Лабораторна робота 5. Синхронізація роботи мікросервісів. Тестування мікросервісів.

Мета. Вивчити процес взаємодії мікросервісів, освоїти побудову додатків з мікросервісною архітектурою в Visual Studio 2019.

Завдання: виконати тестування мікросервісів Orders, Products та Search. Забезпечити синхронізацію роботи мікросервісів.

Порядок виконання

Одним з недоліків синхронного зв'язку є те, що доступність може бути зменшена. Це тому, що в цьому зв'язку задіяно більше одного мікросервісу. Давайте дізнаємось, як ми можемо покращити це, додавши в наші сервіси риси стійкості за допомогою Polly.

Polly - це бібліотека стійкості .NET і керування тимчасовими помилками, яку ми можемо використовувати для додавання таких політик, як автоматичний повторний запуск та автоматичний вимикач.

По-перше, давайте подивимося, що буде, якщо мікросервіс Products не працює. Тому клацніть правою кнопкою миші на файлі рішення Solution 'ECommerce', виберіть Set StartUp Projects і встановіть Action для мікросервісу Products як None, щоб запобігти роботі цього мікросервісу.

Далі протестуйте зроблене, натиснувши кнопку Start в верхній панелі.

Відкрийте Postman і натисніть кнопку Send. Відбудеться довге очікування відповіді і з'явиться наступне.

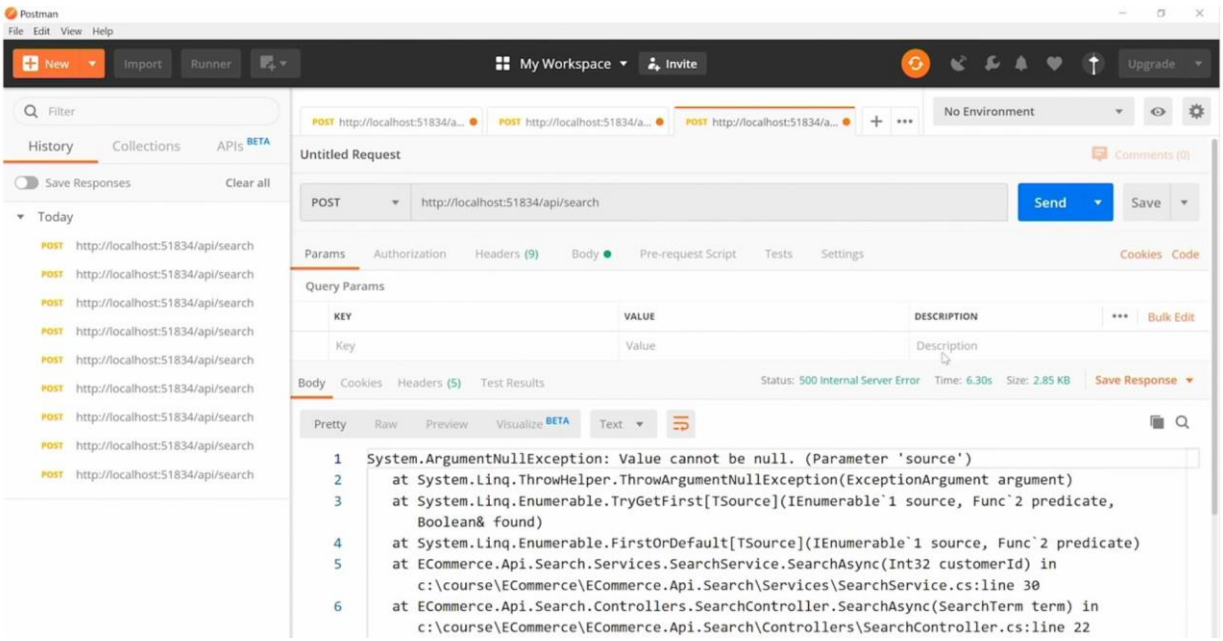

Рис. 1. Відповідь Postman з помилкою.

Це відбувається тому, що мікросервіс Projects не працює.

Тому поверніться до Visual Studio і натисніть кнопку Stop Debuggin.

І перше, що ми зробимо, - це додамо пакет Polly NuGet до цього проекту Search. Тому натисніть праву кнопку на Dependencies для проекту Search і виберіть Manage NuGet Packages. В пошуковому рядку наберіть Microsoft.Extensions.Http.Polly і після того, як пакет буде знайдено, натисніть Install. Погодьтесь з ліцензійною угодою.

Відкрийте файл Startup.cs, додайте простір імен

```
using Polly;
```

і в методі AddHttpClient для мікросервісу Products додайте використання AddTransientHttpErrorPolicy:

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddScoped<ISearchService, SearchService>();
            services.AddScoped<IOrdersService, OrdersService>();
            services.AddScoped<IProductsService,
ProductsService>();
            services.AddHttpClient("OrdersService", config =>
            {
                config.BaseAddress = new
Uri(Configuration["Services:Orders"]);
            });
            services.AddHttpClient("ProductsService", config =>
            {
                config.BaseAddress = new
Uri(Configuration["Services:Products"]);
            }).AddTransientHttpErrorPolicy(p=>
p.WaitAndRetryAsync(5, _ => TimeSpan.FromMilliseconds(500)));
            services.AddControllers();
        }
```

Ми тут специфікували політику WaitAndRetry, 5 означає п'ять спроб, а далі йде функція, яка задає паузу між спробами.

Далі відредагуйте клас SearchService.

```
namespace ECommerce.Api.Search.Services
{
    public class SearchService : ISearchService
    {
        private readonly IOrdersService ordersService;
        private readonly IProductsService productsService;

        public     SearchService(IOrdersService     ordersService,
IProductsService productsService)
        {
            this.ordersService = ordersService;
            this.productsService = productsService;
        }
        public async Task<(bool IsSuccess, dynamic SearchResults)>
SearchAsync(int customerId)
        {
            var ordersResult = await
ordersService.GetOrdersAsync(customerId);
            var productsResult = await
productsService.GetProductsAsync();

            if (ordersResult.IsSuccess)
            {
```

```
                    foreach (var order in ordersResult.Orders)
                    {
                        foreach (var item in order.Items)
                        {

        item.ProductName = productsResult.IsSuccess ?

productsResult.Products.FirstOrDefault(p => p.Id == item.ProductId)?.Name
:
                                    "Product information is not available";
                        }
                    }


                    var result = new
                    {
                        Orders = ordersResult.Orders
                    };
                    return (true, result);
                }
                return (false, null);
            }
        }
    }
```

Далі протестуйте зроблене, натиснувши кнопку Start в верхній панелі. Відкрийте Postman і натисніть кнопку Send.

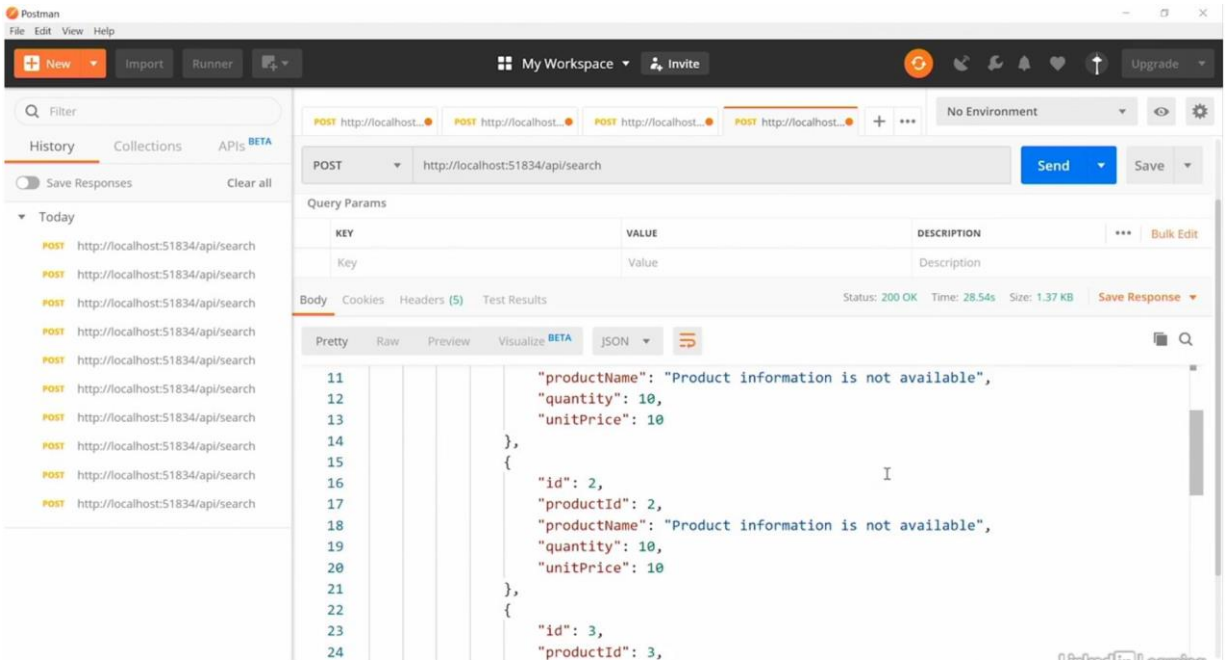

Рис. 2. Відповідь Postman у випадку використання Polly.

Як ви бачите в цьому випадку, productName буде "Product information is not available", тому що 5 спроб Polly не дали результат і прийшлось видавати значення за замовчуванням.

Зрозуміло, що якби ми використовували цю техніку у своїх мікросервісах, ми могли б створити більш надійні програмні рішення.

Синхронізація зв'язку з мікросервісом Customers.

Мета полягає в тому, щоб ви отримали реальну практику, яка реалізує синхронний зв'язок між мікросервісами. Це потребує деякого часу.

Додайте наступні нові файли до проекту: інтерфейс ICustomersService, клас CustomersService, модель Customer.

Також треба вирішити наступні завдання:

- Реалізувати логіку CustomersService;
- Конфігурувати DI контейнер;
- Додати HttpClient об'єкт;
- Використати ICustomersService в SearchService;
- Повернути дані користувача.

рол.

Тож розглянемо, як здійснити комунікацію до мікросервісу Customers.

Додайте інтерфейс ICustomersService.

```
namespace ECommerce.Api.Search.Interfaces
{
    public interface ICustomersService
    {
        Task<(bool IsSuccess, dynamic Customer, string
ErrorMessage)> GetCustomerAsync(int id);
    }
}
```

Тут ми повертаємо динамічний об'єкт, а не конкретний модельний об'єкт. Це тому, що ми використовуємо всю інформацію про клієнтів, яку отримуємо від мікросервісу клієнтів. Але ви можете додати певний клас моделі та десеріалізувати дані до цього типу, якщо потрібно.

Далі реалізуйте цей інтерфейс в класі CustomersService, куди ми через конструктор вводимо залежності від IHttpClientFactory та ILogger.

```
namespace ECommerce.Api.Search.Services
{
    public class CustomersService : ICustomersService
    {
        private readonly IHttpClientFactory httpClientFactory;
        private readonly ILogger<CustomersService> logger;

        public CustomersService(IHttpClientFactory
httpClientFactory, ILogger<CustomersService> logger)
        {
            this.httpClientFactory = httpClientFactory;
            this.logger = logger;
        }
        public async Task<(bool IsSuccess, dynamic Customer, string
ErrorMessage)> GetCustomerAsync(int id)
        {
            try
            {
```

```
                    var client =
httpClientFactory.CreateClient("CustomersService");
                    var response = await
client.GetAsync($"api/customers/{id}");
                    if (response.IsSuccessStatusCode)
                    {
                        var content = await
response.Content.ReadAsByteArrayAsync();
                        var options = new JsonSerializerOptions() {
PropertyNameCaseInsensitive = true };
                        var result =
JsonSerializer.Deserialize<dynamic>(content, options);
                        return (true, result, null);
                    }
                    return (false, null, response.ReasonPhrase);
                }
                catch (Exception ex)
                {
                    logger?.LogError(ex.ToString());
                    return (false, null, ex.Message);
                }
            }
        }
    }
```

Далі, у асинхронному методі GetCustomerAsync, ми викликаємо мікросервіс Customers, використовуючи кінцеву точку api/customers/{id}.

Коли ми отримуємо відповідь, тоді ми десеріалізуємо дані, використовуючи клас JsonSerializer.

Також модифікуйте клас Startup.

```
    public void ConfigureServices(IServiceCollection services)
            {
                services.AddScoped<IProductsService,
ProductsService>();
                services.AddScoped<IOrdersService, OrdersService>();
             services.AddScoped<ICustomersService, CustomersService>();
                services.AddScoped<ISearchService, SearchService>();
                services.AddHttpClient("ProductsService", config =>
                {
                    config.BaseAddress = new
Uri(Configuration["Services:Products"]);
                }).AddTransientHttpErrorPolicy(p =>
p.WaitAndRetryAsync(5, _ => TimeSpan.FromMilliseconds(500)));
                services.AddHttpClient("OrdersService", config =>
                {
                    config.BaseAddress = new
Uri(Configuration["Services:Orders"]);
                });
                services.AddHttpClient("CustomersService", config =>
                {
                    config.BaseAddress = new
Uri(Configuration["Services:Customers"]);
                });
                services.AddControllers();
            }
```

Отже, ми тут додали дані щодо конкретної реалізації інтерфейса ICustomersService у контейнер для ін'єкцій залежностей. Ми також додали

HttpClient object з посиланням на сервіс Customers в секції Services файлу appsettings.json.

Нарешті, змініть клас SearchService, ввівши залежність від ICustomersService, і, як ви можете бачити тут, змінюючи виконання методу SearchAsync з викликом GetCustomerAsync.

```
namespace ECommerce.Api.Search.Services
{
    public class SearchService : ISearchService
    {
        private readonly IProductsService productsService;
        private readonly IOrdersService ordersService;
        private readonly ICustomersService customersService;

        public SearchService(IProductsService productsService,
            IOrdersService ordersService, ICustomersService
customersService)
        {
            this.productsService = productsService;
            this.ordersService = ordersService;
            this.customersService = customersService;
        }
        public async Task<(bool IsSuccess, dynamic SearchResults)>
SearchAsync(int customerId)
        {
            var customersResult = await
customersService.GetCustomerAsync(customerId);
            var ordersResult = await
ordersService.GetOrdersAsync(customerId);
            var productsResult = await
productsService.GetProductsAsync();

            if (ordersResult.IsSuccess)
            {
                foreach (var orders in ordersResult.Orders)
                {
                    foreach (var item in orders.Items)
                    {
                        item.ProductName = productsResult.IsSuccess
?

productsResult.Products.FirstOrDefault(p => p.Id == item.ProductId)?.Name
:
                                "Product information is not available";
                    }
                }
                var result = new
                {
                    Customer = customersResult.IsSuccess ?
                               customersResult.Customer :
                               new { Name = "Customer information
is not available"},
                    Orders = ordersResult.Orders
                };

                return (true, result);
            }
            return (false, null);
```

```
            }
        }
}
```

Далі протестуйте зроблене, натиснувши кнопку Start в верхній панелі. Відкрийте Postman, натисніть кнопку Send і чекайте на результат.

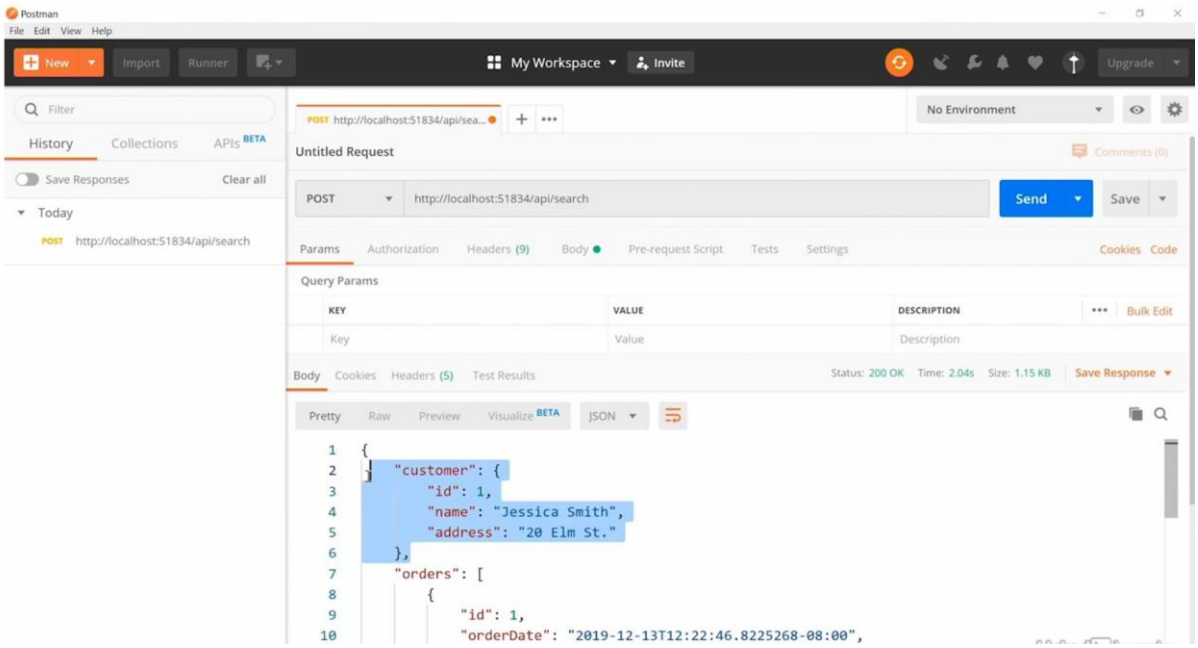

Рис. 3. Відповідь Postman містить інформацію про клієнта.

Як ви бачите тут, ми отримуємо інформацію про клієнта, яка надходить від мікросервісу Customers.

Будь ласка, врахуйте, що ми можемо також додати політику Polly для клієнтів HttpClient мікросервісу Customers.

Мікросервіси принесуть більше шкоди, ніж користі, якщо у вас немає культури DevOps у вашій організації [1]. DevOps - це об'єднання людей, процесів та технологій для постійного надання цінності клієнтам.

Одна з найбільш значущих переваг DevOps - це автоматизація процесів.

Автоматичне виконання unit тестів - лише один із прикладів автоматизації процесів. Мета unit тесту - перевірити правильність поведінки кожного блоку системи.

Створимо новий проект у рішенні на основі шаблону тестового проекту xUnit. Натисніть праву кнопку на Solution ‘ECommerce’ і виберіть Add => New Project, знайдіть шаблон xUnit Test Project (.NET Core) і натисніть Next. Дайте ім'я ECommerce.Api.Products.Tests.

Одна з найважливіших частин перед тим, як написати UnitTest, - це визначити, що ми будемо перевіряти, а що ні. Існує навіть метрика покриття коду, яка відображає те, наскільки ваш код покритий тестами. Ми не будемо настільки одержимими цією метрикою, але це може дати правильну орієнтацію, як робити справи.

У цьому конкретному випадку ми збираємося перевірити функціональність постачальника продуктів, який повинен повертати деякі дані з заданого контексту даних.

Тож давайте додамо посилання на проект Products. Натисніть праву кнопку на Dependencies, виберіть Add Reference а потім ECommerce.Api.Products.

В файлі UnitTest1.cs перейменуйте клас UnitTest1 на ProductsServiceTest (разом з перейменуванням і файлу UnitTest1.cs), а метод Test1 - на GetProductsReturnsAllProducts.

Далі давайте створимо об'єкт постачальника продуктів ProductsProvider (підключивши також простір імен using ECommerce.Api.Products.Providers;).

```
namespace ECommerce.Api.Products.Tests
{
    public class ProductsServiceTest
    {
        [Fact]
     public void GetProductsReturnsAllProducts()
     {
//var productsProvider = new ProductsProvider();
     }
    }
}
```

Але ми ще не готові задати параметру конструктору ProductsProvider, тому поки-що закоментуйте доданий рядок.

Як ви можете згадати, ми використовуємо постачальник даних in memory для .NET Framework Core всередині мікросервісу Products, але це лише через часові обмеження. UnitTest нічого не знає про внутрішню роботу мікросервісу, тобто дані можуть надходити з будь-якого підключеного джерела. З цієї причини нам потрібно вказати якийсь набір даних для тестування.

Тож давайте створимо dbContext нашого продукту (з простору імен ECommerce.Api.Products.Db). Але перш ніж це зробити, нам потрібно додати посилання на NuGet пакет Microsoft.EntityFrameworkCore. Тож натисніть праву кнопку на Dependencies, виберіть Manage NuGet Packages, в рядку пошуку введіть Microsoft.EntityFrameworkCore, виберіть знайдений пакет і інсталюйте його.

Далі в Solution Explorer натисніть праву кнопку на проекті ECommerce.Api.Products.Tests і виберіть Rebuild.

Тепер повернемось до створення dbContext. Використовувати вираз типу var dbContext = ProductsDbContext(options) ми не можемо, тому що не готовий параметр options. Для його створення потрібно написати наступне:

```
namespace ECommerce.Api.Products.Tests
{
    public class ProductsServiceTest
    {
        [Fact]
     public void GetProductsReturnsAllProducts()
     {
var options = new DbContextOptionsBuilder<ProductsDbContext>()
    .UseInMemoryDatabase(nameof(GetProductsReturnsAllProducts))
    .Options;
var dbContext = new ProductsDbContext(options);
```

```
        }
    }
}
```

Тепер ми можемо створити кілька прикладних даних для нашого UnitTest.

Ми збираємося створити новий метод під назвою CreateProducts, і передати йому цей об'єкт dbContext.

```
namespace ECommerce.Api.Products.Tests
{
    public class ProductsServiceTest
    {
        [Fact]
     public void GetProductsReturnsAllProducts()
     {
var options = new DbContextOptionsBuilder<ProductsDbContext>()
    .UseInMemoryDatabase(nameof(GetProductsReturnsAllProducts))
    .Options;
var dbContext = new ProductsDbContext(options);
   CreateProducts(dbContext);
   private void CreateProducts(ProductsDbContext dbContext)
   {
      throw new NotImplementedException();
   }
     }
    }
}
```

В методі CreateProducts створіть 10 підроблених продуктів.

```
        private void CreateProducts(ProductsDbContext dbContext)
        {
            for (int i = 1; i <= 10; i++)
            {
                dbContext.Products.Add(new Product()
                {
                    Id = i,
                    Name = Guid.NewGuid().ToString(),
                    Inventory = i + 10,
                    Price = (decimal)(i * 3.14)
                });
            }
            dbContext.SaveChanges();
        }
```

Тепер давайте оцінемо, чи можемо ми вже створити об’єкт ProductsProvider. Ми можемо використовувати dbContext для першого параметру, нам не потрібен Logger, тому другим параметром можна передати null, а третім параметром нам потрібен Mapper. Тож це цікаво, оскільки нам потрібно створити mock object для цього картографа, або ми можемо створити real mapper на основі профілів, які є у мікросервісу. І ми думаємо, що це кращий вибір, тому ми збираємося створити об’єкт профілю продукту типу ProductProfile, який входить до простору імен Ecommerce.Api.Products.Profiles.

В результаті в фінальному вигляді ми будемо мати наступне:

```
using AutoMapper;
using ECommerce.Api.Products.Db;
using ECommerce.Api.Products.Profiles;
using ECommerce.Api.Products.Providers;
using Microsoft.EntityFrameworkCore;
using System;
```

```
using System.Linq;
using System.Threading.Tasks;
using Xunit;

namespace ECommerce.Api.Products.Tests
{
    public class ProductsServiceTest
    {
        [Fact]
        public async Task GetProductsReturnsAllProducts()
        {
            var options = new
DbContextOptionsBuilder<ProductsDbContext>()

.UseInMemoryDatabase(nameof(GetProductsReturnsAllProducts))
                .Options;
            var dbContext = new ProductsDbContext(options);
            CreateProducts(dbContext);

            var productProfile = new ProductProfile();
            var configuration = new MapperConfiguration(cfg =>
cfg.AddProfile(productProfile));
            var mapper = new Mapper(configuration);

            var productsProvider = new ProductsProvider(dbContext,
null, mapper);

            var product = await productsProvider.GetProductsAsync();
            Assert.True(product.IsSuccess);
            Assert.True(product.Products.Any());
            Assert.Null(product.ErrorMessage);
        }

        private void CreateProducts(ProductsDbContext dbContext)
        {
            for (int i = 1; i <= 10; i++)
            {
                dbContext.Products.Add(new Product()
                {
                    Id = i,
                    Name = Guid.NewGuid().ToString(),
                    Inventory = i + 10,
                    Price = (decimal)(i * 3.14)
                });
            }
            dbContext.SaveChanges();
        }
    }
}
```

Схоже, ми закінчили, давайте побудуємо це і запустимо UnitTest.

Тому в Solution Explorer натисніть праву кнопку на проекті ECommerce.Api.Products.Tests і виберіть Rebuild.

Далі натисніть праву кнопку на рядку **`public class ProductsServiceTest`** і виберіть Run Test(s).

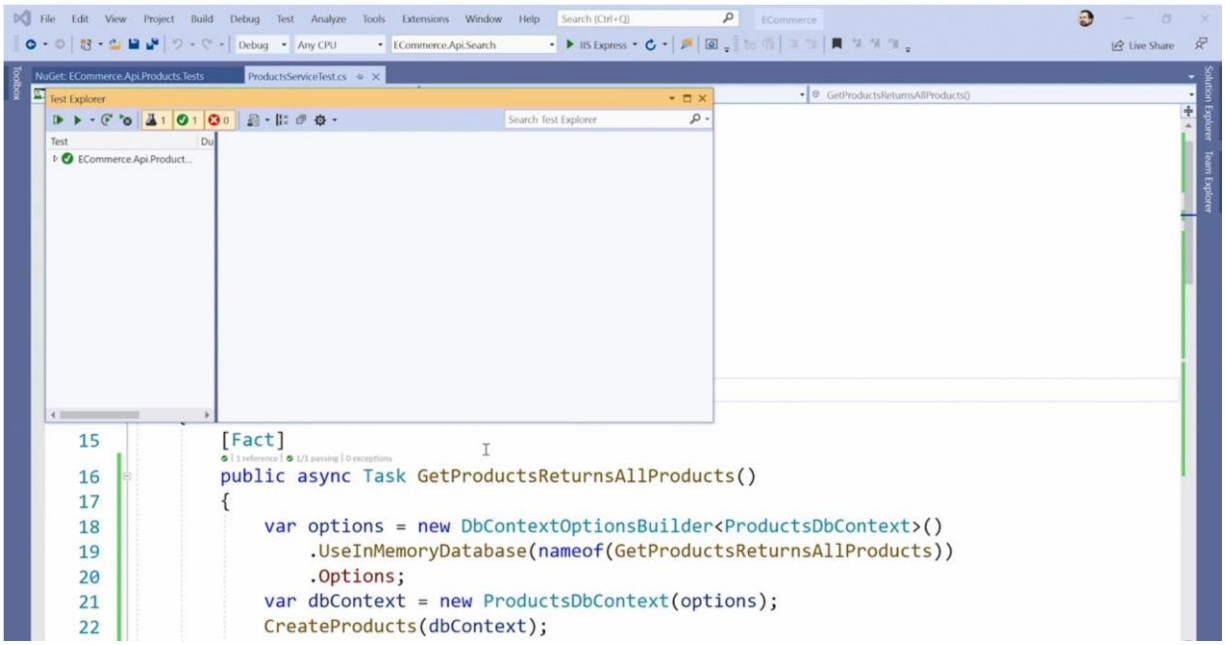

Рис. 4. Успішне проходження тесту.

Як бачите, тест був успішним.

ВИКОРИСТАНІ ДЖЕРЕЛА

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