Лабораторна робота 2. Додавання database context до мікросервісу Products. Отримання всіх продуктів та одного продукту в мікросервісі Products.

Мета. Вивчити принципи функціонування Entity Framework Core, освоїти побудову моделей, пов'язаних з бізнес-логікою .

Завдання: Додати database context до мікросервісу Products. Отримати всі продукти та один конкретний продукт в мікросервісі Products.

Порядок виконання

1. Створення database context для мікросервісу Products.

В цьому пункті лабораторної роботи ми додамо DB Context для мікросервісу Products проекту ЛР №1. DB Context відповідає за всі зміни в базі даних і він базується на Entity Framework Core, яка втілює технологію об'єктно-реляційного відображення від Microsoft.

Додамо папку Db до проекту. Тож розкрийте папку проекта ECommerce.Api.Products, натисніть праву кнопку на проекті, виберіть Add => New Folder і назвіть папку Db.

Далі ми створимо новий клас. Натисніть праву кнопку на папці Db, виберіть Add => Class, дайте ім'я ProductsDbContext. Пам'ятайте, що об'єкт цього класу відповідатимо за всі операції, пов'язані з базою даних.

Успадкуйте в файлі коду цей клас від DbContext і натиснувши праву кнопку на заголовку класу додайте простір імен Microsoft.EntityFrameworkCore. Чудово!

Далі додайте клас Product в папку Db. Це буде клас моделі для мікросервісу Products. Додайте властивості Id, Name, Price і Inventory до класу.

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int Inventory { get; set; }
}
```

Далі поверніться до класу ProductsDbContext. Додамо до класу автовластивість Products типу DbSet<Product>. Ви можете асоціювати цю властивість з таблицею бази даних.

Також додайте конструктор до класу ProductsDbContext.

```
public class ProductsDbContext : DbContext
{
 public DbSet<Product> Products { get; set; }

 public ProductsDbContext(DbContextOptions options): base(options)
 {

 }
}
```

Далі відкрийте файл Startup.cs і в метод ConfigureServices додайте перед першим рядком наступне:

```
services.AddDbContext<ProductsDbContext>(options =>
{
  options.UseInMemoryDatabase("Products");
});
```

Додайте також посилання на простір імен Microsoft.EntityFrameworkCore.

Тепер ми готові реалізувати products provider.

2. Створення провайдера продуктів.

Спершу треба реалізувати інтерфейс IProductsProvider. Але перед цим додайте нову папку з назвою Models до проекту Products.

Далі додайте в цю папку клас Product з тими ж властивостями, що має клас Product з папки Db. Об’єкти цього класу буде повертати провайдер, і цей клас ми будемо використовувати для доступу до бази даних.

Далі додайте нову папку з назвою Interfaces до проекту Products.

Далі ми створимо новий інтерфейс. Натисніть праву кнопку на папці Interfaces, виберіть Add => Class, дайте ім'я IProductsProvider. В вихідному коді змініть class на interface:

```
public interface IProductsProvider
{
  Task<(bool IsSuccess, IEnumerable<Models.Product> Products, string
ErrorMessage)> GetProductsAsync();
}
```

Також додайте простір імен (using ECommerce.Api.Products.Models;).

Асинхронний метод цього інтерфейсу повертає кортеж (tuple), однією з складових якого є сукупність всіх продуктів. Першою складовою цього кортежу є елемент IsSuccess, друга складова має тип IEnumerable<Product>, а третя складова – це рядок помилки.

Далі ми реалізуємо реальний клас ProductsProvider.

Додайте до проекту папку Providers, а всередину цієї папки додайте клас ProductsProvider. Цей клас повинен реалізувати інтерфейс IProductsProvider. Додайте посилання на простір імен (using ECommerce.Api.Products.Interfaces;).

Ми поки-що не готові реалізувати цей інтерфейс через те, що нам знадобляться додаткові об'єкти. Тому додайте до класу конструктор, через який буде впроваджено залежність від ProductsDbContext.

```
public class ProductsProvider : IProductsProvider
    {
        private readonly ProductsDbContext dbContext;

        public ProductsProvider(ProductsDbContext dbContext,. . .)
        {
            this.dbContext = dbContext;

        }
. . . .
```

Далі таким же способом впровадьте залежності від ILogger (це один з базових інтерфейсів .NET Core) та IMapper (пов'язаний з AutoMapper framework):

```
public class ProductsProvider : IProductsProvider
    {
        private readonly ProductsDbContext dbContext;
        private readonly ILogger<ProductsProvider> logger;
        private readonly IMapper mapper;

        public     ProductsProvider(ProductsDbContext     dbContext,
ILogger<ProductsProvider> logger, IMapper mapper)
        {
            this.dbContext = dbContext;
            this.logger = logger;
            this.mapper = mapper;

            SeedData();
        }
.  .  .  .  .
```

Не забудьте також додати відповідні простори імен.

Додайте також метод SeedData(). В цьому новому методі сформуйте якісь тестові дані для мікросервісу Products.

```
private void SeedData()
        {
            if (!dbContext.Products.Any())
            {
                dbContext.Products.Add(new  Db.Product()  {  Id  =  1,
Name = "Keyboard", Price = 20, Inventory = 100 });
                dbContext.Products.Add(new  Db.Product()  {  Id  =  2,
Name = "Mouse", Price = 5, Inventory = 200 });
                dbContext.Products.Add(new  Db.Product()  {  Id  =  3,
Name = "Monitor", Price = 150, Inventory = 1000 });
                dbContext.Products.Add(new  Db.Product()  {  Id  =  4,
Name = "CPU", Price = 200, Inventory = 2000 });
                dbContext.SaveChanges();
            }
        }
```

Додайте до проекту папку Profiles, тому що нам потрібно специфікувати відображення між моделлю Product і сутністю Product.

Додайте в цю папку новий клас ProductProfile. Успадкуйте його від AutoMapper.Profile. Створіть в класі конструктор без параметрів, в якому викличте generic-метод CreateMap<Db.Product, Models.Product>();

І на кінець ми повинні відредагувати клас Startup. Ми повинні визначити який з IProducs.Provider треба використати всередині DependencyInjectionContainer. Тому в методі ConfigureServices з першого рядка додаємо

```
services.AddScoped<IProductsProvider, ProductsProvider >();
services.AddAutoMapper(typeof(Startup));
```

Після цього ми готові реалізувати логіку ProductsProvider.

3. Отримання всіх продуктів в мікросервісі Products.

Тепер перейдемо до реалізації метода GetProductsAsync в класі ProductsProvider.

```
public  async  Task<(bool  IsSuccess,  IEnumerable<Models.Product>
Products, string ErrorMessage)> GetProductsAsync()
```

```
    {
        try
        {
           var products = await dbContext.Products.ToListAsync();
           if (products != null && products.Any())
           {
             var result = mapper.Map<IEnumerable<Db.Product>,
IEnumerable<Models.Product>>(products);
             return (true, result, null);
           }
             return (false, null, "Not found");
        }
         catch (Exception ex)
        {
           logger?.LogError(ex.ToString());
           return (false, null, ex.Message);
        }
    }
```

В цьому асинхронному методі ми за допомогою dbContext.Products.ToListAsync() отримуємо у вигляді списку всі продукти, далі, коли список не пустий, виконуємо відображення в потрібний формат і повертаємо дані у вигляді кортежу.

Далі поверніться в Solution Explorer і створіть в проекті нову папку Controllers. Система видасть вікно з повідомленням, що така папка вже існує. Це виникло через те, що на початку ми видалили папку не потрібного нам контролера WeatherController. Тому знайдіть зверху в Solution Explorer піктограму кнопки Show All Files і натисніть її. Ви побачите приховану папку Controllers. Натисніть на ній праву кнопку і виберіть пункт Include in Project.

Додайте в цю папку клас ProductsController. Він буде представляти публічний набір API для мікросервісу Products.

В ASP.NET Core всі контролери повинні спадкувати від класу ControllersBase. Тому зробіть це для класу ProductsController. Додайте відповідний простір імен.

Додайте також перед класом атрибути ApiController і Route(“api/products”).

Далі ми додамо метод, що відповідає методу get протокола HTTP. В результаті маємо наступне:

```
    [ApiController]
        [Route("api/products")]
        public class ProductsController : ControllerBase
        {
          private readonly IProductsProvider productsProvider;

          public ProductsController(IProductsProvider productsProvider)
          {
             this.productsProvider = productsProvider;
          }

          [HttpGet]
          public async Task<IActionResult> GetProductsAsync()
          {
             var result = await productsProvider.GetProductsAsync();
             if (result.IsSuccess)
             {
```

```
                return Ok(result.Products);
            }
              return NotFound();
          }
      }
```

Зауважте як через конструктор впроваджується залежність від ProductsProvider.

Чудово! Давайте протестуємо наш ProductsController. Для цього натисніть на кнопку IIS Express на панелі інструментів в верхній частині вікна. З'явиться вікно браузера з помилкою. Відредагуйте рядок URL: замість **weatherforecast** введіть **api/products**. Повторіть запит і ви побачите результат.

Поверніться в Solution Explorer, розкрийте Properties і потім файл launchSettings.json. Змініть в двох місцях **weatherforecast** на **api/products**. Тепер при натисканні кнопки IIS Express зразу буде видано результат.

Таким чином, ми побудували повнофункціональний контролер, який видає весь каталог продуктів з бази даних, що знаходиться в пам'яті.

4. Отримання одного продукту в мікросервісі Products.

Тепер нас цікавить повернення одного конкретного продукту. Подивимось як це можна зробити.

Перше, що треба зробити, - це додати метод GetProductAsync(int id) в клас ProductsProvider.

```
    public async Task<(bool IsSuccess, Models.Product Product, string
ErrorMessage)> GetProductAsync(int id)
    {
     try
     {
      var product = await dbContext.Products.FirstOrDefaultAsync(p =>
p.Id == id);

      if (product != null)
      {
       var result = mapper.Map<Db.Product, Models.Product>(product);
       return (true, result, null);
      }
       return (false, null, "Not found");
     }
      catch (Exception ex)
      {
       logger?.LogError(ex.ToString());
       return (false, null, ex.Message);
      }
    }
```

Далі треба відредагувати інтерфейс IProductsProvider, додавши до нього ще один метод.

```
    public interface IProductsProvider
        {
            Task<(bool IsSuccess, IEnumerable<Product> Products, string
ErrorMessage)> GetProductsAsync();
            Task<(bool IsSuccess, Product Product, string ErrorMessage)>
GetProductAsync(int id);
        }
```

І, на завершення, треба додати метод в ProductsController:

```
[HttpGet("{id}")]
 public async Task<IActionResult> GetProductAsync(int id)
 {
   var result = await productsProvider.GetProductAsync(id);
   if (result.IsSuccess)
   {
      return Ok(result.Product);
   }
     return NotFound(result.ErrorMessage);
 }
```

Далі протестуйте сервіс натиснувши кнопку IIS Express і набравши в кінці URL **api/products/1** або інший номер продукту.

ВИКОРИСТАНІ ДЖЕРЕЛА

Lynda - Azure Microservices with .NET Core for Developers (2020). – URL: https://www.lynda.com/Azure-tutorials/Azure-Microservices-NET-Core-Developers/2825264-2.html