Лабораторная работа №4

Работа с REST API

1. Цель работы.

Знакомство с принципом работы сервисов REST.

2. Задача работы

Научиться создавать REST API сервисы и взаимодействовать с ними из приложений.

Время выполнения работы: 2 часа

3. Выполнение работы

3.1. Создание проекта

Добавьте в решение проект ASP NET Core Web API.

Название проекта XXX.API, где XXX – название вашего решения.

Добавьте в проект ссылку на проект XXX.Domain.

В данном проекте будем использовать порты **7002** и **5002**. Сделайте соответствующие изменения в файле launchSettings.json.

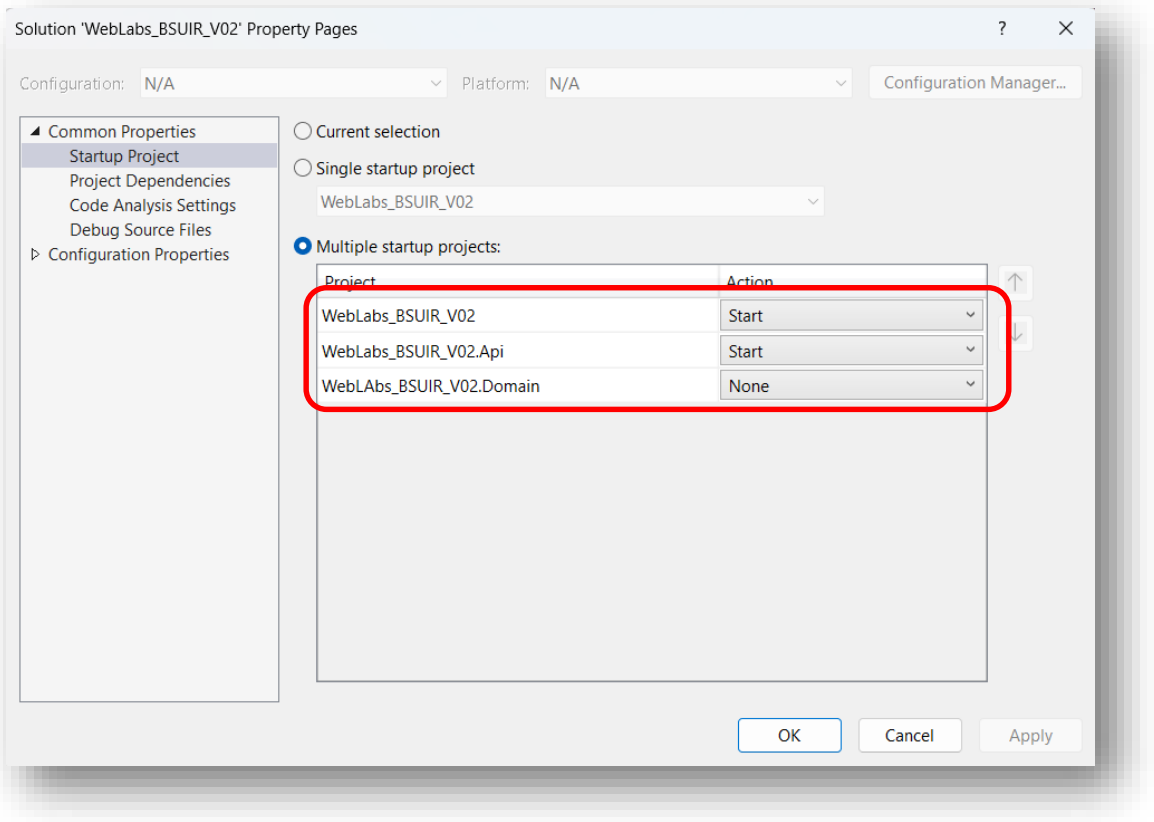
Добавьте следующие пакеты NuGet:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Tools

Добавьте в проект папку wwwroot/Images. В этой папке будут храниться файлы изображений. Скопируйте в нее файлы из соответствующей папки основного проекта.

Внимание: Для доступа к файлам изображений в классе program добавьте в конвейер Middleware компонент статических файлов.

В настройках решения укажите, чтобы запускались сразу оба проекта (такая возможность есть только в Visual Studio). В VScode проекты нужно будет запускать отдельно:



Чтобы при запуске проекта XXX.API не открывался браузер, в файле launchSettings.json закомментируйте строку

```
"launchBrowser": true,
```

3.2. Выбор БД

3.2.1. Возможные варианты

В качестве СУБД можно рассмотреть следующие реляционные базы данных:

- MS SqlServerExpress
- SqLite
- PosrgeSQL
- MySQL

Вы можете использовать любую другую СУБД

В приведенных ниже примерах используется PostgreSQL

3.2.2. SqLite

Для использования базы данных SqLite достаточно добавить в проект NuGet пакет поставщика SqLite для EntityFramework: **Microsoft.EntityFrameworkCore.Sqlite**.

3.2.3. PostgreSQL

Для работы с PostgreSQL добавьте в проект NuGet пакет **Aspire.Npgsql.EntityFrameworkCore.PostgreSQL**

3.3. Установка сервера PostgreSQL

PostgreSQL можно установить, скачав с официального сайта:

https://www.postgresql.org/download/

pgAdmin4 можно скачать с официального сайта:

https://www.pgadmin.org/download/

Ниже приводится установка в контейнере Docker.

Актуальные версии образов PostgreSQL можно найти сайте Docker Hub:

https://hub.docker.com/_/postgres

Актуальные версии образов pgAdmin4 можно найти сайте Docker Hub:

https://hub.docker.com/r/dpage/pgadmin4/

Создайте файл **compose.yaml**

В созданном файле опишите сервисы для Postgres и pgAdmin. Например:

```
services:
  postgres:
    image: postgres:17.5-bookworm
    container_name: postgres
    env_file:
      - postgres.env
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    ports:
```

```
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - postgres_network

  pgadmin:
    image: dpage/pgadmin4:9.4.0
    container_name: pgadmin4
    env_file:
      - postgres.env
    environment:
      PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL}
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD}
      PGADMIN_LISTEN_PORT: 5454
    ports:
      - "5454:5454"
    networks:
      - postgres_network
    depends_on:
      - postgres

volumes:
  postgres_data:
    driver: local

networks:
  postgres_network:
    driver: bridge
```

Переменные опишите в файле postgres.env:

```
POSTGRES_USER=admin
POSTGRES_PASSWORD=123456
POSTGRES_DB=postgres
PGADMIN_DEFAULT_EMAIL=admin@example.com
PGADMIN_DEFAULT_PASSWORD=123456
```

Примечание: значения переменных выберите сами.

Откройте консоль. Перейдите в папку, в которой расположен файл compose.yaml, и выполните команду:

docker compose --env-file postgres.env up –d

В браузере подключитесь к pgadmin (в приведенном файле compose указан порт 5454). Убедитесь, что сервер PosgreSql работает (обращаться к серверу нужно по имени – в приведенном файле compose указано имя posgres)

3.4. Создание контекста БД

В проект XXX.API добавьте папку Data

В папку Data добавьте класс контекста базы данных с именем AppDbContext. Конструктор класса должен принимать объект DbContextOptions<AppDbContext>.

Опишите в контексте свойства DbSet, содержащие коллекции сущностей предметной области из библиотеки XXX.Domain.

3.5. Начальная миграция БД

Добавьте в файл appsettings.json секцию, описывающую строку подключения к БД. В данном примере используется база данных PostgreSQL:

```
"ConnectionStrings": {
  "Postgres": "Server=127.0.0.1; Port=5432; Database=MenuDvb_V01; User Id=admin;
Password=123456"
}
```

Порт указан в файле compose, User и Password указаны в файле postgres.env.

Зарегистрируйте созданный контекст базы данных в качестве сервиса в файле program.cs (в опциях укажите `options.UseNpgsql(connString)`). Строку подключения к БД получите с помощью объекта builder.Configuration.

Создайте начальную миграцию.

Выполните команду Update для создания базы данных.

Откройте PgAgmin и убедитесь, что в PostgreSQL появилась ваша база данных.

3.6. Заполнение базы начальными данными

В папку Data добавьте класс DbInitializer.

Опишите в классе DbInitializer статический метод для заполнения базы исходными данными:

```
public static async Task SeedData(WebApplication app)
```

Для заполнения данными можно использовать код из класса MemoryProductSerbvice основного проекта, за исключением следующего:

– данные должны записываться не в коллекции, а в контекст БД с последующим сохранением в БД

– при создании объектов не должен указываться Id объекта, т.к. он будет назначен базой данных

– изображения будут сохраняться в папке **wwwroot/Images** проекта **XXX.Api**. (создайте соответствующую папку и поместите в нее файлы изображений)

3.6.1. Рекомендации к заданию 3.4.

– Контекст БД нужно получить из сервисов с помощью параметра `WebApplication app`, передаваемого в метод.

– Контекст БД – это «Scoped» сервис.

– Перед заполнением БД выполните миграцию

Пример:

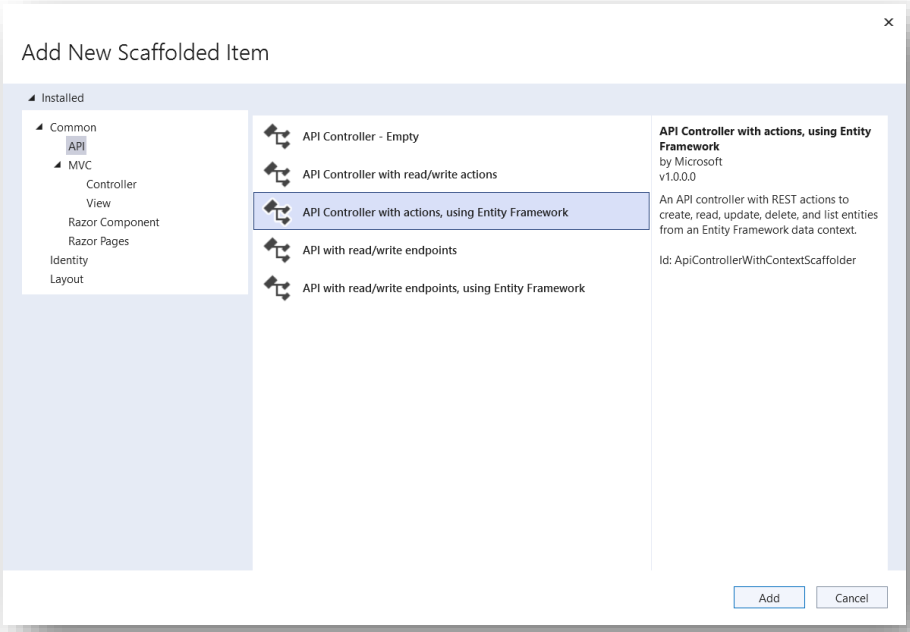
```
// Получение контекста БД
using var scope = app.Services.CreateScope();
var context =
    scope.ServiceProvider.GetRequiredService<AppDbContext>();
// Выполнение миграций
await context.Database.MigrateAsync();
```

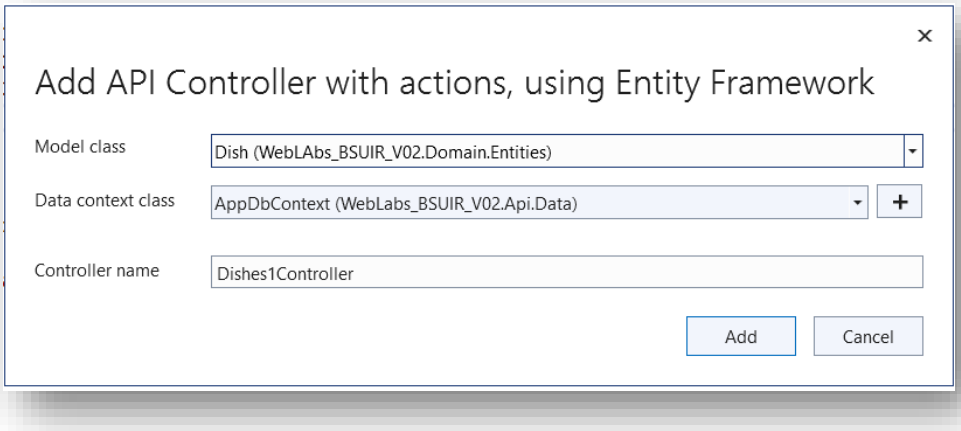
– Адрес изображения будет содержать Url приложения XXX.Api (в данном проекте это https://localhost:7002 – см. п.3.1). Добавьте в файл **appsettings.json** секцию с данным адресом. В методе SeedData получите адрес из этой секции с помощью **app.Configuration**.

3.7. Создание контроллера API

В папку Controllers проекта XXX.Api добавьте REST Api контроллер для ваших категорий объектов (групп объектов). Для этого:

В VisualStudio выберите Add-New scaffolded item или Add-Controller, в открывшемся окне выберите **API – API controller with actions, using EntityFramework**.





В VS Code или в VisualStudio for MAC:

Добавьте пакеты NuGet

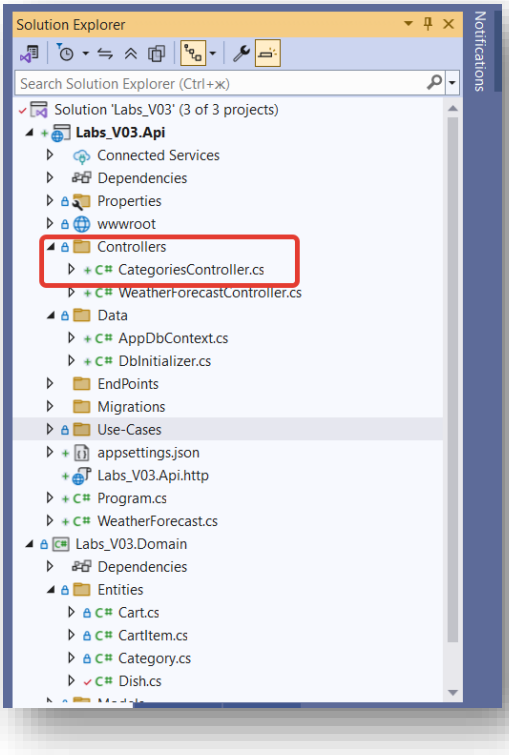
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design -v 7.0.0

dotnet add package Microsoft.EntityFrameworkCore.Design -v 7.0.0

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer -v 7.0.0
dotnet tool uninstall -g dotnet-aspnet-codegenerator
dotnet tool install -g dotnet-aspnet-codegenerator
```

Выполните команду:

```
dotnet aspnet-codegenerator controller -name [ИМЯ КОНТРОЛЛЕРА] -async -api
-m [КЛАСС МОДЕЛИ] -dc [КОНТЕКСТ БД] -outDir Controllers
```

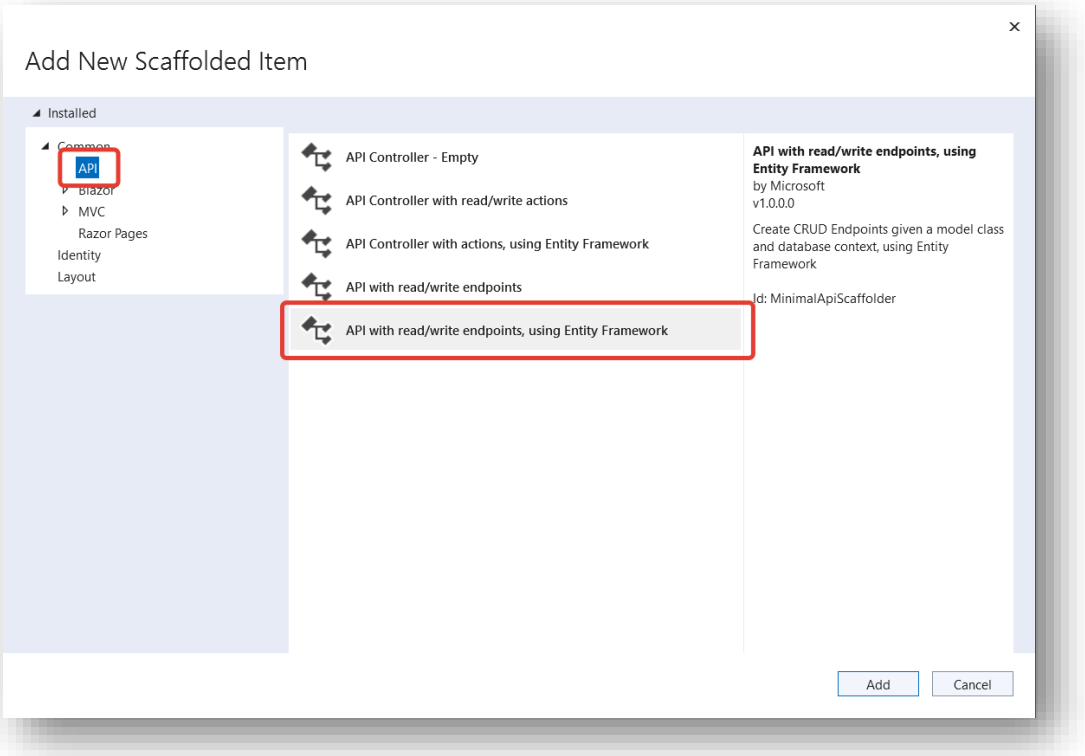


Откройте класс созданного контроллера. Ознакомьтесь с содержимым. Обратите внимание на регистрацию маршрутов конечных точек с помощью атрибутов.

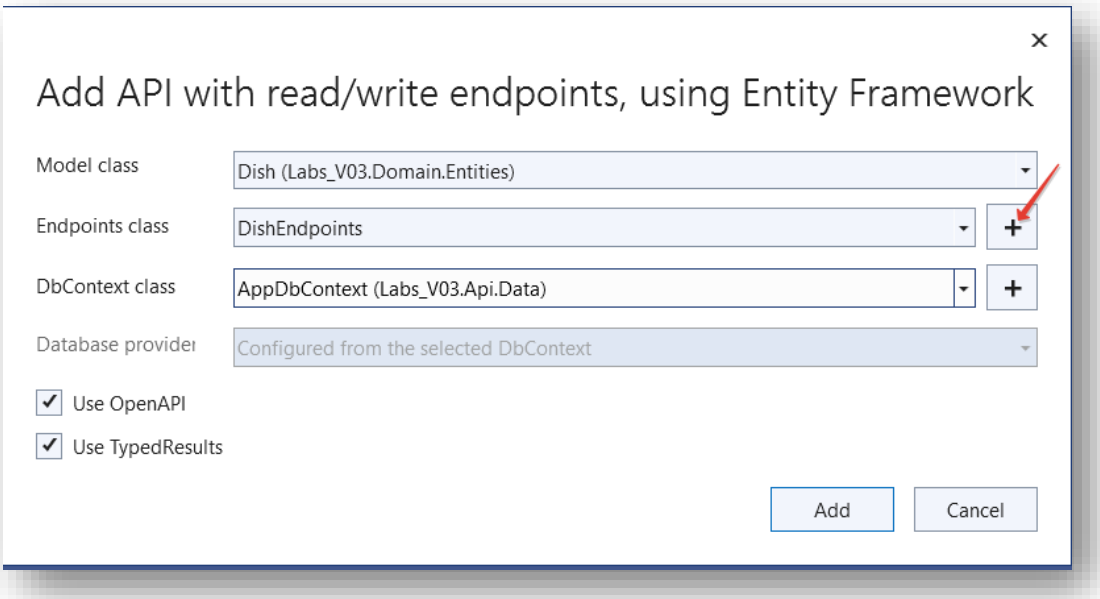
В классе Program найдите регистрацию сервисов контроллеров (`builder.Services.AddControllers();`) и регистрацию маршрутов к контроллерам (`app.MapControllers();`)

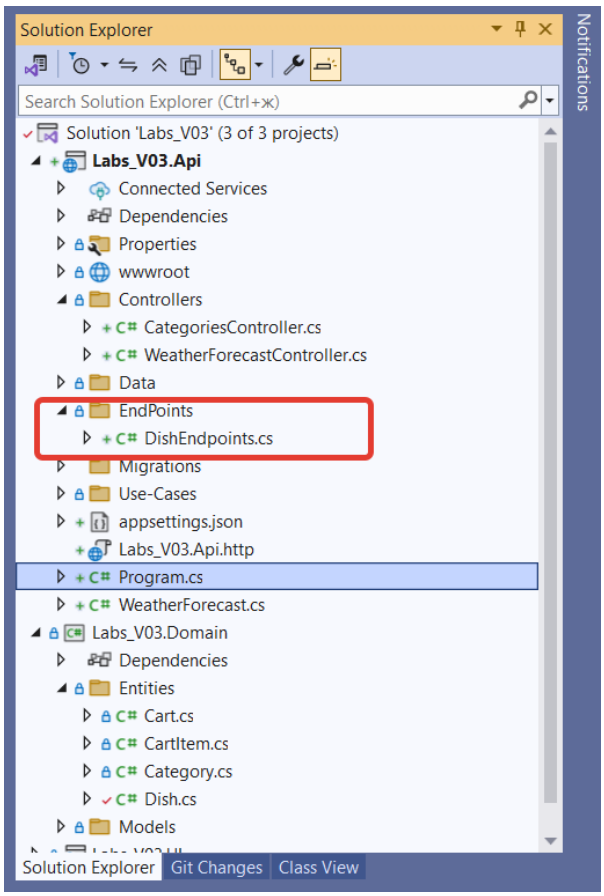
3.8. Создание minimal API

Для API объектов вашей предметной области используем Minimal API. Для этого создайте папку EndPoints. Кликните по созданной папке правой кнопкой мыши и выберите **Add - New scaffolded item – API – API with read/write endpoints, using Entity Framework**



Выберите имя класса, в котором будут описаны конечные точки API:





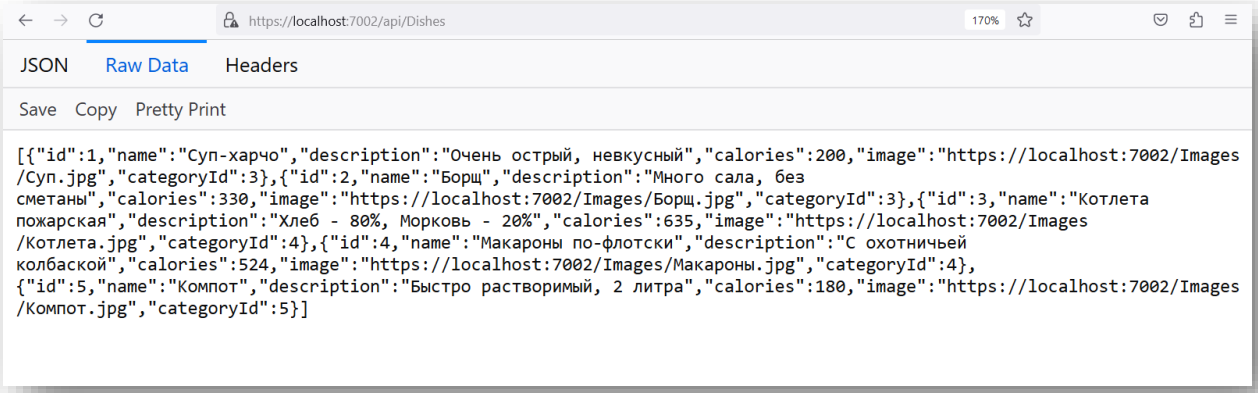
Откройте созданный файл. Ознакомьтесь с тем, как регистрируются конечные точки.

В классе Program найдите регистрацию описанных конечных точек: `app.MapDishEndpoints();`

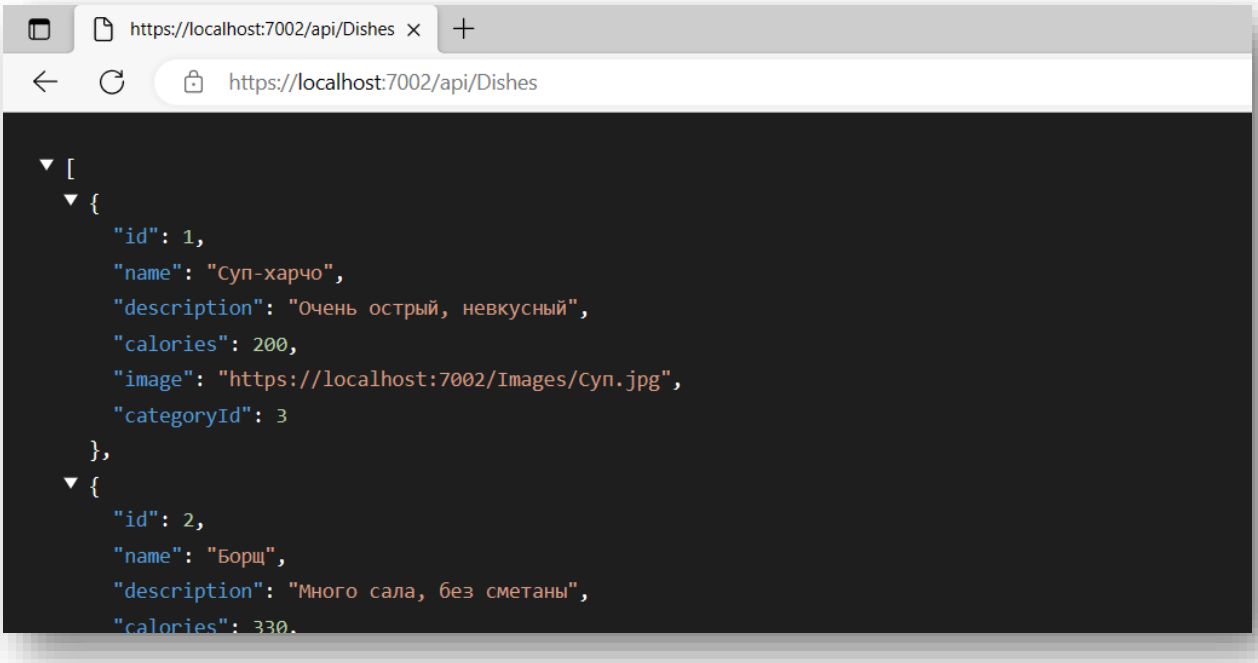
3.9. Проверка API

Запустите проект.

В адресной строке браузера наберите адреса Api объектов и категорий. Убедитесь, что браузер получает данные в формате Json:



Или:



Для проверки функций добавления/удаления/редактирования можно воспользоваться программами Postman, Insomnia и др.

3.10. Доработка кода конечной точки `group.MapGet("/")`

Задание:

Конечная точка `group.MapGet("/")` должна возвращать объекты класса **ResponseData** (см. п.3.2. лабораторной работы №3). Предусмотреть фильтрацию по категории и разбиение на страницы. Номер страницы и размер

страницы могут передаваться **в строке запроса**. Имя категории может передаваться как **сегмент маршрута**.

> Предусмотрите ограничение на размер страницы:
>
> // максимальный размер страницы
>
> private readonly int _maxPageSize = 20;

Выполнение:

Поскольку конечная точка, как и метод контроллера, не должны содержать много кода и напрямую взаимодействовать с хранилищем данных, воспользуйтесь библиотекой MediatR.

Добавьте в проект NuGet пакет MediatR, зарегистрируйте сервис MediatR в классе Program.

Добавьте в проект папку Use-Cases. В созданную папку добавьте файл GetListOfProducts.cs. В созданном файле опишите запрос и его обработчик:

```
public sealed record GetListOfProducts(
        string? categoryNormalizedName,
        int pageNo=1,
        int pageSize=3) : IRequest<ResponseData<ListModel<Dish>>>;

public class GetListOfProductsHandler(AppDbContext db) :
IRequestHandler<GetListOfProducts, ResponseData<ListModel<Dish>>>
{
         private readonly int _maxPageSize = 20;

     . . .

}
```

В качестве примера реализации можно использовать MemoryProductService проекта XXX.UI.

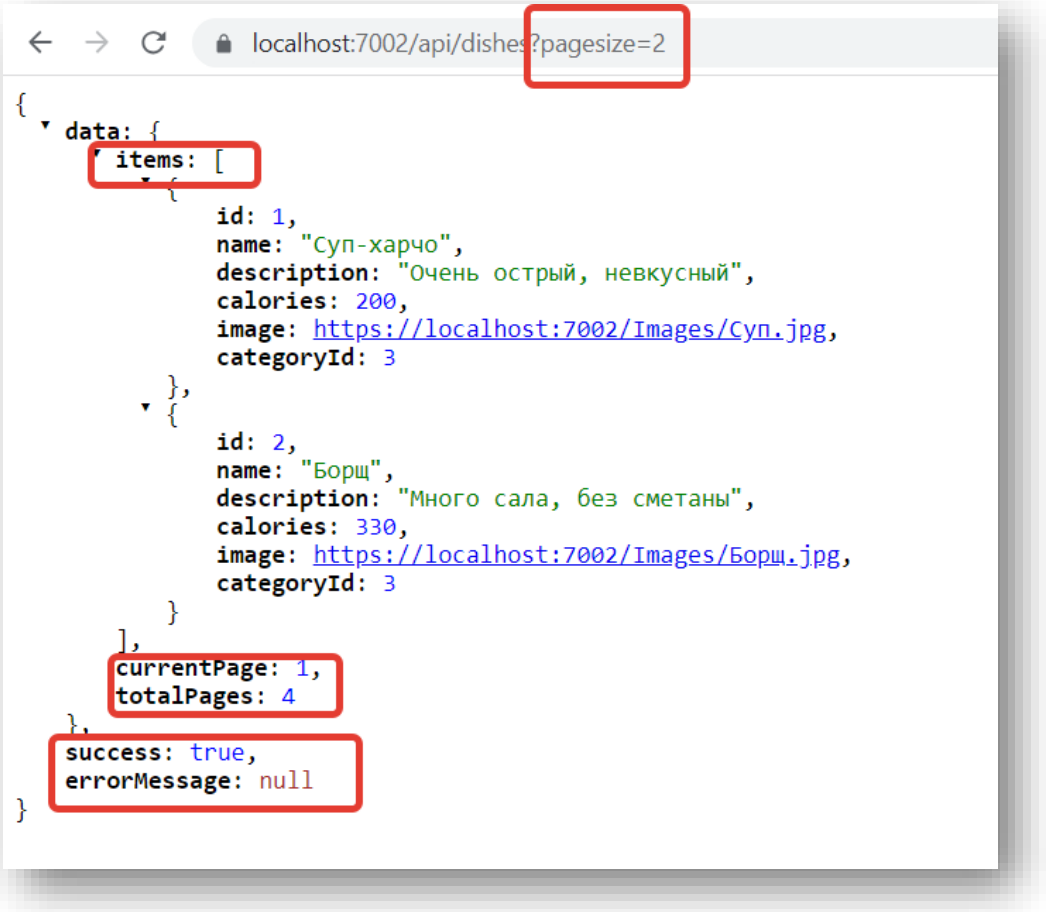
Измените описание конечной точки:

```
group.MapGet("/{category:alpha?}",
           async (IMediator mediator, string? category, int pageNo=1) =>
{
    var data =
           await mediator.Send(new GetListOfProducts(category, pageNo));
    return TypedResults.Ok(data);
})
.WithName("GetAllDishes")
.WithOpenApi();
```

3.11. Проверка API

Запустите проект.

В адресной строке браузера наберите адреса Api объектов и категорий. Убедитесь, что браузер получает данные в формате Json:



3.12. Получение данных с API в проекте XXX.UI

3.12.1. Подготовка проекта

Для обращения к Api понадобится базовый адрес Api-сервиса. Для хранения этой информации опишите **в проекте XXX.UI** класс UriData:

```
public class UriData
{
	public string ApiUri { get; set; }=string.Empty;
}
```

ApiUri – адрес Api – сервиса;

В файле appsettings.json добавьте раздел с адресами сервисов:

```
"UriData": {
    "ApiUri": "https://localhost:7002/api/"
}
```

В классе Program получите **UriData** из объекта IConfiguration.

3.12.2. Создание сервиса для запросов к Api

В проекте XXX.UI опишите класс ApiCategoryService и ApiProductService, реализующие интерфейсы ICategoryService и IProductService посредством взаимодействия с API.

Для отправки запросов понадобится объект IHttpClient. Зарегистрируйте в классе program клиента для IProductService и ICategoryService, например:

```
builder.Services.AddHttpClient<IProductService, ApiProductService>(opt
          => opt.BaseAddress = new Uri(UriData.ApiUri+"dish"));
builder.Services.AddHttpClient<ICategoryService, ApiCategoryService>(opt
          => opt.BaseAddress = new Uri(UriData.ApiUri+"categories"));
```

Пример конструктора:

```
public ApiProductService(HttpClient httpClient,
                         IConfiguration configuration,
                         ILogger<ApiProductService> logger)
{
       _httpClient = httpClient;
       _pageSize = configuration.GetSection("ItemsPerPage").Value;
       _serializerOptions = new JsonSerializerOptions()
            {
                PropertyNamingPolicy = JsonNamingPolicy.CamelCase
            };
       _logger = logger;
}
```

Пример чтения данных:

```
public async Task<ResponseData<ListModel<Dish>>> GetProductListAsync(
                                 string? categoryNormalizedName,
                                 int pageNo = 1)
{
     // подготовка URL запроса
     var urlString
          = new
     StringBuilder($"{_httpClient.BaseAddress.AbsoluteUri}");
     // добавить категорию в маршрут
     if (categoryNormalizedName != null)
     {
          urlString.Append($"{categoryNormalizedName}/");
     };
     // добавить номер страницы в маршрут
```

```
        if (pageNo > 1)
        {
            urlString.Append($"page{pageNo}");
        };
        // добавить размер страницы в строку запроса
        if (!_pageSize.Equals("3"))
        {
            urlString.Append(QueryString.Create("pageSize", _pageSize));
        }

        // отправить запрос к API
        var response = await _httpClient.GetAsync(
                                        new Uri(urlString.ToString()));

        if(response.IsSuccessStatusCode)
        {
            try
            {
                return await response
                    .Content
                    .ReadFromJsonAsync<ResponseData<ListModel<Dish>>>
                                                (_serializerOptions);
            }
            catch(JsonException ex)
            {
                _logger.LogError($"-----> Ошибка: {ex.Message}");

                return ResponseData<ListModel<Dish>>
                            .Error($"Ошибка: {ex.Message}")
            }
        }
        _logger.LogError($"-----> Данные не получены от сервера. Error:
{response.StatusCode.ToString()}");
        return ResponseData<ListModel<Dish>>
            .Error($"Данные не получены от сервера. Error:
        {response.StatusCode.ToString()}")
}
```

Пример записи данных:

```
public async Task<ResponseData<Dish>> CreateProductAsync(Dish product)
{
    product.Image = "Images/noimage.jpg";
    var request = new HttpRequestMessage
    {
        Method = HttpMethod.Post,
        RequestUri = _httpClient.BaseAddress)
    };
    request.Content =
            new StringContent(JsonSerializer.Serialize(product));
    var response = await _httpClient.SendAsync(
                            request,
                            CancellationToken.None);
    if (response.IsSuccessStatusCode)
    {
```

```
        var responseData = await response
            .Content
            .ReadFromJsonAsync<ResponseData<Dish>> (_serializerOptions);
        return responseData;
    }

    _logger.LogError($"-----> object not created. Error:
                        {response.StatusCode.ToString()}");
    return ResponseData<Dish>
                            .Error($"Объект не добавлен. Error:
                                 {response.StatusCode.ToString()}");
}
```

Запустите проект. Убедитесь, что данные на странице «Каталог» отображаются правильно.

4. Контрольные вопросы

1. Чем контроллер API отличается от обычного контроллера?
2. Как осуществляется выбор метода (Action) контроллера API при обработке запроса Http?
3. Где в запросе и в каком виде передаются данные в контроллер API?
4. Как в коде получить Scoped сервис?
5. Что могут возвращать методы контроллера API?
6. Что такое Minimal API?
7. Как зарегистрировать конечную точку для Minimal API?
8. Как зарегистрировать группу конечных точек для Minimal API?