19.10.2025

Laboratorium nr 1

Temat: Tworzenie API REST

Paweł Myszka nr. Indeksu: 331720

Zad 1

1. Utworzenie projektu

Utworzyłem projekt Web API w .NET przy użyciu polecenia:

```
base) PS C:\Users\pawel\Desktop\Desktop\z2> dotnet new webapi -n ShopAPI.Api
>>
.NET 9.0 — Zapraszamy!
----------
Wersja zestawu SDK: 9.0.306
```

Projekt został otwarty w Visual Studio Code, a wbudowany terminal umożliwił jego uruchomienie i dalszą pracę.

2. Uruchomienie projektu i test startowy

Projekt uruchomiłem poleceniem:

```
(base) PS C:\Users\pawel\Desktop\Desktop\z2\ShopAPI.Api> dotnet run
```

Serwer lokalnie nasłuchuje pod adresem:

```
    Now listening on: http://localhost:5235
```

Po uruchomieniu serwera dostępne jest środowisko do testowania endpointów.

3. Problemy z Swagger UI

Podczas pierwszych prób uruchomienia Swagger UI pojawiały się błędy 404 oraz brak połączenia z adresem http://localhost:5235/swagger/index.html. Przyczyny problemu:

- W automatycznie wygenerowanym projekcie użyto metod builder.Services.AddOpenApi() i app.MapOpenApi(), które nie generują klasycznego Swagger UI.

W celu rozwiązania projektu dodałem pakiet Swashbuckle.AspNetCore.

Włączyłem klasyczny Swagger UI poprzez builder.Services.AddSwaggerGen(), app.UseSwagger() i app.UseSwaggerUI().

Zad 2

1. **W terminalu sprawdziłem obecność certyfikatu developerskiego poleceniem:**

```
(base) PS C:\Users\pawel\Desktop\Desktop\z2\ShopAPI.Api> dotnet dev-certs https --check

A valid certificate was found: CFF11642B6517A4B685F6AE3A000C93C0812EF9A - CN=localhost - Valid from 2025-10-05 10:14:24Z to 2026-10-05 10:14:24Z - IsHttpsDevelopmentCertificate: true - IsExportable: true

Run the command with both --check and --trust options to ensure that the certificate is not only valid but also trusted.

(base) PS C:\Users\pawel\Desktop\Desktop\z2\ShopAPI.Api> dotnet dev-certs https --trust

>>

Trusting the HTTPS development certificate was requested. A confirmation prompt will be displayed if the certificate was not previously trusted. Click yes on the prompt to trust the certificate.

Successfully trusted the existing HTTPS certificate.
```

2. **W przeglądarce Chrome włączyłem flagę Allow invalid certificates for resources loaded from localhost w celu umożliwienia korzystania z lokalnego certyfikatu developerskiego**.

3. **Sprawdzenie działania**

   Serwer uruchomiono na porcie HTTPS poleceniem:

```
dotnet run --launch-profile "https"
```

   Terminal pokazał, że serwer nasłuchuje na adresie:

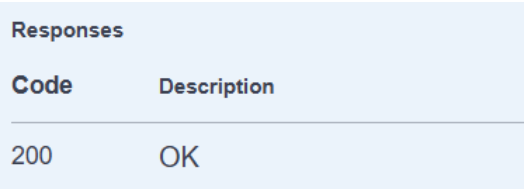
```
info: Microsoft.Hosting.Lifetime[14]
    Now listening on: https://localhost:7221
```

Po otwarciu tego adresu w przeglądarce dostępne było Swagger UI, a połączenie było szyfrowane i zaufane.

Zad 3

1. **Testowanie przez Swagger UI**
   Otworzyłem przeglądarkę i wszedłem pod adres **https://localhost:7221/swagger/index.html**, aby uzyskać dostęp do Swagger UI.

   W interfejsie znalazłem endpoint /weatherforecast, kliknąłem **Try it out**, a następnie **Execute**.



   Otrzymałem odpowiedź w formacie JSON oraz kod statusu HTTP 200 OK, co potwierdziło, że endpoint działa poprawnie.

2. **Testowanie przez curl**
   W terminalu VS Code przetestowałem endpoint /weatherforecast za pomocą polecenia:

```
(base) PS C:\Users\pawel\Desktop\Desktop\z2> curl https://localhost:7221/weatherforecast
```

   Odpowiedź:

```
[{"date":"2025-10-20","temperatureC":7,"summary":"Scorching","temperatureF":44},{"date":"2025-10-21","temperatureC":6,"summary":"Bracing","temperatureF":42},{"date":"2025-10-22","temperatureC":50,"summary":"Mild","temperatureF":121},{"date":"2025-10-23","temperatureC":8,"summary":"Bracing","temperatureF":46},{"date":"2025-10-24","temperatureC":29,"summary":"Chilly","temperatureF":84}]
```

   Otrzymałem odpowiedź w formacie JSON, która zawierała prognozy pogody dla pięciu dni, z polami: date, temperatureC, temperatureF i summary.
   Kod statusu HTTP 200 OK potwierdził poprawne działanie endpointu.

3. **Testowanie przez Postman**
   Uruchomiłem Postmana i wpisałem w nim adres endpointu:

```
https://localhost:7221/weatherforecast
```

   Po kliknięciu Send otrzymałem pełną odpowiedź w formacie JSON zawierającą prognozy pogody dla pięciu dni, z polami: date, temperatureC, temperatureF i summary.

   Kod statusu HTTP 200 OK potwierdził poprawne działanie endpointu.

4. **Testowanie przez pliki .http w VS Code**
   Zainstalowałem rozszerzenie Rest Client do VS code, następnie utworzyłem nowy plik **test.http** i umieściłem w nim proste żądanie get:

```
GET https://localhost:7221/weatherforecast
```

   W odpowiedzi otrzymałem odpowiedź w formacie JSON zawierającą prognozy pogody dla pięciu dni.
   Kod statusu HTTP 200 OK potwierdził poprawne działanie endpointu.

5. **Testowanie w przeglądarce (GET)**
   Uruchomiłem serwer API i w pasku adresu przeglądarki wpisałem:

```
https://localhost:7221/weatherforecast
```

   Po naciśnięciu Enter otrzymałem odpowiedź w formacie JSON zawierającą prognozy pogody dla pięciu dni.

Zad 4

1. **Utworzyłem klasę CitiesController w folderze Controllers, dodałem tam wsyztskie metody, o których była mowa w treści zadania:**
- **GET – pobieranie danych, pojedynczych i wszystkich miast**
- **POST – dodanie nowego miasta**
- **PUT – aktualizacja całego zasobu**
- **PATCH – częściowa aktualizacja, np. tylko pola Country**
- **DELETE – usunięcie zasobu**
- **HEAD – sprawdzenie istnienia zasobu**
- **OPTIONS – lista dostępnych metod HTTP dla endpointu.**

2. **Model City został umieszczony w osobnym pliku City.cs:**

```
public class City

{

   public int Id { get; set; }

   public string Name { get; set; } = string.Empty;

   public string Country { get; set; } = string.Empty;

}
```

3. **Dodatkowo musiałem skonfigurować Program.cs, aby kontrolery działały i były wykrywane poprawnie:**

```
builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();
```

4. **Testowanie endpointów wykonałem w VS Code przy pomocy pliku test.http, wysyłając wszystkie zaimplementowane operacje i sprawdzając odpowiednie kody statusu HTTP oraz poprawność zwracanych danych.**

Zad 5

W projekcie utworzyłem folder Dtos i plik CityDto.cs, który zawiera klasę:

```
PUBLIC CLASS CITYDTO
{
   PUBLIC STRING NAME { GET; SET; } = STRING.EMPTY;
   PUBLIC STRING COUNTRY { GET; SET; } = STRING.EMPTY;
}
```

DTO (Data Transfer Object) jest obiektem służącym do wymiany danych między API a klientem lub innymi warstwami aplikacji.

Jego głównym zadaniem jest oddzielenie warstwy prezentacji od logiki biznesowej, dzięki czemu przesyłane są tylko niezbędne informacje. Stosowanie DTO zwiększa bezpieczeństwo aplikacji i ułatwia wprowadzanie przyszłych zmian w modelach danych bez wpływu na interfejs API.

Zad 6

1. **Utworzenie zmian w CitiesController:**
   POST /api/cities
   - Dodanie nowego miasta zwraca 201 Created.
   - Nagłówek Location wskazuje URI nowo utworzonego zasobu.
   - Walidacja danych wejściowych: brak nazwy miasta skutkuje 400 Bad Request.

   PUT /api/cities/{id}
   - Aktualizacja istniejącego miasta zwraca 204 No Content.
   - Próba aktualizacji nieistniejącego miasta zwraca 404 Not Found.

   PATCH /api/cities/{id}
   - Częściowa aktualizacja (zmiana np. pola Country) zwraca 204 No Content.
   - Próba zmiany nazwy na już istniejącą powoduje 409
2. **Aby dokonać testów utworzyłem nowy plik testZ5.http:**

```
### zwróć 201 Created z nagłówkiem Location
POST https://localhost:7221/api/cities
Content-Type: application/json
{
  "name": "Rome",
  "country": "Italy"
}
### zwróć 404, jeśli miasto nie istnieje
PUT https://localhost:7221/api/cities/999
Content-Type: application/json
{
  "name": "Nowhere",
  "country": "Unknown"
```

```
}
### zwróć 409, jeśli próba zmiany koliduje z innymi danymi
PATCH https://localhost:7221/api/cities/2
Content-Type: application/json

{
   "Name": "Rome"
}
```

3. **Wyniki testów:**

```
HTTP/1.1 201 Created
Connection: close
Content-Type: application/json; charset=utf-8
Date: Sun, 19 Oct 2025 13:44:47 GMT
Server: Kestrel
Location: https://localhost:7221/api/Cities/7
Transfer-Encoding: chunked

{
  "id": 7,
  "name": "Rome",
  "country": "Italy"
}
HTTP/1.1 404 Not Found
Connection: close
Content-Type: application/problem+json; charset=utf-8
Date: Sun, 19 Oct 2025 13:45:03 GMT
Server: Kestrel
Transfer-Encoding: chunked

{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.5",
  "title": "Not Found",
  "status": 404,
  "traceId": "00-72298e453650cb6b92f66a6c93d8c227-2c88ea8a5ebb982e-00"
}
HTTP/1.1 409 Conflict
Connection: close
Content-Type: text/plain; charset=utf-8
Date: Sun, 19 Oct 2025 13:45:16 GMT
Server: Kestrel
Transfer-Encoding: chunked

City with this name already exists.
```

Zad 7

1. **Po otwarciu „https://localhost:7221/swagger/v1/swagger.json” , znalazłem definicje CityDto:**

```
"CityDto": {
 "type": "object",
 "properties": {
  "name": { "type": "string", "nullable": true },
  "country": { "type": "string", "nullable": true }
 },
 "additionalProperties": false
}
```

2. **NSwag automatycznie odwzorowuje modele i endpointy w formie metod asynchronicznych. Wynik wygenerowanego klienta wygląda tak:**

```
public class CityDto
{
  public string? Name { get; set; }
  public string? Country { get; set; }
}

public class CitiesClient
{
  private readonly HttpClient _httpClient;

  public CitiesClient(HttpClient httpClient)
  {
    _httpClient = httpClient;
  }
  public async Task<List<CityDto>> GetAllAsync()
  {
    return await _httpClient.GetFromJsonAsync<List<CityDto>>("/api/Cities");
  }
  public async Task<CityDto> CreateAsync(CityDto city)
  {
    var response = await _httpClient.PostAsJsonAsync("/api/Cities", city);
    response.EnsureSuccessStatusCode();
    return await response.Content.ReadFromJsonAsync<CityDto>();
  }
}
```

3. **Dzięki wykorzystaniu definicji CityDto z pliku Swagger (swagger.json) udało się stworzyć klasę klienta C# (CitiesClient).**
   Klasa ta umożliwia bezpośrednią komunikację z API poprzez metody asynchroniczne (GetAllAsync, CreateAsync), bez konieczności ręcznego pisania kodu do wysyłania żądań HTTP i parsowania odpowiedzi.
   Dzięki temu proces integracji aplikacji klienckiej z API jest znacznie prostszy, bezpieczniejszy i mniej podatny na błędy.

Zad 8

1. **Utworzenie aplikacji konsolowej**
   Stworzyłem nowy projekt ShopAPI.Client i dodałem pakiet System.Net.Http.Json, co pozwoliło na łatwe wysyłanie zapytań HTTP oraz odbiór i wysyłkę danych w formacie JSON.

2. **Pobieranie danych z API**
   Wykorzystałem metodę GetFromJsonAsync<List<CityDto>>("/api/cities"), aby pobrać listę miast i wyświetlić ich nazwy w konsoli. Dzięki temu zweryfikowałem, że API poprawnie zwraca wszystkie zasoby.

3. **Praktyczne wykorzystanie DTO**
   Klasa CityDto została wykorzystana zarówno do odbioru, jak i wysyłki danych, co oddziela strukturę przesyłaną w API od modelu domenowego i zapewnia spójność oraz bezpieczeństwo danych.

```
class Program{
  static async Task Main(string[] args)
  {
    var client = new HttpClient();
    client.BaseAddress = new Uri("https://localhost:7221");
    // GET
    var cities = await client.GetFromJsonAsync<List<CityDto>>("/api/cities");
    Console.WriteLine($"Miasta: {string.Join(", ", cities.Select(c => c.Name))}");
    // POST
    var newCity = new CityDto { Name = "Warsaw", Country = "Poland" };
    var response = await client.PostAsJsonAsync("/api/cities", newCity);
    Console.WriteLine($"Dodano miasto, status: {response.StatusCode}");
  }
}
```

Wnioski

Dzięki realizacji tego ćwiczenia nauczyłem się tworzyć i testować API REST w .NET, implementować kontroler CitiesController z pełnym zestawem operacji CRUD oraz stosować DTO do bezpiecznej wymiany danych. Poznałem także generowanie klienta C# za pomocą NSwag, co pozwoliło mi łatwo integrować aplikację konsolową z API i praktycznie wykorzystać asynchroniczne metody HTTP.