Лабораторна робота 4. Мікросервіс Search. Отримання послуг мікросервісів Orders та Products з мікросервісу Search.

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

Завдання: Створити мікросервіс Search, організувати його взаємодію з мікросервісами Orders та Products.

Порядок виконання

Мікросервіси Orders, Products та Customers вже побудовані. Тепер побудуємо мікросервіс Search. Цей мікросервіс повинно будувати на основі об'єднання API. Ми збираємось побудувати операцію запиту шляхом виклику цих трьох мікросервісів, тому що вони володіють всіма даними.

В Solution Explorer розкрийте папку проекта ECommerce.Api.Search.

Додайте до проекту папку Models.

Додайте в папку Models клас SearchTerm.

```
namespace ECommerce.Api.Search.Models
{
    public class SearchTerm
    {
        public int CustomerId { get; set; }

    }
}
```

Цей клас має властивість CustomerId, яка буде відповідати ID користувача, для якого здійснюється пошук.

Далі додайте до проекту папку Interfaces.

Додайте в папку Interfaces інтерфейс ISearchService (спочатку додайте як клас, потім змініть слово class на слово interface).

```
namespace ECommerce.Api.Search.Interfaces
{
    public interface ISearchService
    {
        Task<(bool IsSuccess, dynamic SearchResults)>
SearchAsync(int customerId);
    }
}
```

Цей інтерфейс буде мати всі абстракції для нашого мікросервісу Search. В нашому випадку ми збираємось реалізувати один метод SearchAsync, який робить запит по customerId. Цей метод повертає задачу, всередині якої буде кортеж з двох елементів.

Далі поверніться в Solution Explorer і в папці Controllers створіть клас SearchController.

```
namespace ECommerce.Api.Search.Controllers
{
    [ApiController]
    [Route("api/search")]
```

```
        public class SearchController : ControllerBase
        {
            private readonly ISearchService searchService;

            public SearchController(ISearchService searchService)
            {
                this.searchService = searchService;
            }

            [HttpPost]

    public async Task<IActionResult> SearchAsync(SearchTerm term)
            {
         var result = await searchService.SearchAsync(term.CustomerId);
                if (result.IsSuccess)
                {
                    return Ok(result.SearchResults);
                }
                return NotFound();
            }
        }
    }
```

Цей клас декорований атрибутами ApiController та Route з маршрутом api/search.

Через конструктор класу ми впровадили залежність від ISearchService, яку зберегли в полі searchService.

Метод дії SearchAsync контролера приймає параметром об'єкт SearchTerm, асинхронно викликає через залежність метод SearchAsync, передаючи йому CustomerId, і повертає результати пошуку. Метод SearchAsync декоровано атрибутом HttpPost.

Далі поверніться в Solution Explorer, створіть папку Services, тому що нам потрібно створити реалізацію інтерфейса ISearchService.

В папці Services створіть клас SearchService.

```
    namespace ECommerce.Api.Search.Services
    {
        public class SearchService : ISearchService
        {
            public async Task<(bool IsSuccess, dynamic SearchResults)>
SearchAsync(int customerId)
            {
                await Task.Delay(1);
                return (true, new { Message = "Hello" });
            }
        }
    }
```

Тут ми поки-що не реалізуємо логіку пошуку, а просто спробуємо повернути якісь дані.

Тепер, коли ми реалізували інтерфейс ISearchService, ми повинні повідомити контейнеру Dependency Injection, що збираємось використовувати клас SearchService як конкретну реалізацію ISearchService. Для цього відкриваємо файл Startup.cs, і в метод ConfigureServices дописуємо першим рядком

```
    services.AddScoped<ISearchService, SearchService>();
```

Далі, цей мікросервіс Search повинен знати де знаходяться мікросервіси Orders, Products та Customers. В нашому конкретному випадку, наприклад, розгорніть проект ECommercr.Api.Customers, розгорніть Properties і в файлі launchSettings.json побачите, що “applicationURL” для цього сервіса є, наприклад, http://localhost:58323. Ми повинні це специфікувати в мікросервісі Search.

Тож відкрийте файл appsettings.json проекта ECommercr.Api.Search, і поставивши кому після останнього рядка (перед “}”), додайте секцію Services:

```
"Services": {
    "Customers": "http://localhost:58323",
    "Orders": "http://localhost:50385",
    "Products": "http://localhost:60319"
  }
```

Зауважте, що URL сервісів взяті з відповідних файлів launchSettings.json відповідних проектів.

Тепер конфігуруйте ваше рішення, натиснувши праву кнопку на Solution ‘ECommerce’ і вибравши Set StartUp Projects… Ви повинні стартувати всі сервіси для відлагодження мікросервісу Search.

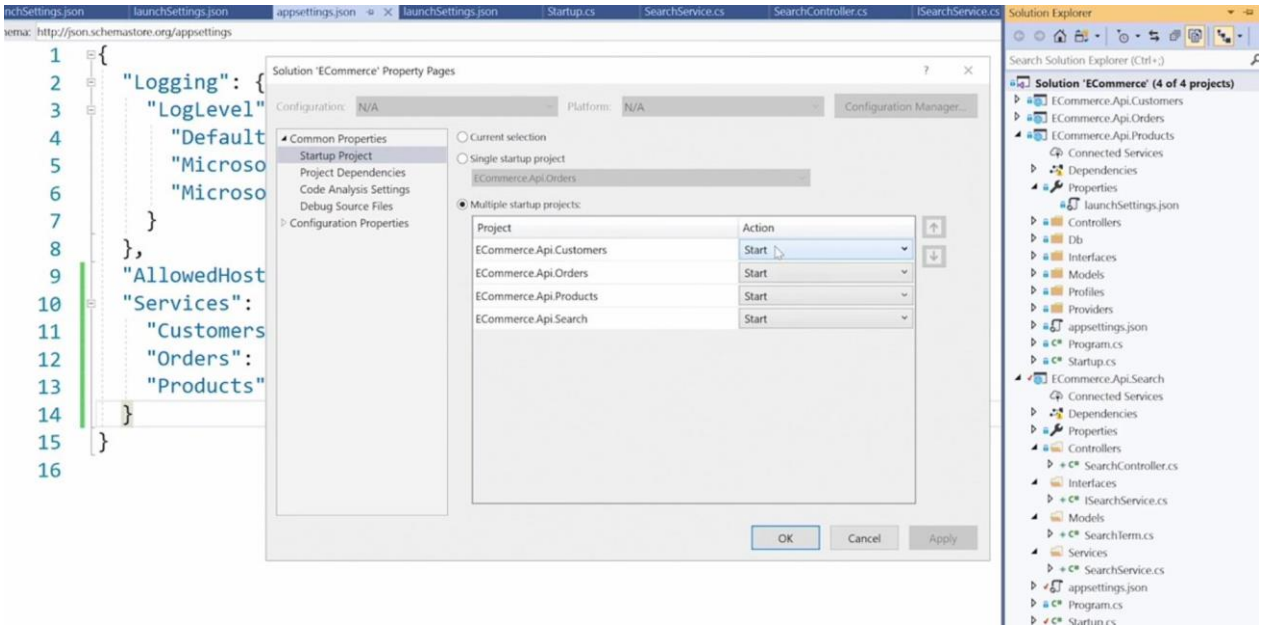

Рис. 1. Запуск усіх мікросервісів.

Таким чином, протестуйте мікросервіс Search, натиснувши кнопку Start в верхній панелі.

Після цього відкрийте застосунок Postman (попередньо його встановивши).

Ми можемо викликати мікросервіс Search, набравши в рядку POST http://localhost:51834/api/search (URL мікросервіса Search). Далі виберіть **Body**, радіокнопку **raw**, виберіть формат JSON і введіть запит

```
{
   “CustomerID”:1
}
```

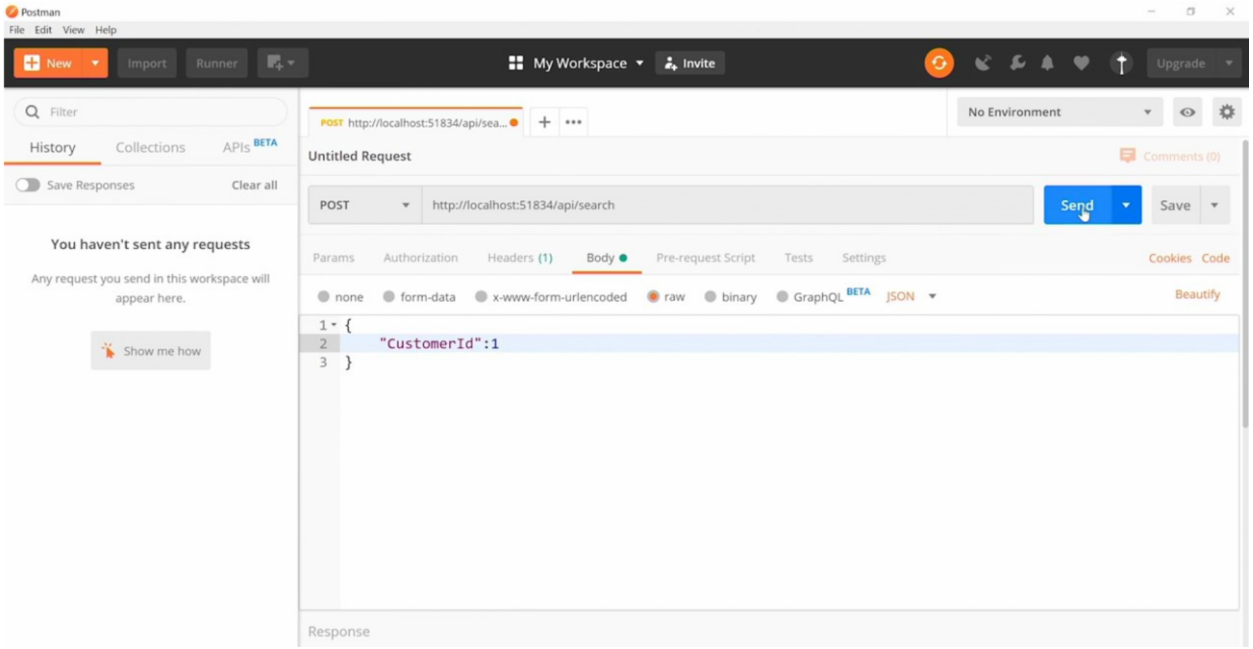

Рис. 2 Виклик мікросервіса Search через Postman.
Натисніть Send, і ви отримаєте результат.

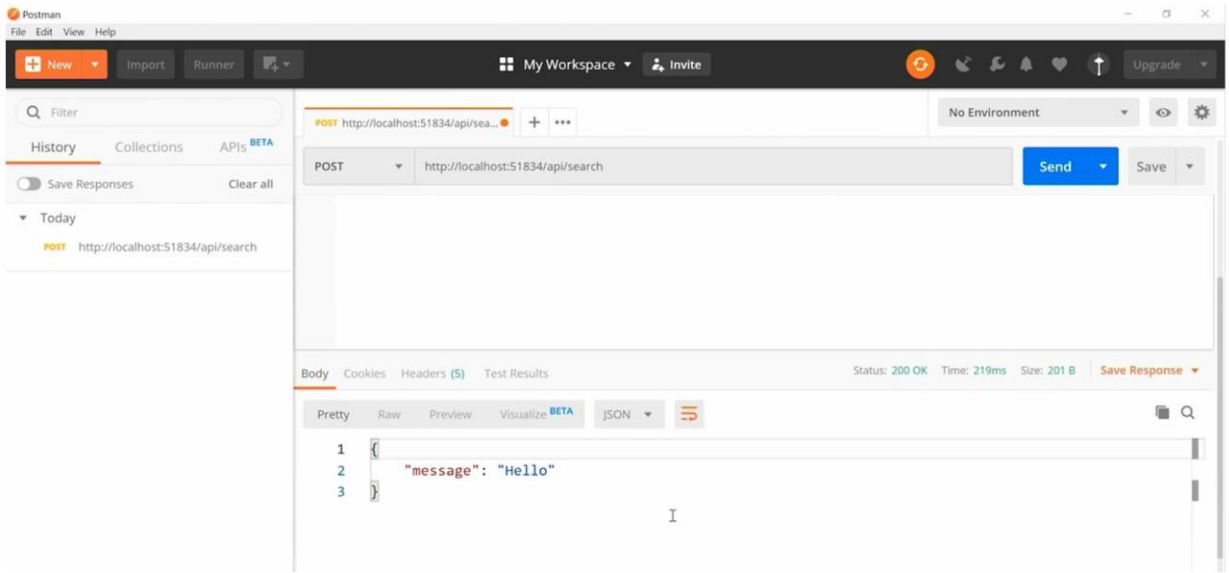

Рис. 3. Тестова відповідь мікросервіса Search.
Таким чином, ми впевнились, що мікросервіс Search працює.

А далі ми додамо виклик мікросервіса Orders.

Перше, що ми зробимо – конфігуруємо HTTP Client factory object, який ми будемо використовувати для комунікації з мікросервісом Orders. Відкрийте файл Startup.cs проекта ECommerce.Api.Search, і в метод ConfigureServices допишіть після першого рядка (те, що виділено жирним).

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddScoped<ISearchService, SearchService>();
 services.AddHttpClient("OrdersService", config =>
 {
    config.BaseAddress = new Uri(Configuration["Services:Orders"]);
```

```
        });
        services.AddControllers();
    }
```

Тут, якщо ви помітили, є посилання на секцію Services, яку ми дописали в файлі appsettings.json.

Далі ми додамо декілька класів в папку Models. Першим додайте клас Order.

```
namespace ECommerce.Api.Search.Models
{
    public class Order
    {
        public int Id { get; set; }
        public DateTime OrderDate { get; set; }
        public decimal Total { get; set; }
        public List<OrderItem> Items { get; set; }
    }
}
```

Другим додайте клас OrderItem.

```
namespace ECommerce.Api.Search.Models
{
    public class OrderItem
    {
        public int Id { get; set; }
        public int ProductId { get; set; }
        public string ProductName { get; set; }
        public int Quantity { get; set; }
        public decimal UnitPrice { get; set; }
    }
}
```

Зауважте, що ці класи відрізняються від одноіменних класів в проекті ECommerce.Api.Orders.

Далі поверніться в Solution Explorer і в папку Interfaces додайте інтерфейс IOrdersService, який знадобиться для комунікації з сервісом Orders.

```
namespace ECommerce.Api.Search.Interfaces
{
    public interface IOrdersService
    {
        Task<(bool IsSuccess, IEnumerable<Order> Orders, string
ErrorMessage)>
            GetOrdersAsync(int customerId);
    }
}
```

Далі ми додамо конкретну реалізацію цього інтерфейсу. Додайте в папку Services новий клас OrdersService.

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

        public OrdersService(IHttpClientFactory httpClientFactory,
ILogger<OrdersService> logger)
```

```
            {
                this.httpClientFactory = httpClientFactory;
                this.logger = logger;
            }
            public async Task<(bool IsSuccess, IEnumerable<Order>
Orders, string ErrorMessage)>
                GetOrdersAsync(int customerId)
            {
                try
                {
                    var client =
httpClientFactory.CreateClient("OrdersService");
                    var response = await
client.GetAsync($"api/orders/{customerId}");
                    if (response.IsSuccessStatusCode)
                    {
                        var content = await
response.Content.ReadAsByteArrayAsync();
                        var options = new JsonSerializerOptions() {
PropertyNameCaseInsensitive = true };
                        var result =
JsonSerializer.Deserialize<IEnumerable<Order>>(content, options);
                        return (true, result, null);
                    }
                    return (false, null, response.ReasonPhrase);
                }
                catch (Exception ex)
                {
                    logger?.LogError(ex.ToString());
                    return (false, null, ex.Message);
                }
            }
        }
    }
```

Тут за допомогою конструктора впроваджується залежності від IHttpClientFactory та ILogger<OrdersService>.

В асинхронному методі GetOrdersAsync спочатку створюється клієнт викликом методу CreateClient з вказуванням імені сервісу OrdersService. Потім відбувається виклик клієнта разом з викликом мікросервісу Orders.

Далі, в разі успіху, відповідь (response) десеріалізується. Ми отримуємо контент шляхом виклику метода ReadAsByteArrayAsync. І далі ми десеріалізуємо контент використовуючи метод Deserialize класу JsonSerializer. При цьому ми через змінну options вказуємо, що читати треба залежні від регістру дані. В блоці catch ми задіюємо Logger, залежність від якого ми впровадили раніше.

Далі відкрийте клас SearchService тому що нам потрібно впровадити залежність від IOrdersService. Внесіть наступні зміни.

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

            public SearchService(IOrdersService ordersService)
```

```
            {
                this.ordersService = ordersService;
            }
            public async Task<(bool IsSuccess, dynamic SearchResults)>
SearchAsync(int customerId)
            {
     var ordersResult = await ordersService.GetOrdersAsync(customerId);

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

І, на завершення, поверніться до файлу Startup.cs щоб специфікувати конкретне впровадження IOrdersService.

```
    public void ConfigureServices(IServiceCollection services)
            {
                services.AddScoped<ISearchService, SearchService>();
                services.AddScoped<IOrdersService, OrdersService>();
                services.AddHttpClient("OrdersService", config =>
                {
        config.BaseAddress = new Uri(Configuration["Services:Orders"]);
                });
                });
                services.AddControllers();
            }
```

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

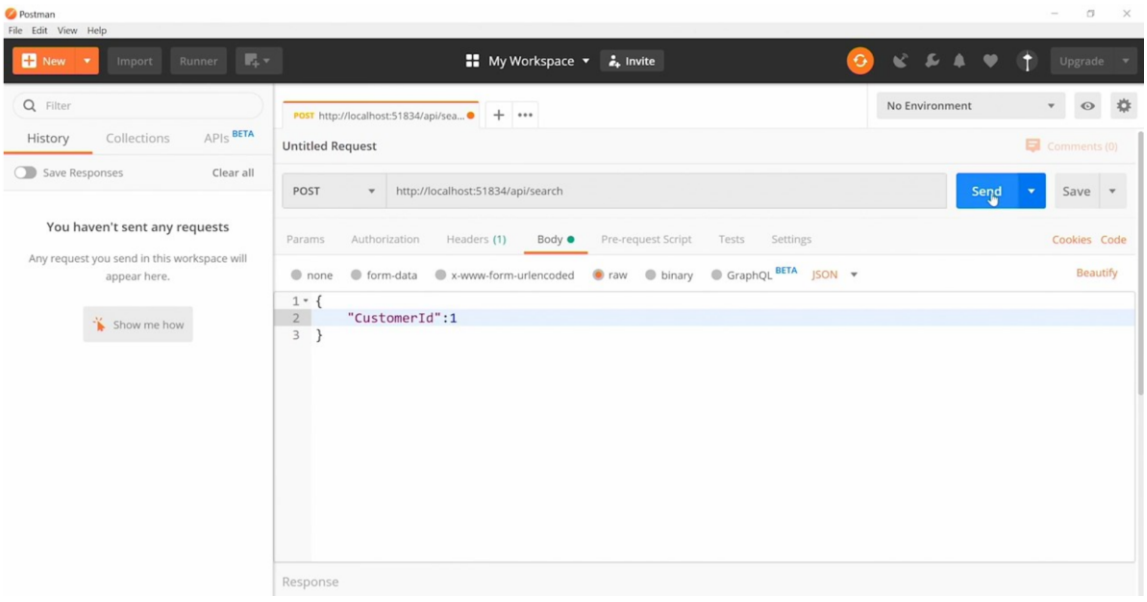

Рис. 4. Запуск Postman.

Ви повинні отримати результат з мікросервісу Orders.

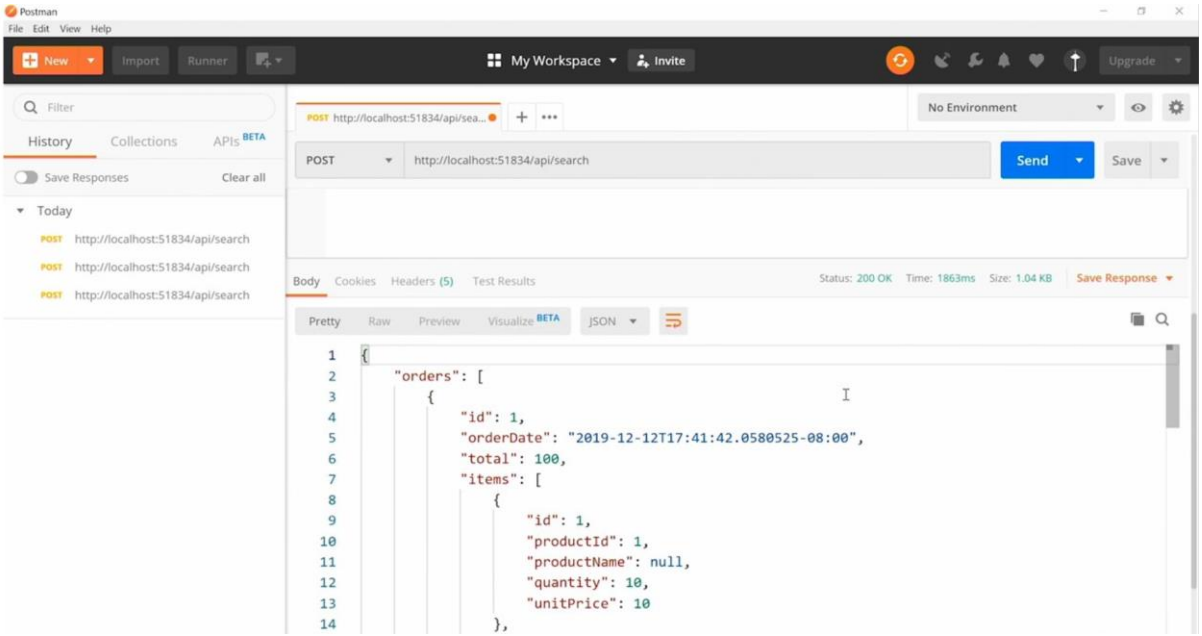

Рис. 5. Дані, отримані з мікросервісу Orders.

Спробуйте повторити запит, вказуючи інший номер користувача.

Таким чином, ми вже впровадили взаємодію з мікросервісом Orders.
А тепер додамо взаємодію з мікросервісом Products.
Спершу відкрийте файл Startup.cs щоб конфігурувати HTTP Client object, який ми збираємось використати для комунікації з мікросервісом Products.

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

Далі додайте в папку Models клас Product.

```
namespace ECommerce.Api.Search.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
}
```

Цей клас буде відрізнятися від подібного класу в проекті Products.

Далі додайте в папку Interfaces інтерфейс IProductsService.

```
namespace ECommerce.Api.Search.Interfaces
{
    public interface IProductsService
    {
        Task<(bool IsSuccess, IEnumerable<Product> Products, string
ErrorMessage)>
            GetProductsAsync();
    }
}
```

Потім додайте реалізацію цього інтерфейсу в конкретному класі. Тож додайте в папку Services новий клас ProductsService.

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

        public ProductsService(IHttpClientFactory httpClientFactory,
ILogger<ProductsService> logger)
        {
            this.httpClientFactory = httpClientFactory;
            this.logger = logger;
        }
        public async Task<(bool IsSuccess, IEnumerable<Product>
Products, string ErrorMessage)> GetProductsAsync()
        {
            try
            {
    var client = httpClientFactory.CreateClient("ProductsService");
    var response = await client.GetAsync("api/products");
                if (response.IsSuccessStatusCode)
                {
    var content = await response.Content.ReadAsByteArrayAsync();
    var     options     =     new     JsonSerializerOptions()     {
PropertyNameCaseInsensitive = true };
    var result =
JsonSerializer.Deserialize<IEnumerable<Product>>(content, options);
                    return (true, result, null);
                }
                return (false, null, response.ReasonPhrase);
            }
            catch (Exception ex)
            {
                logger?.LogError(ex.ToString());
                return (false, null, ex.Message);
            }
        }
    }
}
```

Тут за допомогою конструктора впроваджується залежності від IHttpClientFactory та ILogger<ProductsService>.

В асинхронному методі GetProductsAsync спочатку створюється клієнт викликом методу CreateClient з вказуванням імені сервісу ProductsService. Потім

відбувається виклик клієнта разом з викликом мікросервісу Products вказуванням кінцевої точки api/products.

Далі, в разі успіху, відповідь (response) десеріалізується. Ми отримуємо контент шляхом виклику метода ReadAsByteArrayAsync. І далі ми десеріалізуємо контент використовуючи метод Deserialize класу JsonSerializer. При цьому ми через змінну options вказуємо, що читати треба залежні від регістру дані. В блоці catch ми задіюємо Logger, залежність від якого ми впровадили раніше.

Далі давайте використаємо ProductsService всередині SearchService.

Відкрийте клас SearchService тому що нам потрібно впровадити залежність від IOrdersService. Внесіть наступні зміни (вказані жирним шрифтом).

```
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.Products.FirstOrDefault(p => p.Id == item.ProductId)?.Name;
                    }
                }


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

Ми тут додали ім'я кожному продукту всередині кожного замовлення (два цикли foreach).

І, на завершення, поверніться до файлу Startup.cs щоб специфікувати конкретне впровадження IProductsService.

```
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"]);
            });
            services.AddControllers();
        }
```

Тепер ми готові тестувати наші мікросервіси. Тож натисніть кнопку Start в верхній панелі.

Відкрийте Postman і натисніть кнопку Send.

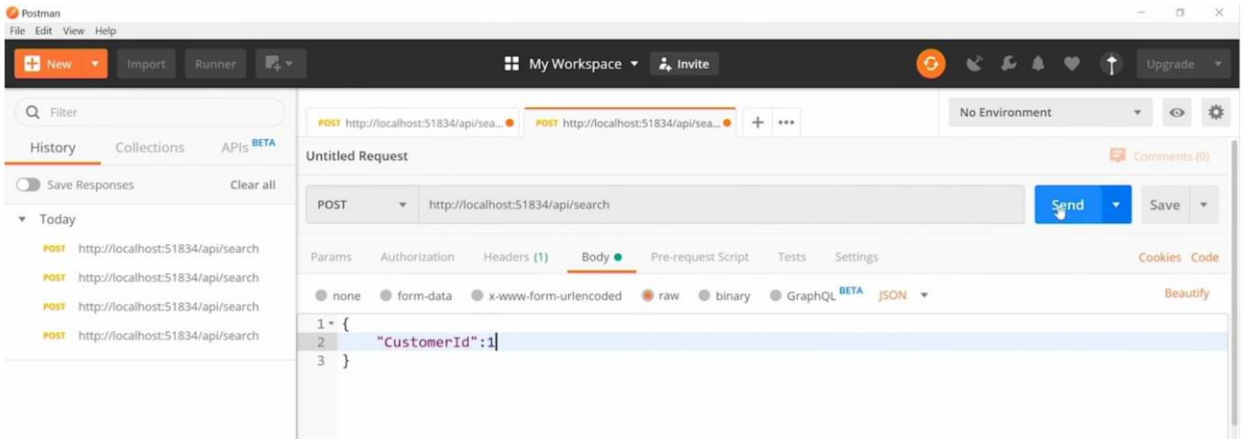

Рис. 6. Запит замовлень першого користувача в Postman.

І, як ви бачите (Рис. 7), ми отримали дані по замовленням користувача з CustomerId = 1 з мікросервісів Orders та Products.

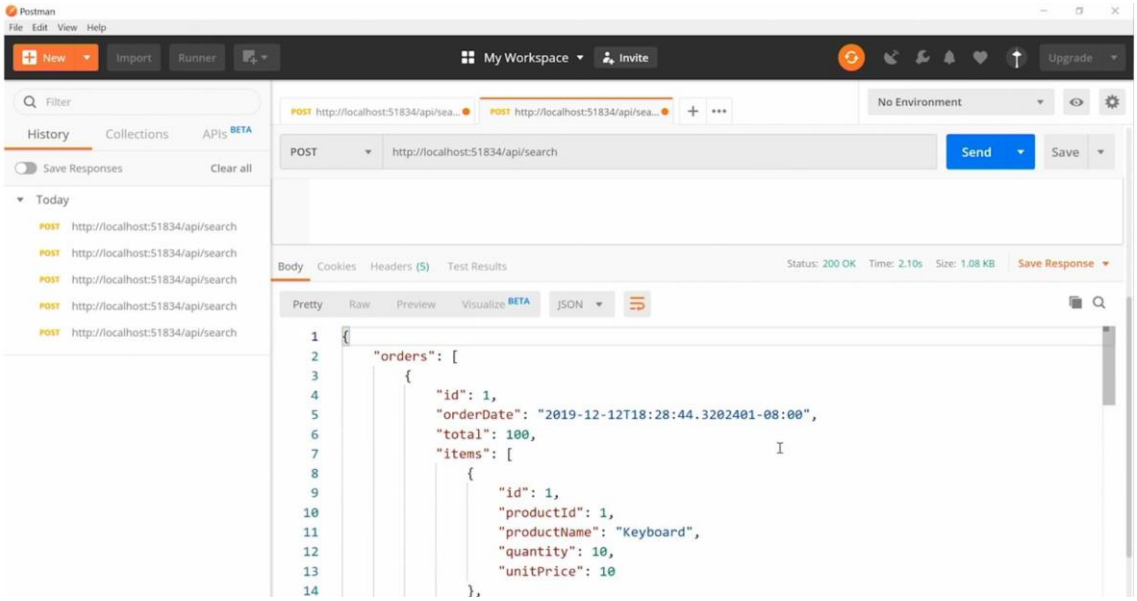

Рис. 7. Відповідь на запит в Postman.

Цікаво, а що буде, якщо мікросервіс Products впаде? Ми це розглянемо далі (в наступній лабораторній роботі).

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