API

Program

Główny punkt wejścia aplikacji ASP.NET Core. Zawiera konfigurację serwera, dodawanie usług, middleware, routing i obsługę migracji bazy danych.

Architektura:

1. **Warstwa konfiguracji**:
    a. Dodawanie niestandardowych usług i konfiguracji, np. SignalR, JWT, CORS.
2. **Warstwa middleware**:
    a. Obsługa wyjątków, autoryzacja, uwierzytelnianie i routing.
3. **Migracje i dane testowe**:
    a. Automatyczna migracja bazy danych i seeding w trakcie uruchamiania.
4. **SignalR**:
    a. Komunikacja w czasie rzeczywistym z endpointem `/chat`.

Tworzenie `WebApplication`:

- Tworzy host aplikacji ASP.NET Core.
- **builder**: Umożliwia konfigurację usług, ustawień i środowiska.

Dodawanie usług:

- Dodaje obsługę kontrolerów MVC.
- **AuthorizeFilter**:
    - Globalna polityka autoryzacji wymaga uwierzytelnionego użytkownika dla wszystkich kontrolerów.
- **AddApplicationServices**:
    - Dodaje niestandardowe usługi aplikacyjne, np. MediatR, AutoMapper, SignalR, konfigurację CORS.
- **AddIdentityServices**:
    - Dodaje usługi tożsamości (Identity) i konfigurację JWT.

Budowanie aplikacji:

- **app = builder.Build():** Buduje aplikację na podstawie wcześniej skonfigurowanych usług i ustawień.

Middleware:

- Obsługa wyjątków **app.UseMiddleware<ExceptionMiddleware>():** Obsługuje globalne wyjątki za pomocą niestandardowego middleware.
- Swagger: Włącza Swagger dla API, ale tylko w środowisku deweloperskim.
- CORS: Używa polityki CORS zdefiniowanej w ApplicationServiceExtensions
- **`UseAuthentication`**: Ustawia middleware do weryfikacji tokenów JWT.
- **`UseAuthorization`**: Weryfikuje uprawnienia użytkownika do dostępu do zasobów.

Mapowanie endpointów:

- **`MapControllers`**: Mapuje kontrolery na ich odpowiednie trasy.
- **`MapHub<ChatHub>`**: Mapuje SignalR na endpoint `/chat`.

Migracje i seeding bazy danych:

- Tworzy zakres usług (`CreateScope`), aby uzyskać dostęp do `DataContext` i `UserManager<AppUser>`.
- Wykonuje migracje bazy danych i dodaje dane testowe za pomocą `Seed.SeedData`.
- Obsługuje błędy migracji i loguje je.

Uruchamianie aplikacji:

- **app.Run():** Uruchamia aplikację i zaczyna obsługiwać żądania.

Controllers

AccountController (Rejestracja, logowanie, pobieranie użytkownika)

- **`[ApiController]`**: Informuje, że klasa obsługuje żądania API i zapewnia domyślną walidację modelu.
- **`[Route("api/[controller]")]`**: Ustawia adres URL dla kontrolera jako `api/account`.
- **`UserManager<AppUser>`**: Zarządza użytkownikami, ich tworzeniem, aktualizacją i weryfikacją hasła.
- **`TokenService`**: Tworzy tokeny JWT dla użytkowników.
- **`[AllowAnonymous]`**: Oznacza, że endpoint nie wymaga autoryzacji.
- **`LoginDto`**: Obiekt DTO zawierający dane logowania (email i hasło).
- **`[HttpPost("register")]:`** Sprawdza, czy nazwa użytkownika lub email są już zajęte. Tworzy nowego użytkownika z `AppUser` i zapisuje w bazie danych.
- **`[Authorize]`**: Endpoint wymaga autoryzacji.
- **GetCurrentUser:** Znajduje bieżącego użytkownika na podstawie emaila w tokenie JWT.

- **CreateUserObject** Tworzy obiekt `UserDto`, który zawiera dane użytkownika, token JWT oraz zdjęcie profilowe.

ActivitiesController (Zarządzanie aktywnościami)

- **GetActivities:** Pobiera listę aktywności z paginacją.
- **GetActivity:** Pobiera szczegóły aktywności na podstawie ID.
- **`Create` Activity**: Tworzy aktywność.
- **`Edit`**: Aktualizuje istniejącą aktywność.
- **`Delete`**: Usuwa aktywność.

BaseApiController (Podstawowy kontroler)

- Obsługuje zwracanie odpowiedzi API w zależności od powodzenia lub błędu.

BugsController (Testowanie błędów)

- Testowe endpointy generujące różne odpowiedzi HTTP (`NotFound`, `BadRequest`, `Exception`).

FollowController (Zarządzanie obserwacjami)

- **`Follow`**: Pozwala użytkownikowi zacząć/przerwać obserwowanie innego użytkownika.
- **`GetFollowings`**: Pobiera listę obserwowanych/obserwujących.

PhotosController (Zarządzanie zdjęciami)

- **`Add`**: Dodaje zdjęcie.
- **`Delete`**: Usuwa zdjęcie.
- **`SetMain`**: Ustawia zdjęcie jako główne.

ProfilesController (Zarządzanie profilami użytkowników)

- **`GetProfile`**: Pobiera profil użytkownika.
- **`Edit`**: Edytuje profil.
- **`GetUserActivities`**: Pobiera aktywności użytkownika.

DTOs (Data Transfer Object)

DTOs wspierają operacje takie jak:

- **`LoginDto`:** Przesyłanie danych logowania.

- **`RegisterDto`:** Przesyłanie danych rejestracyjnych z walidacją.
- **`UserDto`:** Zwracanie szczegółów użytkownika i tokenu po operacjach takich jak logowanie i rejestracja.

LoginDto

- **Cel:** Używany do przesyłania danych logowania z klienta do serwera.
- **Pola:**
  - `Email`: Email użytkownika.
  - `Password`: Hasło użytkownika.

RegisterDto

- **Cel:** Używany podczas rejestracji nowych użytkowników.
- **Walidacja:** Dzięki atrybutom, takim jak `[Required]`, `EmailAddress`, i `RegularExpression`, serwer automatycznie weryfikuje poprawność danych.
- **Pola:**
  - `Email`: Email użytkownika, musi być poprawnym adresem email (walidacja `[EmailAddress]`).
  - `Password`: Hasło użytkownika, wymaga złożoności (co najmniej jedna cyfra, mała i wielka litera, długość 4–8 znaków).
  - `DisplayName`: Wyświetlana nazwa użytkownika.
  - `Username`: Unikalna nazwa użytkownika.

UserDto

- **Cel:** Używany do przesyłania danych o użytkowniku z serwera do klienta po zalogowaniu lub rejestracji.
- **Pola:**
  - `DisplayName`: Wyświetlana nazwa użytkownika.
  - `Token`: Token JWT do autoryzacji kolejnych żądań.
  - `Image`: URL zdjęcia profilowego użytkownika (opcjonalne).
  - `Username`: Nazwa użytkownika.

Extensions

ApplicationServiceExtensions

Ten plik definiuje rozszerzenia dla usług aplikacji, umożliwiając konfigurację różnych funkcjonalności w aplikacji.

Metoda AddApplicationServices konfiguruje kluczowe usługi dla aplikacji:

- **Swagger:** Dodaje generowanie dokumentacji API.
- **Baza danych:** Konfiguruje połączenie z bazą danych SQLite przy użyciu `DbContext`.
    - Używany jest connection string z konfiguracji (`DefaultConnection`).
- CorsPolicy: Konfiguruje CORS, aby umożliwić żądania z http://localhost:3000.
    - Pozwala na dowolne nagłówki i metody.
- **MediatR:** Dodaje wzorzec Mediator do obsługi żądań (CQRS).
- **AutoMapper:** Konfiguruje automatyczne mapowanie obiektów.
- **FluentValidation:** Dodaje walidację do modeli.
- **Scoped Services:**
    - **`IUserAccessor`:** Dostęp do informacji o bieżącym użytkowniku.
    - **`IPhotoAccessor`:** Zarządzanie zdjęciami za pomocą Cloudinary.
- **Cloudinary Settings:** Konfiguruje ustawienia Cloudinary.
- **SignalR:** Dodaje obsługę komunikacji w czasie rzeczywistym.

HttpExtensions

Definiuje rozszerzenia dla HTTP, pomagając w obsłudze nagłówków odpowiedzi.

Metoda AddPaginationHeader:

- **Cel:** Dodaje nagłówek `Pagination` do odpowiedzi HTTP, zawierający informacje o stronicowaniu.
- **Parametry:**
    - `currentPage`: Bieżąca strona.
    - `itemsPerPage`: Liczba elementów na stronie.
    - `totalItems`: Całkowita liczba elementów.
    - `totalPages`: Łączna liczba stron.
- **Działanie:**
    - Serializuje dane stronicowania do JSON i dodaje je do nagłówka `Pagination`.
    - Dodaje nagłówek `Access-Control-Expose-Headers`, aby umożliwić klientowi dostęp do nagłówka `Pagination`.

IdentityServiceExtensions

Rozszerzenia dla usług tożsamości i autoryzacji.

- **`ApplicationServiceExtensions`:** Konfiguruje usługi aplikacji, takie jak MediatR, FluentValidation i SignalR.

- **`HttpExtensions`:** Umożliwia dodanie nagłówków stronicowania do odpowiedzi HTTP.
- **`IdentityServiceExtensions`:** Obsługuje tożsamość, uwierzytelnianie JWT i polityki autoryzacji.

Metoda AddIdentityServices konfiguruje:

- **Identity Core:**
  - Włącza `AppUser` jako podstawową klasę użytkownika.
  - Wymaga unikalnych adresów email i wyłącza wymóg znaków specjalnych w hasłach.
- **JWT Authentication:**
  - Konfiguruje uwierzytelnianie JWT przy użyciu klucza symetrycznego (`TokenKey` z konfiguracji).
  - Parametry weryfikacji tokenu:
    - Sprawdzanie podpisu (`ValidateIssuerSigningKey`).
    - Wyłączona walidacja `Issuer` i `Audience`.
  - Obsługa komunikacji SignalR przez token w parametrach URL.
- **Authorization Policies:**
  - Polityka `IsActivityHost` wymaga, aby użytkownik był gospodarzem aktywności.
- **Scoped Services:**
  - **`TokenService`:** Tworzenie tokenów JWT.
- **Transient Services:**
  - **`IAuthorizationHandler`:** Obsługa wymagań dla polityki `IsActivityHost`.

Middleware

Przepływ działania:

1. Middleware przechwytuje każde żądanie HTTP.
2. Przekazuje żądanie dalej w potoku za pomocą `_next(context)`.
3. Jeśli wystąpi wyjątek:
   a. Loguje szczegóły błędu.
   b. Tworzy obiekt odpowiedzi (zależnie od środowiska).
   c. Serializuje odpowiedź do JSON.
   d. Ustawia odpowiedni kod statusu HTTP.
   e. Wysyła odpowiedź do klienta.

ExceptionMiddleware

Ten middleware służy do globalnej obsługi wyjątków w aplikacji ASP.NET Core. Jego celem jest przechwytywanie nieobsłużonych wyjątków, logowanie ich i zwracanie spójnej odpowiedzi HTTP klientowi.

Konstruktor:

- **Parametry:**
  - `RequestDelegate next`: Reprezentuje następny element w potoku przetwarzania HTTP.
  - `ILogger<ExceptionMiddleware> logger`: Logger do rejestrowania informacji o wyjątkach.
  - `IHostEnvironment environment`: Środowisko uruchomieniowe aplikacji (np. Development, Production).
- **Działanie:** Przechowuje przekazane zależności jako pola klasy.

Metoda InvokeAsync:

- **Parametr:**
  - `HttpContext context`: Kontekst bieżącego żądania HTTP.
- **Działanie:**
  - Próbuje przekazać żądanie do kolejnego komponentu w potoku za pomocą `_next(context)`.
  - Jeśli wystąpi wyjątek, przechwyci go w bloku `catch`.

Blok catch:

- Logowanie wyjątku: Loguje szczegóły wyjątku, w tym jego wiadomość.
- Ustawienie odpowiedzi HTTP:
  - Typ treści odpowiedzi ustawiony na JSON (`application/json`).
  - Status odpowiedzi ustawiony na **500 Internal Server Error**.
- Tworzenie obiektu odpowiedzi:
  - Jeśli aplikacja działa w trybie **Development**, odpowiedź zawiera:
    - Kod statusu.
    - Wiadomość wyjątku.
    - Ślad stosu (ang. *stack trace*).
  - W trybie **Production** zawiera tylko kod statusu i ogólny komunikat "Internal Server Error".
- Serializacja do JSON:
  - Serializuje obiekt odpowiedzi do formatu JSON.
  - Wymusza użycie konwencji *camelCase* w nazwach właściwości.

- Zapisanie odpowiedzi:
    - Wysyła serializowaną odpowiedź JSON do klienta.

Klasa pomocnicza: `AppException`

- Jest to obiekt reprezentujący szczegóły wyjątku.
- Przechowuje:
    - Kod statusu HTTP.
    - Wiadomość błędu.
    - Szczegóły śladu stosu (opcjonalnie).

Properties

launchSettings.json

Plik `launchSettings.json` jest używany w projektach ASP.NET Core do definiowania ustawień uruchamiania projektu. Określa konfigurację środowiska, adresy URL aplikacji oraz inne właściwości związane z uruchamianiem.

- Plik `launchSettings.json` jest automatycznie wykorzystywany podczas uruchamiania aplikacji w środowiskach developerskich (np. Visual Studio, `dotnet run`).
- Ustawia środowisko aplikacji (`Development` w tym przypadku).
- Określa adres URL (`http://localhost:5000`).
- Kontroluje, czy przeglądarka ma być otwierana automatycznie.
- Można zmodyfikować wartość `ASPNETCORE_ENVIRONMENT`, np. na `Production`.

$schema:

- Określa schemat JSON używany do walidacji struktury tego pliku.
- Dzięki temu, narzędzia takie jak Visual Studio mogą zapewnić podpowiedzi i weryfikację poprawności.

profiles:

- Sekcja `profiles` zawiera różne konfiguracje środowisk uruchamiania aplikacji.
- W tym przypadku istnieje jeden profil: `http`.
- **Kluczowe pola:**
    - **`commandName`:** `"Project"`
        - Wskazuje, że aplikacja będzie uruchamiana jako projekt .NET Core.
        - Alternatywne wartości mogą obejmować np. `"IISExpress"`.
    - **`dotnetRunMessages`:** `true`

        - Włącza szczegółowe wiadomości w konsoli podczas używania `dotnet run`.
    - **`launchBrowser`:** `false`
        - Określa, czy przeglądarka ma być automatycznie uruchamiana po starcie aplikacji.
        - Wartość `false` oznacza, że przeglądarka nie zostanie otwarta.
    - **`applicationUrl`:** `"http://localhost:5000"`
        - Definiuje adres URL, pod którym aplikacja będzie dostępna lokalnie.
        - Można zmienić ten adres, np. na `"http://localhost:5001"` dla protokołu HTTPS.
    - **environmentVariables:** Konfiguruje zmienne środowiskowe dla aplikacji. **`ASPNETCORE_ENVIRONMENT`**:
        - Ustawia środowisko aplikacji na `Development`.
        - W środowisku deweloperskim aplikacja wyświetla bardziej szczegółowe logi i dane debugowania.
        - Inne wartości: `Production`, `Staging`.

Services

TokenServices

Token składa się z trzech części:

1. **Nagłówek (Header)**: Metadane tokena (np. algorytm HMAC SHA-512).
2. **Payload**: Roszczenia (claims) użytkownika.
3. **Podpis (Signature)**: Hasz nagłówka i payloadu z użyciem klucza tajnego.

Namespace i zależności:

- **`System.IdentityModel.Tokens.Jwt`**: Używane do obsługi tokenów JWT, w tym ich generowania i serializacji.
- **`System.Security.Claims`**: Zapewnia klasy do zarządzania roszczeniami (claims), które są używane w celu opisania użytkownika.
- **`Microsoft.IdentityModel.Tokens`**: Biblioteka do pracy z kluczami kryptograficznymi i podpisami tokenów.
- **`Domain`**: Definiuje klasę `AppUser`, która reprezentuje użytkownika aplikacji.

Konstruktor:

- **`IConfiguration`**: Interfejs używany do odczytywania ustawień konfiguracyjnych (np. klucz tajny dla tokenów).

- Klucz tajny (TokenKey) jest przechowywany w pliku konfiguracyjnym aplikacji, np. `Appsettings.json`.

CreateToken:

- **`List<Claim>`**: Lista roszczeń przypisanych do tokena. Roszczenia to informacje o użytkowniku, które można uwierzytelnić.
  - **`ClaimTypes.Name`**: Nazwa użytkownika.
  - **`ClaimTypes.NameIdentifier`**: Identyfikator użytkownika.
  - **`ClaimTypes.Email`**: Adres e-mail użytkownika
- **`SymmetricSecurityKey`**: Tworzy klucz symetryczny z tajnego ciągu znaków (TokenKey).
  - Klucz powinien być przechowywany w bezpiecznym miejscu (np. zmiennych środowiskowych).
- **`SigningCredentials`**: Definiuje algorytm podpisu tokena. W tym przypadku używany jest **HMAC SHA-512**.
- **tokenDescriptor:**
  - **`Subject`**: Zawiera roszczenia użytkownika.
  - **`Expires`**: Ustawia czas wygaśnięcia tokena (7 dni od wygenerowania).
  - **`SigningCredentials`**: Określa klucz i algorytm podpisu
- **`JwtSecurityTokenHandler`**: Klasa pomocnicza do obsługi tokenów JWT.
- **`CreateToken`**: Generuje token na podstawie deskryptora.
- **`WriteToken`**: Serializuje token do ciągu znaków, który może być przesyłany w nagłówkach HTTP (np. jako `Authorization: Bearer <token>`).

SignalR

SignalR w praktyce

Komunikacja z klientem:

- Klient (np. aplikacja React) subskrybuje zdarzenia SignalR, takie jak:
  - `"ReceiveComment"`: Odbieranie nowych komentarzy.
  - `"LoadComments"`: Ładowanie istniejących komentarzy po podłączeniu.

Przykładowy scenariusz:

- Użytkownik otwiera stronę aktywności.
- Klient wysyła żądanie połączenia z SignalR z `activityId` w zapytaniu.
- Serwer dodaje użytkownika do grupy i ładuje istniejące komentarze.

- Gdy ktoś doda nowy komentarz, serwer wysyła go do wszystkich użytkowników w grupie.

ChatHub

Klasa **ChatHub** implementuje komunikację w czasie rzeczywistym za pomocą SignalR. Umożliwia przesyłanie i odbieranie komentarzy w kontekście konkretnych aktywności.

Importowanie zależności:

- **`Application.Comments`**: Zawiera logikę obsługi komentarzy, np. klasy `Create.Command` i `List.Query`.
- **`MediatR`**: Służy do obsługi wzorca CQRS (Command Query Responsibility Segregation).
- **`Microsoft.AspNetCore.SignalR`**: Biblioteka do obsługi komunikacji w czasie rzeczywistym za pomocą SignalR.

Konstruktor:

- **`IMediator`**: Używany do wysyłania komend i zapytań w ramach wzorca CQRS. Pozwala na oddzielenie logiki biznesowej od kodu SignalR.

Metoda `SendComment:`

- **Cel**: Wysyłanie nowego komentarza do grupy użytkowników powiązanych z daną aktywnością.
- **Parametr**: Create.Command
  - Zawiera dane nowego komentarza (np. treść, identyfikator aktywności).
- **Obsługa grup**:
  - **Clients.Group**: Wysyła wiadomość do wszystkich klientów w grupie.
  - **command.ActivityId.ToString()**: Identyfikator aktywności, służy jako nazwa grupy.
- **SendAsync("ReceiveComment", comment.Value)**:
  - Wysyła do klientów zdarzenie "ReceiveComment" z treścią komentarza.

Metoda `OnConnectedAsync`:

- **Cel**: Obsługa połączenia nowego klienta.
- **Kroki**:
  1. **`Context.GetHttpContext()`**:
     - Pobiera kontekst HTTP dla bieżącego połączenia.
  2. **`httpContext.Request.Query["activityId"]`**:

    - Odczytuje identyfikator aktywności z parametrów zapytania.
3. **Dodanie do grupy**:
    - **`Groups.AddToGroupAsync(Context.ConnectionId, activityId)`**:
        - Dodaje klienta do grupy SignalR na podstawie identyfikatora aktywności.
4. **Ładowanie istniejących komentarzy**:
    - **`List.Query`**: Wysyła zapytanie, aby pobrać istniejące komentarze dla danej aktywności.
    - **`Clients.Caller.SendAsync("LoadComments", result.Value)`**:
        - Wysyła istniejące komentarze do nowo podłączonego klienta.

Application

Aplikacja zarządza "działaniami" (np. wydarzeniami, spotkaniami) w systemie. W folderze `Application/Activities` zdefiniowane są klasy i logika dotycząca operacji na tych działaniach, takie jak tworzenie, edytowanie, usuwanie, wyświetlanie szczegółów itp.

Każda klasa pełni określoną funkcję w procesie zarządzania działaniami w aplikacji.

Activities

ActivityDto

`ActivityDto` to obiekt transferu danych, który reprezentuje "działanie" (np. wydarzenie) w aplikacji. Zawiera właściwości takie jak tytuł, data, opis, kategoria, miejsce, organizator i lista uczestników.

`AttendeeDto` reprezentuje uczestnika wydarzenia z dodatkowymi informacjami (takimi jak nazwa wyświetlana, bio, zdjęcie, liczbę obserwujących i liczbę osób, które obserwują danego użytkownika).

ActivityParams

`ActivityParams` rozszerza klasę `PagingParams` (paginacja) i dodaje możliwość filtrowania działań na podstawie tego, czy użytkownik bierze udział w wydarzeniu (`IsGoing`), czy jest gospodarzem (`IsHost`), oraz ustawia datę początkową (`StartDate`).

ActivityValidator

`ActivityValidator` wykorzystuje bibliotekę FluentValidation do walidacji właściwości obiektu `Activity`. Sprawdza, czy wszystkie wymagane pola (tytuł, opis, data, kategoria, miasto, miejsce) są wypełnione.

Create

`Create`: Definiuje operację tworzenia nowego wydarzenia. Wydarzenie jest zapisywane w bazie danych, a organizator (gospodarz) jest dodawany do listy uczestników.

Delete

`Delete`: Definiuje operację usuwania wydarzenia z bazy danych na podstawie przekazanego identyfikatora.

Details

`Details`: Pobiera szczegóły wydarzenia na podstawie jego identyfikatora i zwraca je w formacie `ActivityDto`.

Edit

`Edit`: Obsługuje edycję istniejącego wydarzenia. Aktualizuje dane wydarzenia w bazie danych.

List

`List`: Lista wydarzeń z paginacją, uwzględniająca filtry na podstawie statusu użytkownika (czy bierze udział, czy jest gospodarzem).

UpdateAttendance

`UpdateAttendance`: Obsługuje logikę związaną ze zmianą statusu uczestnictwa w wydarzeniu (dodanie, usunięcie uczestnika, zmiana statusu gospodarza).

Comments

CommentDto

`CommentDto` to obiekt transferu danych (DTO), który reprezentuje komentarz. Zawiera następujące właściwości:

- `Id`: Identyfikator komentarza.
- `CreatedAt`: Data i godzina utworzenia komentarza.
- `Body`: Treść komentarza.
- `Username`: Nazwa użytkownika, który napisał komentarz.
- `DisplayName`: Nazwa wyświetlana użytkownika.

- `Image`: Obrazek użytkownika (np. zdjęcie profilowe).

Create

`Komenda: Stworzenie komentarza`

`Command`: Reprezentuje żądanie stworzenia nowego komentarza. Zawiera:

- `Body`: Treść komentarza.
- `ActivityId`: Identyfikator aktywności, do której komentarz jest dodawany.

Walidator dla Komendy Tworzenia

- **`CommandValidator`** zapewnia, że treść komentarza (`Body`) nie jest pusta przed przetworzeniem żądania

Handler dla Komendy Tworzenia

`Handler` przetwarza tworzenie nowego komentarza. Działa w następujący sposób:

- Pobiera aktywność związaną z komentarzem (`ActivityId`).
- Pobiera aktualnego użytkownika (`_userAccessor.GetUsername()`).
- Dodaje komentarz do aktywności.
- Zapisuje zmiany w bazie danych.
- Zwraca obiekt `CommentDto` w przypadku powodzenia lub komunikat o błędzie, jeśli operacja nie powiodła się.

List

Zapytanie: Wyszukiwanie komentarzy do aktywności

- **`Query`** reprezentuje zapytanie o listę komentarzy dla danej aktywności (`ActivityId`).

`Handler` dla wyświetlania komentarzy:

- Pobiera wszystkie komentarze związane z określoną aktywnością.
- Sortuje je według daty utworzenia w porządku malejącym.
- Przekształca encje `Comment` na obiekty DTO (`CommentDto`) za pomocą AutoMappera.
- Zwraca listę komentarzy do klienta.

Core

MappingProfiles

`MappingProfiles` wykorzystuje bibliotekę **AutoMapper** do mapowania obiektów w aplikacji. Mapowanie obejmuje:

- `Activity` do `ActivityDto`, w tym pole `HostUsername`.
- `ActivityAttendee` do `AttendeeDto`, z dodatkowymi informacjami o użytkowniku, takie jak `DisplayName`, `Username`, `FollowersCount` itp.
- `AppUser` do `Profile`, mapując zdjęcie profilowe i liczbę obserwujących.
- `Comment` do `CommentDto`, mapując dane autora komentarza.
- `ActivityAttendee` do `UserActivityDto`, mapując dane o aktywnościach użytkownika.

AppException

`AppException` to klasa reprezentująca wyjątek aplikacji. Służy do zwracania informacji o błędach, takich jak:

- `StatusCode`: Kod statusu HTTP, np. 404 dla "Not Found".
- `Message`: Wiadomość o błędzie.
- `Details`: Opcjonalne szczegóły dotyczące błędu.

PagedList

`PagedList<T>` to klasa, która umożliwia paginację (stronicowanie) wyników. Pomaga w zwracaniu wyników w mniejszych porcjach (stronach). Zawiera:

- `CurrentPage`: Numer aktualnej strony.
- `TotalPages`: Łączna liczba stron.
- `PageSize`: Liczba elementów na stronie.
- `TotalCount`: Całkowita liczba elementów.
- **`CreateAsync`**: Metoda statyczna, która tworzy obiekt `PagedList` na podstawie zapytania do bazy danych, rozdzielając wyniki na strony.

PagingParams

`PagingParams` to klasa, która reprezentuje parametry stronicowania, takie jak:

- `PageNumber`: Numer strony (domyślnie 1).
- `PageSize`: Liczba elementów na stronie, z limitem 50 (maksymalna liczba wyników na stronie).

Result

`Result<T>` to klasa pomocnicza do reprezentowania wyników operacji. Zawiera:

- `IsSuccess`: Flaga wskazująca, czy operacja zakończyła się sukcesem.
- `Value`: Wartość zwrócona przez operację (jeśli sukces).
- `Error`: Komunikat błędu (jeśli niepowodzenie).
- **`Success` i `Failure`**: Statyczne metody do tworzenia wyników sukcesu lub błędu.

Followers

FollowToggle

`FollowToggle` obsługuje logikę dodawania lub usuwania obserwujących.

- **`Command`**: Zawiera dane niezbędne do wykonania operacji, tj. `TargetUsername` (nazwa użytkownika, którego chce się obserwować lub przestać obserwować).
- **`Handler`**: Wykonuje operację na bazie danych:
  - Pobiera użytkownika obserwującego (`observer`) oraz użytkownika, którego obserwowanie ma być zmienione (`target`).
  - Sprawdza, czy istnieje już relacja obserwujący-obserwowany. Jeśli nie, tworzy nową; jeśli tak, usuwa istniejącą.
  - Zapisuje zmiany w bazie danych i zwraca wynik operacji.
- Zwraca wynik operacji, sukces lub błąd.

List

`List` obsługuje pobieranie listy obserwujących lub obserwowanych użytkowników.

- **`Query`**: Zawiera dane wejściowe dla zapytania:
  - `Predicate`: Określa, czy chcemy pobrać "followers" (obserwujących) czy "following" (obserwowanych).
  - `Username`: Nazwa użytkownika, dla którego chcemy pobrać listę obserwujących lub obserwowanych.
- **`Handler`**: Wykonuje zapytanie do bazy danych w zależności od wartości `Predicate`:
  - Jeśli `Predicate` to `"followers"`, wybiera obserwujących użytkownika.
  - Jeśli `Predicate` to `"following"`, wybiera użytkowników, których dany użytkownik obserwuje.
  - Obiekty są mapowane na profil użytkownika za pomocą AutoMappera.
- Zwraca wynik operacji, który zawiera listę profili.

Interfaces

IPhotoAccessor

- **`IPhotoAccessor`**: Interfejs odpowiedzialny za operacje związane z zarządzaniem zdjęciami w aplikacji.
    - **`AddPhoto(IFormFile file)`**: Metoda dodająca zdjęcie do systemu. Przyjmuje plik typu `IFormFile` (typ pliku z formularza HTTP) i zwraca wynik dodania zdjęcia w postaci obiektu `PhotoUploadResult`.
    - **`DeletePhoto(string publicId)`**: Metoda do usuwania zdjęcia na podstawie `publicId` – identyfikatora zdjęcia. Zwraca ciąg znaków (np. potwierdzenie usunięcia lub błąd).

Interfejs ten jest używany w aplikacji do interakcji z systemem przechowywania zdjęć, np. w chmurze (np. Cloudinary lub AWS S3).

IUserAccessor

- **`IUserAccessor`**: Interfejs odpowiedzialny za dostęp do informacji o użytkowniku.
    - **`GetUsername()`**: Metoda, która zwraca nazwę użytkownika (prawdopodobnie z aktualnej sesji). Może być wykorzystywana do uzyskiwania nazwy zalogowanego użytkownika w aplikacji.

Ten interfejs jest przydatny do uzyskiwania dostępu do informacji o aktualnie zalogowanym użytkowniku, np. przy dodawaniu komentarzy, obserwowaniu innych użytkowników czy przy edytowaniu profilu.

Photos

Add

`Add`: Zawiera logikę dodawania zdjęcia do użytkownika.

- **`Command`**: Klasa reprezentująca polecenie dodania zdjęcia. Zawiera plik `IFormFile`, który jest zdjęciem przesyłanym w formularzu.
- **`Handler`**: Zajmuje się obsługą logiki dodawania zdjęcia. Po dodaniu zdjęcia do systemu (za pomocą `IPhotoAccessor`), zapisuje dane zdjęcia w bazie danych, przypisując je do użytkownika. Jeśli użytkownik nie ma jeszcze głównego zdjęcia, to nowe zdjęcie staje się głównym.

Delete

`Delete`: Zawiera logikę usuwania zdjęcia użytkownika.

- **`Command`**: Klasa reprezentująca polecenie usunięcia zdjęcia na podstawie jego `Id`.
- **`Handler`**: Weryfikuje, czy zdjęcie istnieje, a także sprawdza, czy nie jest ono zdjęciem głównym (jeśli tak, operacja jest zabroniona). Następnie usuwa zdjęcie z systemu i baz danych.

PhotoUploadResult

`PhotoUploadResult`: Klasa reprezentująca wynik operacji przesyłania zdjęcia, zawierająca:

- **`PublicId`**: Unikalny identyfikator zdjęcia w systemie przechowywania zdjęć (np. w chmurze).
- **`Url`**: URL do zdjęcia, które zostało przesłane.

SetMain

`SetMain`: Zawiera logikę ustawiania zdjęcia jako główne.

- **`Command`**: Klasa reprezentująca polecenie zmiany głównego zdjęcia użytkownika na podstawie jego `Id`.
- **`Handler`**: Wyszukuje zdjęcie o podanym `Id` i ustawia je jako główne zdjęcie użytkownika. Jeśli już istnieje zdjęcie główne, zostaje ono ustawione na nie-główne.

Profiles

Details

`Details`: Zawiera logikę pobierania szczegółów profilu użytkownika na podstawie nazwy użytkownika.

- **`Query`**: Klasa reprezentująca zapytanie o szczegóły profilu użytkownika na podstawie `Username`.
- **`Handler`**: Obsługuje zapytanie, pobierając dane użytkownika z bazy danych i mapując je na obiekt `Profile`. Zwraca wynik w postaci obiektu `Profile`.

Edit

`Edit`: Zawiera logikę edytowania profilu użytkownika.

- **`Command`**: Klasa reprezentująca polecenie edycji profilu. Zawiera informacje o zmienianych polach (np. `DisplayName` i `Bio`).
- **`CommandValidator`**: Walidator sprawdzający, czy `DisplayName` nie jest pusty.
- **`Handler`**: Obsługuje polecenie, aktualizując dane użytkownika w bazie danych.

ListActivities

`ListActivities`: Zawiera logikę pobierania listy aktywności związanych z użytkownikiem.

- **`Query`**: Klasa reprezentująca zapytanie o aktywności użytkownika na podstawie `Username` i `Predicate` (np. "past", "hosting").
- **`Handler`**: Obsługuje zapytanie, filtrując aktywności na podstawie daty oraz typu aktywności (np. przeszłe lub nadchodzące).

Profile

`Profile`: Klasa reprezentująca profil użytkownika, zawierająca pola takie jak:

- `Username`: Nazwa użytkownika.
- `DisplayName`: Wyświetlana nazwa użytkownika.
- `Bio`: Krótkie bio użytkownika.
- `Image`: URL do zdjęcia profilowego.
- `Following`: Flaga wskazująca, czy użytkownik śledzi obecnego użytkownika.
- `FollowersCount` i `FollowingCount`: Liczba obserwujących i obserwowanych.
- `Photos`: Kolekcja zdjęć powiązanych z profilem.

UserActivityDto

`UserActivityDto`: Klasa reprezentująca dane o aktywności użytkownika, zawierająca:

- `Id`: Unikalny identyfikator aktywności.
- `Title`: Tytuł aktywności.
- `Category`: Kategoria aktywności.
- `Date`: Data aktywności.
- `HostUsername`: Nazwa użytkownika prowadzącego aktywność (zignorowana przy serializacji).

Domain

Activity

`Activity`: Reprezentuje aktywność w systemie, z takimi polami jak:

- `Id`: Unikalny identyfikator aktywności.
- `Title`: Tytuł aktywności.
- `Date`: Data aktywności.
- `Description`: Opis aktywności.
- `Category`: Kategoria aktywności (np. sport, kultura).

- `City`: Miasto, w którym odbywa się aktywność.
- Venue: Miejsce odbywania się aktywności.
- `IsCancelled`: Flaga wskazująca, czy aktywność została anulowana.
- `Attendees`: Kolekcja uczestników aktywności, reprezentowanych przez klasę `ActivityAttendee`.
- `Comments`: Kolekcja komentarzy do aktywności, reprezentowanych przez klasę `Comment`.

ActivityAttendee

`ActivityAttendee`: Reprezentuje powiązanie między użytkownikiem a aktywnością, wskazując, czy użytkownik jest gospodarzem aktywności.

- `AppUserId`: Identyfikator użytkownika (`AppUser`).
- `AppUser`: Użytkownik biorący udział w aktywności.
- `ActivityId`: Identyfikator aktywności (`Activity`).
- `Activity`: Aktywność, w której bierze udział użytkownik.
- `IsHost`: Flaga wskazująca, czy użytkownik jest gospodarzem aktywności.

AppUser

`AppUser`: Reprezentuje użytkownika aplikacji, rozszerzając `IdentityUser` (domyślną klasę użytkownika w ASP.NET Core Identity). Zawiera dodatkowe właściwości:

- `DisplayName`: Wyświetlana nazwa użytkownika.
- `Bio`: Biografia użytkownika.
- `Activities`: Kolekcja powiązań użytkownika z aktywnościami (`ActivityAttendee`).
- `Photos`: Kolekcja zdjęć użytkownika (`Photo`).
- `Followings`: Kolekcja użytkowników, których użytkownik śledzi (`UserFollowing`).
- `Followers`: Kolekcja użytkowników, którzy śledzą tego użytkownika (`UserFollowing`).

Comment

`Comment`: Reprezentuje komentarz do aktywności.

- `Id`: Unikalny identyfikator komentarza.
- `Body`: Treść komentarza.
- `Author`: Użytkownik, który napisał komentarz (`AppUser`).
- `Activity`: Aktywność, do której komentarz należy (`Activity`).
- `CreatedAt`: Data utworzenia komentarza (domyślnie ustawiona na aktualny czas).

Photo

`Photo`: Reprezentuje zdjęcie użytkownika.

- `Id`: Unikalny identyfikator zdjęcia.
- `Url`: URL do zdjęcia.
- `IsMain`: Flaga wskazująca, czy zdjęcie jest głównym zdjęciem użytkownika.

UserFollowing

- **`UserFollowing`**: Reprezentuje relację śledzenia między dwoma użytkownikami.
  - `ObserverId`: Identyfikator użytkownika, który śledzi (`AppUser`).
  - `Observer`: Użytkownik, który śledzi.
  - `TargetId`: Identyfikator użytkownika, który jest śledzony (`AppUser`).
  - `Target`: Użytkownik, który jest śledzony.

Infrastructure

Photos

CloudinarySettings

`CloudinarySettings`: Klasa, która przechowuje ustawienia konfiguracyjne wymagane do połączenia z Cloudinary.

- `CloudName`: Nazwa chmury w Cloudinary.
- `ApiKey`: Klucz API używany do uwierzytelnienia w Cloudinary.
- `ApiSecret`: Sekretny klucz API używany do uwierzytelnienia w Cloudinary.

PhotoAccessor

`PhotoAccessor`: Implementuje interfejs **`IPhotoAccessor`** i jest odpowiedzialna za interakcję z usługą Cloudinary w celu dodawania i usuwania zdjęć.

- **Konstruktor**:
  - Pobiera konfigurację Cloudinary (z `IOptions<CloudinarySettings>`) i tworzy instancję obiektu `Cloudinary`, który będzie używany do przesyłania i usuwania zdjęć.
- **`AddPhoto`**: Metoda do przesyłania zdjęć do Cloudinary.
  - Oczekuje pliku (`IFormFile`).
  - Sprawdza, czy plik ma zawartość, a następnie przekształca go na stream.
  - Tworzy parametry przesyłania (`ImageUploadParams`), w tym transformację obrazu (zmiana rozmiaru na 500x500 pikseli i wypełnienie).

    - Przesyła obraz do Cloudinary, a wynik (w tym URL i PublicId) jest zwracany w obiekcie `PhotoUploadResult`.
    - Jeśli wystąpi błąd, wyjątek jest rzucany z komunikatem błędu.
- **`DeletePhoto`**: Metoda do usuwania zdjęć z Cloudinary.
    - Wymaga `publicId` zdjęcia, które ma zostać usunięte.
    - Tworzy parametry do usunięcia zdjęcia (`DeletionParams`) i wywołuje metodę `DestroyAsync` na obiekcie `Cloudinary`.
    - Zwraca wynik operacji (jeśli wynik to "ok", zdjęcie zostało pomyślnie usunięte).

Security

Folder **Security** w **Infrastructure** zawiera dwie klasy, które odpowiadają za zarządzanie autoryzacją i dostępem do danych użytkowników w kontekście aplikacji:

1. **`IsHostRequirement` oraz `IsHostRequirementHandler`**: Te klasy implementują niestandardowy mechanizm autoryzacji, który sprawdza, czy użytkownik jest gospodarzem danej aktywności (eventu). Jest to przydatne, gdy chcemy ograniczyć dostęp do pewnych zasobów lub akcji w aplikacji tylko do gospodarzy danej aktywności.
2. **`UserAccessor`**: Klasa ta umożliwia aplikacji dostęp do nazwy użytkownika aktualnie zalogowanego użytkownika, wykorzystując dane z kontekstu HTTP (np. token JWT).

IsHostRequirement

`IsHostRequirement`: Klasa, która implementuje interfejs **`IAuthorizationRequirement`** i pełni rolę wymogu autoryzacji, który będzie sprawdzał, czy użytkownik jest gospodarzem (hostem) danej aktywności.

`IsHostRequirementHandler`: Klasa obsługująca logikę dla **`IsHostRequirement`**.

- **Konstruktor**: Przyjmuje instancje `DataContext` (do pracy z bazą danych) oraz `IHttpContextAccessor` (do dostępu do kontekstu HTTP).
- **`HandleRequirementAsync`**: Metoda, która jest wywoływana, aby sprawdzić, czy użytkownik spełnia wymagania.
    - Pobiera `userId` z kontekstu użytkownika (z tokena).
    - Wydobywa `activityId` z wartości trasy (route value) w żądaniu HTTP.
    - Sprawdza w bazie danych, czy użytkownik jest zapisany jako uczestnik danej aktywności i czy jest jej gospodarzem (`IsHost`).
    - Jeśli użytkownik jest gospodarzem, metoda wzywa `context.Succeed(requirement)`, co oznacza, że spełnia wymagania.

UserAccessor

`UserAccessor`: Klasa implementująca interfejs **`IUserAccessor`**, który zapewnia dostęp do nazwy użytkownika (username) z kontekstu HTTP.

- **Konstruktor**: Przyjmuje instancję **`IHttpContextAccessor`**, która pozwala na dostęp do bieżącego kontekstu HTTP.
- **`GetUsername`**: Metoda, która zwraca nazwę użytkownika (username) pobraną z tokena JWT lub ciasteczka w kontekście HTTP. Używa `ClaimTypes.Name`, aby znaleźć wartość `Name` z roszczenia (claim).

Persistence

Migrations

Folder **Migrations** zawiera pliki migracji, które są tworzone podczas procesu zmiany schematu bazy danych. Migracje są używane do śledzenia zmian w modelach danych (np. dodawanie nowych tabel, zmiana istniejących) i stosowania tych zmian do bazy danych.

Migracje zawierają:

- **Pliki migracji (np. `20250106120000_InitialCreate.cs`)**: Każdy plik migracji zawiera zmiany, które zostały wprowadzone do schematu bazy danych.
- **Plik `ModelSnapshot.cs`**: Określa aktualny stan modelu bazy danych. Jest to reprezentacja bazy danych w danym momencie.

Każda migracja generuje dwie główne metody:

1. **Up()**: Określa operacje, które mają być wykonane podczas migracji (np. tworzenie tabel, dodawanie kolumn).
2. **Down()**: Określa operacje, które mają przywrócić bazę danych do poprzedniego stanu (np. usunięcie tabel, usunięcie kolumn).

Migracje są zwykle tworzone za pomocą polecenia `dotnet ef migrations add <Name>` i stosowane przy pomocy `dotnet ef database update`.

DataContext

`DataContext`: Klasa pochodna po `IdentityDbContext<AppUser>`, która zarządza dostępem do bazy danych i mapuje modele na tabele w bazie danych.

- **DbSet<TEntity>**: Zbiory reprezentujące tabele w bazie danych dla poszczególnych modeli, takich jak `Activity`, `ActivityAttendee`, `Photo`, `Comment` i `UserFollowing`.
- **`OnModelCreating`**: Metoda konfiguracyjna, która ustawia szczegóły mapowania encji, takie jak klucze obce i zachowanie przy usuwaniu danych (np. `DeleteBehavior.Cascade`).

Seed

Seed: Klasa, której celem jest inicjalizacja danych w bazie danych, jeśli baza jest pusta.

- **`SeedData`**: Metoda statyczna, która tworzy użytkowników i aktywności w bazie danych, jeżeli w bazie danych nie ma jeszcze żadnych użytkowników ani aktywności.
- Tworzy przykładowych użytkowników (`AppUser`) oraz aktywności (`Activity`) i dodaje je do bazy danych.