Analiza kodu ProductsController

Poniższy dokument opisuje działanie kontrolera API w ASP.NET Core, który obsługuje operacje CRUD dla produktów.

1. Importowanie przestrzeni nazw

Kod na początku pliku importuje niezbędne przestrzenie nazw:

```
using Core.Entities;
using Infrastructure.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
```

- Core.Entities – zawiera definicję encji Product.

- Infrastructure.Data – prawdopodobnie zawiera StoreContext, czyli klasę kontekstu bazy danych.

- Microsoft.AspNetCore.Mvc – umożliwia tworzenie kontrolerów API.

- Microsoft.EntityFrameworkCore – obsługuje operacje na bazie danych przy użyciu Entity Framework Core.

2. Definicja kontrolera

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
```

- [ApiController] – informuje, że jest to kontroler API.

- [Route("api/[controller]")] – ustala, że ścieżka URL dla tego kontrolera to api/products.

- ProductsController dziedziczy po ControllerBase, co oznacza, że nie obsługuje widoków (czysty REST API).

3. Wstrzykiwanie kontekstu bazy danych

```
private readonly StoreContext context;

public ProductsController(StoreContext context)
{
```

```
    this.context = context;
}
```

Dzięki Dependency Injection kontroler może komunikować się z bazą danych poprzez StoreContext.

4. Operacje CRUD

4.1 Pobranie wszystkich produktów

```
[HttpGet]
public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
{
    return await context.Products.ToListAsync();
}
```

Metoda zwraca listę wszystkich produktów w bazie.

4.2 Pobranie pojedynczego produktu

```
[HttpGet("{id:int}")]
public async Task<ActionResult<Product>> GetProduct(int id)
{
    var product = await context.Products.FindAsync(id);

    if (product == null) return NotFound();

    return product;
}
```

Metoda wyszukuje produkt po ID i zwraca go lub zwraca 404, jeśli nie znaleziono.

4.3 Tworzenie nowego produktu

```
[HttpPost]
public async Task<ActionResult<Product>> CreateProduct(Product product)
{
    context.Products.Add(product);
    await context.SaveChangesAsync();
    return product;
}
```

Metoda dodaje nowy produkt do bazy i zapisuje zmiany.

4.4 Aktualizacja produktu

```
[HttpPut("{id:int}")]
public async Task<ActionResult> UpdateProduct(int id, Product product)
{
   if (product.Id != id || !ProductExists(id))
      return BadRequest("Cannot update this product");

   context.Entry(product).State = EntityState.Modified;
   await context.SaveChangesAsync();
   return NoContent();
}
```

Metoda aktualizuje istniejący produkt, sprawdzając, czy ID się zgadza.

4.5 Usunięcie produktu

```
[HttpDelete("{id:int}")]
public async Task<ActionResult> DeleteProduct(int id)
{
   var product = await context.Products.FindAsync(id);
   if (product == null) return NotFound();

   context.Products.Remove(product);
   await context.SaveChangesAsync();
   return NoContent();
}
```

Metoda usuwa produkt z bazy, jeśli istnieje.

4.6 Metoda pomocnicza

```
private bool ProductExists(int id)
{
   return context.Products.Any(x => x.Id == id);
}
```

Metoda sprawdza, czy produkt o danym ID istnieje w bazie.

5. Podsumowanie

Kontroler ProductsController implementuje pełny CRUD dla produktów przy użyciu Entity Framework Core. Wykorzystuje wstrzykiwanie zależności i operacje asynchroniczne.