCardDetails';

export default function PromotionsItem({ promotion, isFavourite, onFavouriteToggle }) {
  const discountAmount =
    promotion.discountAmount ??
    (promotion.originalPrice != null && promotion.priceAfterPromotion != null
      ? (promotion.originalPrice - promotion.priceAfterPromotion).toFixed(2)
      : null);

  const now = new Date();
  const expired = promotion.endDate && new Date(promotion.endDate) < now;

  return (
    <Card
      sx={{
        height: '100%',
        borderRadius: 3,
        boxShadow: 3,
        transition: 'transform 0.18s, box-shadow 0.18s',
        position: 'relative',
        bgcolor: expired ? 'grey.100' : 'background.paper',
        opacity: expired ? 0.75 : 1,
        '&:hover': {
          transform: 'translateY(-6px) scale(1.03)',
          boxShadow: 6,
          borderColor: 'primary.light'
        }
      }}
    >
      <PromotionCardHeader
        promotion={promotion}
        isFavourite={isFavourite}
        onFavouriteToggle={onFavouriteToggle}
      />
      <PromotionCardDetails
        promotion={promotion}
        discountAmount={discountAmount}
        expired={expired}
      />
    </Card>
  );
}
```

Rys. 14.2. Struktura komponentu PromotionsItem – podział na nagłówek oraz szczegóły.

PromotionCardHeader

Ten komponent prezentuje podstawowe informacje o promocji oraz umożliwia dodanie lub usunięcie oferty z ulubionych (widoczne tylko dla zalogowanego użytkownika).

```
9       <CardHeader
10        action={
11          user && (
12            <Tooltip title={isFavourite ? 'Usuń z ulubionych' : 'Dodaj do ulubionych'}>
13              <IconButton onClick={onFavouriteToggle}>
14                {isFavourite ? <Favorite color="error" /> : <FavoriteBorder />}
15              </IconButton>
16            </Tooltip>
17          )
18        }
```

Rys. 14.3. Fragment kodu odpowiadającego za prezentację statusu ulubionych.

PromotionCardDetails

Odpowiada za szczegółową prezentację informacji o promocji: aktualnej cenie, cenie przed rabatem, wartości i procencie zniżki, okresie obowiązywania oraz statusie wyprzedaży.

```jsx
export default function PromotionCardDetails({ promotion, discountAmount, expired }) {
  const start = formatDate(promotion.startDate);
  const end = formatDate(promotion.endDate);

  let okres;
  if (start && end) okres = `${start} – ${end}`;
  else if (start && !end) okres = `${start} – Brak informacji`;
  else if (!start && end) okres = `Brak informacji – ${end}`;
  else okres = 'Brak informacji';

  return (
    <CardContent sx={{ pt: 1, display: 'flex', flexDirection: 'column', gap: 1 }}>
      <Box sx={{ display: 'flex', alignItems: 'baseline', gap: 2, flexWrap: 'wrap' }}>
        {promotion.priceAfterPromotion != null && (
          <Typography variant="h5" color="primary" sx={{ fontWeight: 700 }}>
            {promotion.priceAfterPromotion.toFixed(2)} zł
          </Typography>
        )}
        {promotion.originalPrice != null && (
          <Typography
            variant="body2"
            sx={{
              textDecoration: 'line-through',
              color: 'text.disabled',
              fontWeight: 500,
              ml: 1
            }}
          >
            {promotion.originalPrice.toFixed(2)} zł
          </Typography>
        )}
        {discountAmount && (
          <Chip
            label={`-${discountAmount} zł`}
            color="success"
            size="small"
            sx={{ ml: 1, fontWeight: 600 }}
          />
        )}
        {promotion.discountPercent != null && (
          <Chip
            label={`-${promotion.discountPercent}%`}
            color="secondary"
            size="small"
            sx={{ ml: 1, fontWeight: 600 }}
          />
        )}
      </Box>
      <Typography variant="body2" color="text.secondary">
        <strong>Okres:</strong> {okres}
      </Typography>
      {promotion.untilOutOfStock !== undefined && (
        <Typography variant="body2" color="text.secondary">
          <strong>Do wyczerpania zapasów:</strong> {promotion.untilOutOfStock ? '✔' : '✖'}
        </Typography>
      )}
      {expired && (
        <Chip label="Zakończona" color="warning" sx={{ mt: 1, fontWeight: 600 }} />
      )}
    </CardContent>
  );
}

```

Rys. 14.4. Prezentacja szczegółów promocji: cena po rabacie, cena oryginalna, rabaty, okres obowiązywania.

Opisane wcześniej moduły to przykład na komponentów podstawowych oraz ich wykorzystania na rzecz agregacji pod komponentem złożonym.

Komponenty podstawowe realizują pojedyncze funkcje, takie jak prezentacja informacji o promocji (`PromotionCard`), kategorii (`CategoryCard`), sklepie (`StoreCard`), czy sekcje informacyjne (np. `FeatureSection`, `HeroSection`). Są one budulcem dla bardziej złożonych widoków.

Komponenty złożone (np. `PromotionsGrid`, `PromotionsList`, `CategoryList`) agregują podstawowe komponenty i służą do prezentowania list lub siatek danych pobieranych przez dedykowane hooki.

14.3. Konteksty aplikacyjne

Aplikacja wykorzystuje React Context do zarządzania globalnym stanem, umożliwiając komponentom dostęp do informacji o bieżącym użytkowniku, ulubionych promocjach czy aktualnych powiadomieniach bez konieczności przekazywania danych w dół drzewa komponentów.

Szczegółowa implementacja oraz mechanizmy zarządzania stanem globalnym zostały opisane w rozdziale 15.

15. Zarządzanie stanem aplikacji

W aplikacji PromoComparer globalny i lokalny stan jest utrzymywany oraz synchronizowany za pomocą **React Context** i dedykowanych hooków. Rozwiązanie to pozwala na centralizację logiki zarządzania danymi oraz wygodne udostępnianie ich w całym drzewie komponentów.

15.1. AuthContext — mechanizmy i implementacja

AuthContext umożliwia:

- Przechowywanie i aktualizację bieżącego użytkownika.
- Logowanie, rejestrację oraz wylogowywanie.
- Automatyczne odświeżanie tokenów dostępowych.

```jsx
// src/presentation/context/AuthContext.js

import React, { createContext, useContext, useState, useEffect } from 'react';
import AuthService from '../../application/services/AuthService';

const AuthContext = createContext({
  user: null,
  login: async () => {},
  logout: () => {}
});

export function AuthProvider({ children }) {
  const [user, setUser] = useState(AuthService.getCurrentUser());

  useEffect(() => {
    const unsubscribe = AuthService.onAuthChange(updatedUser => {
      setUser(updatedUser);
    });
    return unsubscribe;
  }, []);

  const login = async credentials => AuthService.login(credentials);
  const logout = () => AuthService.logout();

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}

export function useAuthContext() {
  return useContext(AuthContext);
}

```

Rys. 15.1. Implementacja kontekstu autoryzacji użytkownika.

15.2. FavouritesContext – zarządzanie ulubionymi promocjami

Do centralnego zarządzania listą ulubionych promocji służy FavouritesContext, którego implementacja znajduje się w pliku `src/presentation/context/FavouritesContext.js`.
Kontekst ten pozwala na pobieranie, dodawanie i usuwanie ulubionych ofert w całej aplikacji, niezależnie od głębokości zagnieżdżenia komponentu.

```
import React, { createContext, useContext, useEffect, useState, useCallback } from 'react';
import FavouritesService from '../../application/services/FavouritesService';
import useAuth from '../../application/hooks/useAuth';

const FavouritesContext = createContext();

export function FavouritesProvider({ children }) {
  const { user } = useAuth();
  const [favourites, setFavourites] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  useEffect(() => {
    if (!user) {
      setFavourites([]);
      setLoading(false);
      return;
    }
    setLoading(true);
    setError(null);
    FavouritesService.getAll()
      .then(setFavourites)
      .catch(setError)
      .finally(() => setLoading(false));
  }, [user]);
```

Rys. 15.2.1. Fragment inicjalizacji kontekstu i obsługa pobierania ulubionych po zalogowaniu użytkownika.

Opis:

- Po zalogowaniu użytkownika lista ulubionych jest pobierana z backendu za pomocą FavouritesService i przechowywana w stanie lokalnym kontekstu.
- Po wylogowaniu użytkownika lista jest czyszczona.

15.3. Zarządzanie stanem lokalnym

Na potrzeby zarządzania stanem lokalnym (np. stan ładowania, filtrowanie) wykorzystuje się hooki takie jak `useState`, `useEffect` oraz dedykowane hooki aplikacyjne (`usePromotionData`, `useShopData`, itd.).

```
29    const add = useCallback(async (id) => {
30      try {
31        const updated = await FavouritesService.add(id);
32        setFavourites(updated);
33      } catch (err) {
34        setError(err);
35      }
36    }, []);
37
38    const remove = useCallback(async (id) => {
39      try {
40        const updated = await FavouritesService.remove(id);
41        setFavourites(updated);
42      } catch (err) {
43        setError(err);
44      }
45    }, []);
```

Rys. 15.2.2. Asynchroniczne akcje dodawania i usuwania promocji z ulubionych.

Opis:

- Akcje `add` oraz `remove` odwołują się do backendu i po sukcesie aktualizują stan ulubionych.
- W przypadku błędu ustawiany jest stan `error`, który może zostać obsłużony w UI.

```
47    return (
48      <FavouritesContext.Provider value={{ favourites, loading, error, add, remove }}>
49        {children}
50      </FavouritesContext.Provider>
51    );
52  }
```

Rys. 15.2.3. Provider udostępnia całą logikę i aktualny stan dzieciom w drzewie komponentów.

FavouritesContext zapewnia dostęp do listy ulubionych promocji, stanu ładowania, obsługi błędów oraz akcji dodawania i usuwania ofert z ulubionych. Dzięki centralizacji tej logiki cały interfejs użytkownika może korzystać z tych samych danych, zachowując spójność i aktualność widoku.

Relatywnie do opisanego powyżej przypadku działa NotificationContext, czyli zarządzanie powiadomieniami.

Aby umożliwić wyświetlanie powiadomień systemowych i błędów w sposób centralny, aplikacja wykorzystuje kontekst powiadomień (NotificationContext), znajdujący się w pliku `src/presentation/context/NotificationContext.js`. Dzięki temu dowolny komponent w aplikacji może wywołać informację o sukcesie, ostrzeżenie lub błąd, które następnie zostanie globalnie wyświetlone użytkownikowi.

```jsx
// src/presentation/context/NotificationContext.js

import React, { createContext, useContext, useState, useCallback } from 'react';
import { v4 as uuidv4 } from 'uuid';

const NotificationContext = createContext({
  notifications: [],
  notify: (message, type) => {},
  removeNotification: id => {}
});

export function NotificationProvider({ children }) {
  const [notifications, setNotifications] = useState([]);

  const notify = useCallback((message, type = 'info') => {
    const id = uuidv4();
    setNotifications(prev => [...prev, { id, message, type }]);
    setTimeout(() => {
      setNotifications(prev => prev.filter(n => n.id !== id));
    }, 5000);
  }, []);

  const removeNotification = useCallback(id => {
    setNotifications(prev => prev.filter(n => n.id !== id));
  }, []);

  return (
    <NotificationContext.Provider value={{ notifications, notify, removeNotification }}>
      {children}
    </NotificationContext.Provider>
  );
}

export function useNotificationContext() {
  return useContext(NotificationContext);
}

```

Rys. 15.2.3. Realny wygląd NotificationContext.js

16. Zarządzanie stanem aplikacji

Stan aplikacji PromoComparer jest zarządzany wielopoziomowo, z wykorzystaniem mechanizmów **React Context** oraz dedykowanych hooków. Celem takiej architektury jest zapewnienie spójności danych, wygodnego dostępu do globalnych informacji oraz wydzielenie lokalnych fragmentów stanu dla poszczególnych komponentów.

16.1. Globalny stan aplikacji (React Context)

Do zarządzania danymi globalnymi, takimi jak:

- **stan autoryzacji użytkownika**,
- **lista ulubionych promocji**,

stosowane są konteksty React (np. `AuthContext`, `FavouritesContext`). Kontekst pozwala dowolnemu komponentowi, niezależnie od poziomu zagnieżdżenia, pobrać i modyfikować globalny stan – np. dodać/usunąć promocję z ulubionych lub sprawdzić, czy użytkownik jest zalogowany.

Przykład udostępniania globalnego stanu ulubionych:

```
48      <FavouritesContext.Provider value={{ favourites, loading, error, add, remove }}>
49        {children}
50      </FavouritesContext.Provider>
```

16.2. Zarządzanie stanem lokalnym

Stan dotyczący konkretnego widoku lub pojedynczego komponentu (np. status ładowania, aktualne filtry, stan rozwinięcia akordeonu) utrzymywany jest lokalnie, przy użyciu hooków React (`useState`, `useEffect`).
To rozwiązanie pozwala na szybkie i niezależne aktualizowanie interfejsu, bez konieczności propagowania zmian przez całe drzewo komponentów.

Przykład zarządzania stanem ładowania:

```
}) {
  const [allItems, setAllItems] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    let mounted = true;
    setLoading(true);
    setError(null);
```

16.3. Synchronizacja i aktualizacja stanu

Połączenie kontekstów z dedykowanymi hookami pozwala na wygodną synchronizację danych – np.

- po zalogowaniu użytkownika aktualizowane są zarówno dane w AuthContext, jak i ulubione w FavouritesContext,
- zmiany ulubionych natychmiast odświeżają listę wyświetlaną użytkownikowi.

16.4. Wzorce użycia i korzyści

- Brak problemu „prop drilling” – najważniejsze dane globalne są dostępne bezpośrednio przez kontekst.
- Komponenty mogą korzystać wyłącznie z tych danych, które są im faktycznie potrzebne.
- Aplikacja pozostaje czytelna, łatwa do rozbudowy i testowania.

17. Komunikacja z backendem

Aplikacja frontendowa PromoComparer komunikuje się z backendem za pomocą interfejsu RESTful API. Wszystkie operacje na zasobach (pobieranie promocji, kategorii, sklepów, zarządzanie użytkownikami, ulubionymi) realizowane są poprzez żądania HTTP do endpointów serwera, zlokalizowanego pod adresem konfigurowanym przez zmienne środowiskowe.

17.1. Obsługa zapytań HTTP

Za realizację zapytań do backendu odpowiada dedykowany klient HTTP (`src/infrastructure/HttpClient.js`), który centralizuje obsługę wszystkich połączeń, w tym dołączanie tokenu autoryzacyjnego oraz obsługę błędów.

```
1  // src/application/services/PromotionService.js
2  import HttpClient from '../../infrastructure/HttpClient';
3
4  const PromotionService = {
5    getActive() {
6      return HttpClient.get('/api/Promotions/active');
7    },
```

Rys.17.1. Przykład wywołania żądania GET, w tym przypadku pobierająca aktywne promocje

Obsługa autoryzacji:
Jeśli użytkownik jest zalogowany, do każdego żądania dołączany jest nagłówek `Authorization: Bearer <token>`, generowany przez warstwę infrastrukturalną (`AuthInterceptor.js`). W przypadku odpowiedzi 401 (brak autoryzacji), aplikacja podejmuje próbę automatycznego odświeżenia tokenu i powtórzenia żądania.

17.2. Struktura zapytań HTTP

Każde wywołanie API opiera się na standardowych metodach HTTP:

- **GET** – pobieranie danych (np. promocji, sklepów, kategorii, ulubionych)
- **POST** – tworzenie nowych rekordów (np. rejestracja użytkownika, dodawanie do ulubionych)
- **PUT** – aktualizacja istniejących danych (np. edycja profilu użytkownika)
- **DELETE** – usuwanie danych (np. usuwanie promocji z ulubionych)

Przykład zapytania POST – dodanie do ulubionych:

```
14      async add(promotionId) {
15        await HttpClient.post('/api/UserPanel', { promotionId });
16        return this.getAll();
17      },
```

Rys.17.2. Przykład wywołania żądania POST dla obsługi ulubionych promocji

17.3. Obsługa błędów

W przypadku napotkania błędu (np. odpowiedzi 4xx/5xx lub problemów z siecią), klient HTTP zwraca błąd do warstwy aplikacyjnej, która może przekazać odpowiednią informację użytkownikowi. Typowe scenariusze obsługi błędów obejmują ponowne logowanie, wyświetlenie komunikatu o braku dostępu lub zgłoszenie problemu z siecią.

```
17  useEffect(() => {
18    let mounted = true;
19    setLoading(true);
20    setError(null);
21    fetchFn()
22      .then(data => { if (mounted) setAllItems(Array.isArray(data) ? data : []); })
23      .catch(err => {
24        if (mounted) {
25          if (err?.response?.status === 404) {
26            setAllItems([]);
27            setError(null);
28          } else {
29            setError(err);
30          }
```

Rys.17.3. Przykład obsługi błędu przy pobieraniu promocji

17.4. Format danych

Komunikacja z backendem odbywa się wyłącznie w formacie JSON, zarówno w ciele żądania (przy metodach POST/PUT), jak i w odpowiedziach serwera. **Nagłówki Content-Type** ustawiane są automatycznie przez HttpClient.

Podsumowanie:
Architektura komunikacji z backendem jest przejrzysta i centralnie zarządzana. Dzięki temu możliwa jest łatwa modyfikacja logiki autoryzacji, adresów endpointów czy obsługi wyjątków bez konieczności wprowadzania zmian w wielu miejscach kodu frontendowego.

18. Narzędzia i funkcje użytkowe (src/utilities)

Dla zapewnienia czytelności oraz ponownego wykorzystania kodu, w projekcie wydzielono moduły narzędziowe (utilities), gromadzące funkcje wspomagające różne aspekty działania aplikacji. Pliki te znajdują się w katalogu `src/utilities`.

18.1. Formatowanie dat

Jednym z podstawowych narzędzi jest funkcja formatowania dat, wykorzystywana przy prezentacji okresów obowiązywania promocji oraz dat powiązanych z ofertami.

Przykładowa implementacja (`formatDate.js`):

```
// src/utils/formatDate.js

/**
 * Parsowanie date z różnych formatów.
 * Obsługa: ISO (2025-06-21T23:59:59) oraz DD-MM-YYYY (29-06-2025)
 * @param {string} date
 * @returns {Date|null}
 */
export function parseDateSmart(date) {
  if (!date) return null;

  let parsed = new Date(date);
  if (!isNaN(parsed.getTime())) return parsed;

  const match = /^(\d{2})-(\d{2})-(\d{4})$/.exec(date);
  if (match) {
    const [, dd, mm, yyyy] = match;
    return new Date(`${yyyy}-${mm}-${dd}`);
  }

  return null;
}

/**
 * Formatowanie daty do "dd-mm-yyyy" lub "Brak informacji"
 * @param {string} date
 * @returns {string}
 */
export function formatDate(date) {
  const parsed = parseDateSmart(date);
  if (!parsed) return 'Brak informacji';
  const dd = String(parsed.getDate()).padStart(2, '0');
  const mm = String(parsed.getMonth() + 1).padStart(2, '0');
  return `${dd}-${mm}-${parsed.getFullYear()}`;
}

```

Rys. 18.1. Przykładowa funkcja formatowania daty na potrzeby interfejsu użytkownika.

18.2. Inne funkcje użytkowe

W katalogu utilities można również umieścić inne, uniwersalne funkcje, które pojawiają się w różnych miejscach projektu, np.:

- konwersje i walidacje danych,
- pomocnicze funkcje przetwarzania tekstów lub liczb,
- obsługa formatowania wartości pieniężnych.

Zalety wydzielenia narzędzi:

- Ułatwienie testowania poszczególnych funkcji.
- Możliwość łatwego refaktoringu i rozbudowy.
- Zapobieganie powielaniu kodu w wielu miejscach aplikacji.

Podsumowanie:
Zastosowanie katalogu narzędziowego pozwala zachować czystość architektury aplikacji oraz wspiera dobry styl programistyczny, oparty na zasadzie DRY (Don’t Repeat Yourself).

19. Proces wdrożenia aplikacji

Proces wdrożenia aplikacji PromoComparer został zaprojektowany w sposób maksymalnie zautomatyzowany, by umożliwić szybkie i bezpieczne publikowanie nowych wersji frontendu.

19.1. Budowanie i uruchamianie aplikacji

Aplikacja została stworzona z wykorzystaniem Create React App lub analogicznego systemu budowania.
Proces uruchamiania i budowania odbywa się w kilku prostych krokach:

1. **Instalacja zależności:**

```
npm install
```

lub

```
yarn install
```

2. **Uruchomienie wersji deweloperskiej:**

```
npm start
```

Aplikacja domyślnie dostępna jest pod adresem http://localhost:3000/.

3. **Budowanie wersji produkcyjnej:**

```
npm run build
```

W katalogu `build/` generowana jest zoptymalizowana paczka statyczna gotowa do wdrożenia na serwerze produkcyjnym.

19.2. CI/CD i środowisko produkcyjne

W celu zapewnienia wysokiej jakości i automatyzacji procesu wdrożenia rekomenduje się zastosowanie narzędzi CI/CD (Continuous Integration / Continuous Deployment), takich jak:

- **GitHub Actions**

- **GitLab CI**
- **Jenkins**
- lub inne platformy dostosowane do wymagań zespołu

Przykładowy proces CI/CD może obejmować:

- Automatyczne testowanie kodu przy każdym commitcie.
- Budowanie aplikacji na serwerze CI.
- Publikację gotowej wersji do środowiska staging/production (np. na serwerze WWW, w kontenerze Docker, lub w chmurze).

Rys. 19.1. Przykładowy przebieg procesu CI/CD:

```
Commit → Testy automatyczne → Budowanie → Publikacja na serwerze produkcyjnym
```

19.3. Konfiguracja zmiennych środowiskowych

Aplikacja korzysta z pliku `.env` do ustawiania zmiennych środowiskowych (np. adresu API backendowego):

```
REACT_APP_API_URL=https://api.twojadomena.pl
```

Przy wdrażaniu do różnych środowisk (deweloperskie, testowe, produkcyjne) wystarczy podmienić wartości w pliku `.env`.

Podsumowanie:
Proces wdrożenia PromoComparer został maksymalnie uproszczony i zautomatyzowany, co pozwala na szybkie publikowanie nowych wersji frontendu oraz bezpieczne zarządzanie konfiguracją środowiskową.