REST

Dockerfile

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build-env

WORKDIR /app

# Copy csproj and restore as distinct layers

COPY \*.csproj ./

RUN dotnet restore

# Copy everything else and build

COPY . ./

RUN dotnet publish -c Release -o out

# Build runtime image

FROM mcr.microsoft.com/dotnet/aspnet:8.0

WORKDIR /app

COPY --from=build-env /app/out .

EXPOSE 80

ENTRYPOINT ["dotnet", "RestApiWithDb.dll"]

Dockerfile описывает процесс создания и сборки образа Docker, который содержит необходимую среду для выполнения приложения. Данный Dockerfile предназначен для построения и запуска .NET приложения. Давайте разберём шаги этого Dockerfile:

**1. FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build-env**

* FROM указывает базовый образ Docker. В данном случае используется образ .NET SDK версии 8.0.
* AS build-env задаёт имя этапа сборки. Позволяет работать с промежуточным этапом для хранения временных файлов, которые не попадают в конечный образ.

**2. WORKDIR /app**

* Устанавливает рабочую директорию в контейнере для выполнения команд.

**3. COPY \*.csproj ./**

* Копирует файлы с расширением .csproj (файлы конфигурации проектов) в текущую директорию контейнера.

**4. RUN dotnet restore**

* Выполняет восстановление NuGet пакетов из файлов конфигурации, чтобы установить все необходимые зависимости для сборки приложения.

**5. COPY . ./**

* Копирует всё содержимое текущей директории в контейнер.

**6. RUN dotnet publish -c Release -o out**

* Компилирует приложение с использованием режима "Release" и сохраняет его в директории out.

**7. FROM mcr.microsoft.com/dotnet/aspnet:8.0**

* Этот этап создает образ с базовым контейнером ASP.NET версии 8.0 для запуска приложения.

**8. WORKDIR /app**

* Устанавливает рабочую директорию в контейнере для выполнения команд.

**9. COPY --from=build-env /app/out .**

* Копирует с предыдущего этапа сборки (из build-env) выходные данные (out) в текущую директорию.

**10. EXPOSE 80**

* Определяет порт, на котором контейнер будет слушать входящие подключения (по умолчанию HTTP - порт 80).

**11. ENTRYPOINT ["dotnet", "RestApiWithDb.dll"]**

* Устанавливает команду запуска контейнера, в данном случае запуск приложения RestApiWithDb.dll с использованием dotnet.

Таким образом, данный Dockerfile создаёт два этапа:

1. Сборка проекта .NET и подготовка файлов для публикации.
2. Запуск приложения внутри контейнера на базе ASP.NET.

Program.cs

using Microsoft.EntityFrameworkCore;

using RestApiWithDb.Data;

using Npgsql.EntityFrameworkCore.PostgreSQL;

using RestApiWithDb.Services;

using SoapCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

builder.Services.AddScoped<IOrderService, OrderService>();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

builder.WebHost.UseUrls("http://\*:8080"); //1

// Configure PostgreSQL connection

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseNpgsql(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddCors(options =>

{

options.AddPolicy("AllowAll",

builder => builder

.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader());

});

builder.Services.AddSoapCore();

var app = builder.Build();

using (var scope = app.Services.CreateScope())

{

var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

dbContext.Database.Migrate();

}

app.UseCors("AllowAll");

app.UseSoapEndpoint<IOrderService>("/Service.asmx",

new SoapEncoderOptions());

app.UseSwagger();

app.UseSwaggerUI();

app.UseAuthorization();

app.MapControllers();

app.Run();

Этот код служит для создания и настройки веб-приложения с поддержкой SOAP и REST API с использованием ASP.NET Core. Давайте разберем каждый блок кода по порядку:

**1. using Microsoft.EntityFrameworkCore;**

* Импорты необходимых библиотек для работы с базой данных и веб-службами.

**2. var builder = WebApplication.CreateBuilder(args);**

* Создается объект WebApplicationBuilder — основной компонент для конфигурации и сборки веб-приложения.

**3. builder.Services.AddControllers();**

* Добавляет поддержку MVC контроллеров в приложение.

**4. builder.Services.AddScoped<IOrderService, OrderService>();**

* Регистрирует сервис IOrderService с конкретной реализацией OrderService в контейнере зависимостей с жизненным циклом Scoped.

**5. builder.Services.AddEndpointsApiExplorer();**

* Добавляет поддержку API-эксплорера для удобного отображения документации по API.

**6. builder.Services.AddSwaggerGen();**

* Добавляет поддержку генерации документации для Swagger.

**7. builder.WebHost.UseUrls("http://\*:8080");**

* Устанавливает возможность для приложения слушать входящие запросы на порту 8080 со всех адресов.

**8. Конфигурация PostgreSQL**

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseNpgsql(builder.Configuration.GetConnectionString("DefaultConnection")));

* Устанавливается подключение к базе данных PostgreSQL. Строка подключения берется из конфигурации приложения.

**9. CORS конфигурация**

builder.Services.AddCors(options =>

{

options.AddPolicy("AllowAll",

builder => builder

.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader());

});

* Добавляется CORS-политика, которая разрешает любые запросы с любого домена (AllowAll).

**10. builder.Services.AddSoapCore();**

* Добавляет поддержку SOAP (Web-сервисы) для приложения.

**11. Создание приложения**

var app = builder.Build();

* Создает веб-приложение на основе конфигурации, определенной в builder.

**12. Миграция базы данных**

using (var scope = app.Services.CreateScope())

{

var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

dbContext.Database.Migrate();

}

* Создает или применяет миграции к базе данных AppDbContext. Это необходимо для создания базы данных или обновления структуры базы данных до последней версии.

**13. Настройка CORS**

app.UseCors("AllowAll");

* Включает CORS-политику для разрешения междоменных запросов.

**14. SOAP-endpoint**

app.UseSoapEndpoint<IOrderService>("/Service.asmx", new SoapEncoderOptions());

* Настраивается точка SOAP-услуги по пути "/Service.asmx".

**15. Swagger**

app.UseSwagger();

app.UseSwaggerUI();

* Включает Swagger для отображения документации API.

**16. Авторизация**

app.UseAuthorization();

* Включает поддержку аутентификации и авторизации.

**17. Роутинг контроллеров**

app.MapControllers();

* Настраивается маршрутизация контроллеров для обработки REST-запросов.

**18. Запуск приложения**

app.Run();

* Запускает веб-приложение.

Таким образом, данный код создаёт веб-приложение с поддержкой SOAP, REST API, а также управляет взаимодействием с базой данных PostgreSQL.

docker-compose.yml

version: '3.8'

services:

api:

build:

context: .

dockerfile: Dockerfile

ports:

- "8080:8080"

depends\_on:

db:

condition: service\_healthy

environment:

- ConnectionStrings\_\_DefaultConnection=Host=db;Port=5432;Database=restapi\_db;Username=postgres;Password=postgres

networks:

- app-network

db:

image: postgres:15

environment:

POSTGRES\_USER: postgres

POSTGRES\_PASSWORD: postgres

POSTGRES\_DB: restapi\_db

ports:

- "5432:5432"

healthcheck:

test: ["CMD-SHELL", "pg\_isready -U postgres"]

interval: 5s

timeout: 10s

retries: 5

networks:

- app-network

networks:

app-network:

driver: bridge

docker-compose.yml — это файл для управления многоконтейнерными приложениями с использованием Docker Compose. Он позволяет упрощенно управлять зависимостями между сервисами, установкой образов Docker и настройкой сети между ними. Давайте разберем каждый компонент данного файла:

**Основные блоки в docker-compose.yml**

**1. version: '3.8'**

* Определяет версию спецификации Docker Compose, в данном случае используется версия 3.8.

**2. services:**

* Определяет конфигурацию для каждого сервиса.

**Сервисы**

**api:**

* **build:**
  + context: . — Указывает, что использовать текущий рабочий каталог как контекст для сборки образа.
  + dockerfile: Dockerfile — Используется Dockerfile для создания образа сервиса API.
* **ports:**
  + - "8080:8080" — Порт 8080 контейнера api доступен для внешнего мира по порту 8080 хоста.
* **depends\_on:**
  + Указывает, что сервис api зависит от сервиса db, и api будет запущен только после успешного состояния службы db (состояние должно быть "healthy").
* **environment:**
  + Устанавливаются переменные окружения для подключения к базе данных:
    - ConnectionStrings\_\_DefaultConnection=Host=db;Port=5432;Database=restapi\_db;Username=postgres;Password=postgres
* **networks:**
  + Связывает сервис api с сетью app-network.

**db:**

* **image: postgres:15**
  + Используется официальный образ PostgreSQL версии 15.
* **environment:**
  + Устанавливаются переменные окружения для базы данных PostgreSQL:
    - POSTGRES\_USER=postgres
    - POSTGRES\_PASSWORD=postgres
    - POSTGRES\_DB=restapi\_db
* **ports:**
  + - "5432:5432" — Порт базы данных PostgreSQL открыт для доступа на порту 5432 хоста.
* **healthcheck:**
  + Проверяет здоровье контейнера каждые 5 секунд с помощью команды pg\_isready -U postgres.
  + Если команда не выполнится за 10 секунд или не пройдет 5 раз подряд, контейнер будет считаться нездоровым.
* **networks:**
  + Связывает сервис db с сетью app-network.

**3. networks:**

* Определяет пользовательскую сеть app-network, которая соединяет сервисы вместе.
  + driver: bridge — Используется сетевой драйвер "bridge".

Order.cs

namespace RestApiWithDb.Models

{

public class Order

{

public int Id { get; set; }

public string ProductName { get; set; }

public int Quantity { get; set; }

public string Status { get; set; }

}

}

Класс Order представляет собой модель для хранения информации о заказах в REST API приложении. Давайте разберем его структуру:

**1. public int Id { get; set; }**

* **Id** — это целочисленное свойство, представляющее уникальный идентификатор заказа.
* public int — это свойство доступно для чтения и записи.

**2. public string ProductName { get; set; }**

* **ProductName** — строковое свойство для хранения названия товара, который заказан.

**3. public int Quantity { get; set; }**

* **Quantity** — целочисленное свойство для хранения количества товаров в заказе.

**4. public string Status { get; set; }**

* **Status** — строковое свойство для хранения статуса заказа, например, "В обработке", "Отгружен", "Завершен" и т. д.

AppDbContext.cs

using Microsoft.EntityFrameworkCore;

using RestApiWithDb.Models;

namespace RestApiWithDb.Data

{

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

public DbSet<Order> Orders { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

optionsBuilder.UseNpgsql("Host=db;Port=5432;Database=restapi\_db;Username=postgres;Password=postgres");

}

}

}

}

Код описывает контекст базы данных AppDbContext для работы с базой данных в .NET приложении с использованием Entity Framework Core. Давайте разберем каждую часть этого кода:

**1. using Microsoft.EntityFrameworkCore;**

* Импортирует пространство имен Entity Framework Core, которое предоставляет инструменты для работы с базами данных.

**2. using RestApiWithDb.Models;**

* Импортирует модель данных Order, которая будет использоваться для работы с таблицей заказов в базе данных.

**3. public class AppDbContext : DbContext**

* AppDbContext — это класс, представляющий контекст базы данных. Наследуется от DbContext, который является основной базой для работы с базой данных в Entity Framework Core.

**4. Конструктор**

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

* Конструктор принимает DbContextOptions<AppDbContext> — настройки для конфигурации контекста. Вызов базового конструктора base(options) позволяет передать настройки базового класса DbContext.

**5. Свойства**

public DbSet<Order> Orders { get; set; }

* Свойство Orders типа DbSet<Order> предоставляет доступ к коллекции заказов в базе данных. Это позволяет добавлять, обновлять и удалять записи заказов.

**6. Метод OnConfiguring**

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

optionsBuilder.UseNpgsql("Host=db;Port=5432;Database=restapi\_db;Username=postgres;Password=postgres");

}

}

* Метод OnConfiguring используется для настройки конфигурации при первом обращении к контексту.
* optionsBuilder.IsConfigured проверяет, была ли уже задана конфигурация.
* Если конфигурация не была задана, то в методе UseNpgsql устанавливается подключение к базе данных PostgreSQL с соответствующими параметрами: хост (Host=db), порт (Port=5432), имя базы данных (Database=restapi\_db), имя пользователя (Username=postgres) и пароль (Password=postgres).

**Общий смысл**

Этот класс AppDbContext предоставляет доступ к модели данных Order и создает соединение с базой данных PostgreSQL. Все взаимодействие с базой данных будет происходить через этот контекст.

ValuesController.cs

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using RestApiWithDb.Data;

using RestApiWithDb.Models;

namespace RestApiWithDb.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class OrderController : ControllerBase

{

private readonly AppDbContext \_context;

public OrderController(AppDbContext context)

{

\_context = context;

}

[HttpGet]

public async Task<IActionResult> GetOrders()

{

return Ok(await \_context.Orders.ToListAsync());

}

[HttpGet("{id}")]

public async Task<IActionResult> GetOrder(int id)

{

var order = await \_context.Orders.FindAsync(id);

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

return Ok(order);

}

[HttpPost]

public async Task<IActionResult> CreateOrder([FromBody] Order order)

{

\_context.Orders.Add(order);

await \_context.SaveChangesAsync();

return CreatedAtAction(nameof(GetOrder), new { id = order.Id }, order);

}

[HttpPut("{id}")]

public async Task<IActionResult> UpdateOrder(int id, [FromBody] Order order)

{

if (id != order.Id) return BadRequest();

\_context.Entry(order).State = EntityState.Modified;

await \_context.SaveChangesAsync();

return NoContent();

}

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteOrder(int id)

{

var order = await \_context.Orders.FindAsync(id);

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

\_context.Orders.Remove(order);

await \_context.SaveChangesAsync();

return NoContent();

}

}

}

Код представляет API контроллер для работы с заказами в приложении REST API с использованием ASP.NET Core. Давайте разберем каждую часть этого кода:

**1. using директивы**

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using RestApiWithDb.Data;

using RestApiWithDb.Models;

* Microsoft.AspNetCore.Mvc — пространство имен, предоставляющее инструменты для создания и обработки запросов API и создания ответов.
* Microsoft.EntityFrameworkCore — пространство имен для работы с Entity Framework Core.
* RestApiWithDb.Data — пространство имен с контекстом базы данных.
* RestApiWithDb.Models — пространство имен с моделями данных.

**2. Контроллер OrderController**

[Route("api/[controller]")]

[ApiController]

public class OrderController : ControllerBase

{

private readonly AppDbContext \_context;

public OrderController(AppDbContext context)

{

\_context = context;

}

}

* Контроллер отвечает за обработку запросов для работы с заказами.
* Route("api/[controller]") — базовый маршрут для всех эндпоинтов контроллера. В данном случае маршрут будет api/order.
* [ApiController] — атрибут, который включает полезные функции для работы с API, такие как автоматическое возвращение форматов ошибок в случае неудачных операций.

**3. Методы контроллера**

**Метод GetOrders**

[HttpGet]

public async Task<IActionResult> GetOrders()

{

return Ok(await \_context.Orders.ToListAsync());

}

* **[HttpGet]** — метод GET, который возвращает список всех заказов из базы данных.
* ToListAsync() — асинхронное получение списка заказов из базы данных.
* Ok(await \_context.Orders.ToListAsync()) — возвращает статус 200 OK с результатом в формате списка заказов.

**Метод GetOrder**

[HttpGet("{id}")]

public async Task<IActionResult> GetOrder(int id)

{

var order = await \_context.Orders.FindAsync(id);

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

return Ok(order);

}

* **[HttpGet("{id}")]** — метод GET с параметром id для получения заказа по его идентификатору.
* FindAsync(id) — метод для нахождения записи заказа по идентификатору.
* Если заказ не найден, возвращается NotFound(). В противном случае возвращается Ok(order).

**Метод CreateOrder**

[HttpPost]

public async Task<IActionResult> CreateOrder([FromBody] Order order)

{

\_context.Orders.Add(order);

await \_context.SaveChangesAsync();

return CreatedAtAction(nameof(GetOrder), new { id = order.Id }, order);

}

* **[HttpPost]** — метод POST для создания нового заказа.
* FromBody — используемое атрибутом, чтобы получить данные из тела запроса.
* Add(order) — добавляет новый заказ в контекст базы данных.
* SaveChangesAsync() — сохраняет изменения в базе данных.
* CreatedAtAction — возвращает результат с кодом состояния 201 Created и с адресом созданного ресурса, передавая id нового заказа.

**Метод UpdateOrder**

[HttpPut("{id}")]

public async Task<IActionResult> UpdateOrder(int id, [FromBody] Order order)

{

if (id != order.Id) return BadRequest();

\_context.Entry(order).State = EntityState.Modified;

await \_context.SaveChangesAsync();

return NoContent();

}

* **[HttpPut("{id}")]** — метод PUT для обновления существующего заказа.
* Проверяется, что id заказа из пути совпадает с id в переданных данных (order). Если нет, возвращается BadRequest().
* EntityState.Modified устанавливает состояние записи в "обновленное".
* SaveChangesAsync() сохраняет изменения в базе данных.
* NoContent() — возвращает пустой ответ с кодом состояния 204 No Content.

**Метод DeleteOrder**

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteOrder(int id)

{

var order = await \_context.Orders.FindAsync(id);

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

\_context.Orders.Remove(order);

await \_context.SaveChangesAsync();

return NoContent();

}

* **[HttpDelete("{id}")]** — метод DELETE для удаления заказа по идентификатору.
* FindAsync(id) ищет заказ по идентификатору.
* Если заказ не найден, возвращается NotFound().
* Remove(order) удаляет заказ из базы данных.
* SaveChangesAsync() сохраняет изменения.
* NoContent() — возвращает пустой ответ с кодом состояния 204 No Content.

appsettings.json

{

"ConnectionStrings": {

"DefaultConnection": "Host=localhost;Port=5432;Database=restapi\_db;Username=postgres;Password=postgres"

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*"

}

Этот JSON-конфигурационный файл используется для настройки приложения ASP.NET Core. Давайте рассмотрим каждую секцию более подробно:

**1. ConnectionStrings**

"ConnectionStrings": {

"DefaultConnection": "Host=localhost;Port=5432;Database=restapi\_db;Username=postgres;Password=postgres"

}

* ConnectionStrings — содержит строку подключения к базе данных.
* DefaultConnection — имя строки подключения.
* Значение строки подключения указывает на сервер (Host=localhost), порт (Port=5432), название базы данных (Database=restapi\_db), а также имя пользователя и пароль (Username=postgres;Password=postgres).

**2. Logging**

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

}

* Logging — настройки логирования.
* LogLevel — уровень логирования.
  + Default: Уровень логов для общих событий (по умолчанию "Information").
  + Microsoft.AspNetCore: Уровень логов для ASP.NET Core событий ("Warning").

**3. AllowedHosts**

"AllowedHosts": "\*"

* AllowedHosts — указывает, какие хосты разрешены для приложения.
  + В данном случае указано "\*" — это означает, что все хосты разрешены (неограниченно).

SOAP

IOrderService.cs

using System.ServiceModel;

using System.Threading.Tasks;

using System.Collections.Generic;

using RestApiWithDb.Models;

namespace RestApiWithDb.Services

{

[ServiceContract]

public interface IOrderService

{

[OperationContract]

Task<List<Order>> GetOrders(int pageNumber, int pageSize);

[OperationContract]

Task<Order> GetOrder(int id);

[OperationContract]

Task<Order> CreateOrder(Order order);

[OperationContract]

Task<Order> UpdateOrder(int id, Order order);

[OperationContract]

Task<bool> DeleteOrder(int id);

}

}

Этот код определяет **SOAP-сервис** для работы с заказами в системе. Интерфейс IOrderService описывает контракты (доступные методы) для взаимодействия с сервисом. Давайте разберем код:

**Ключевые элементы**

1. **[ServiceContract]**
   * Атрибут указывает, что интерфейс IOrderService является контрактом для SOAP-сервиса.
   * Все методы, помеченные атрибутом [OperationContract], будут доступны через SOAP.
2. **Методы сервиса**

**GetOrders**

[OperationContract]

Task<List<Order>> GetOrders(int pageNumber, int pageSize);

* + Возвращает список заказов с учетом пагинации (страниц).
  + Аргументы:
    - pageNumber — номер страницы.
    - pageSize — количество заказов на странице.

**GetOrder**

[OperationContract]

Task<Order> GetOrder(int id);

* + Возвращает заказ по его идентификатору.
  + Аргументы:
    - id — идентификатор заказа.

**CreateOrder**

[OperationContract]

Task<Order> CreateOrder(Order order);

* + Создает новый заказ.
  + Аргументы:
    - order — объект заказа (содержит данные для создания).

**UpdateOrder**

[OperationContract]

Task<Order> UpdateOrder(int id, Order order);

* + Обновляет существующий заказ.
  + Аргументы:
    - id — идентификатор заказа.
    - order — объект с обновленными данными.

**DeleteOrder**

[OperationContract]

Task<bool> DeleteOrder(int id);

* + Удаляет заказ по идентификатору.
  + Аргументы:
    - id — идентификатор заказа.
  + Возвращает:
    - true, если удаление успешно, иначе false.

OrderService.cs

using System.Collections.Generic;

using System.Linq;

using System.ServiceModel;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using RestApiWithDb.Data;

using RestApiWithDb.Models;

namespace RestApiWithDb.Services

{

public class OrderService : IOrderService

{

private readonly AppDbContext \_context;

public OrderService(AppDbContext context)

{

\_context = context;

}

public async Task<List<Order>> GetOrders(int pageNumber, int pageSize)

{

return await \_context.Orders

.Skip((pageNumber - 1) \* pageSize)

.Take(pageSize)

.ToListAsync();

}

public async Task<Order> GetOrder(int id)

{

return await \_context.Orders.FindAsync(id);

}

public async Task<Order> CreateOrder(Order order)

{

\_context.Orders.Add(order);

await \_context.SaveChangesAsync();

return order;

}

public async Task<Order> UpdateOrder(int id, Order order)

{

if (id != order.Id)

throw new FaultException("ID mismatch");

var existingOrder = await \_context.Orders.FindAsync(id);

if (existingOrder == null)

throw new FaultException("Order not found");

\_context.Entry(order).State = EntityState.Modified;

await \_context.SaveChangesAsync();

return order;

}

public async Task<bool> DeleteOrder(int id)

{

var order = await \_context.Orders.FindAsync(id);

if (order == null)

throw new FaultException("Order not found");

\_context.Orders.Remove(order);

await \_context.SaveChangesAsync();

return true;

}

}

}

Этот код реализует интерфейс IOrderService для обработки операций с заказами в базе данных. Разберем основные элементы:

**Класс OrderService**

Класс предоставляет логику работы с сущностями заказов (Order) через Entity Framework Core. Все методы взаимодействуют с базой данных, используя объект AppDbContext.

**Разбор методов**

1. **GetOrders**

public async Task<List<Order>> GetOrders(int pageNumber, int pageSize)

{

return await \_context.Orders

.Skip((pageNumber - 1) \* pageSize)

.Take(pageSize)

.ToListAsync();

}

* + Возвращает список заказов с учетом **пагинации**.
  + **Skip** пропускает записи, относящиеся к предыдущим страницам.
  + **Take** ограничивает количество записей на странице.

1. **GetOrder**

public async Task<Order> GetOrder(int id)

{

return await \_context.Orders.FindAsync(id);

}

* + Находит заказ по идентификатору (id).
  + Используется метод **FindAsync**, который выполняет поиск записи по ключу в таблице Orders.

1. **CreateOrder**

public async Task<Order> CreateOrder(Order order)

{

\_context.Orders.Add(order);

await \_context.SaveChangesAsync();

return order;

}

* + Создает новый заказ:
    - **Add** добавляет запись в контекст данных.
    - **SaveChangesAsync** сохраняет изменения в базе данных.
  + Возвращает созданный объект заказа с обновленным идентификатором (Id), установленным базой данных.

1. **UpdateOrder**

public async Task<Order> UpdateOrder(int id, Order order)

{

if (id != order.Id)

throw new FaultException("ID mismatch");

var existingOrder = await \_context.Orders.FindAsync(id);

if (existingOrder == null)

throw new FaultException("Order not found");

\_context.Entry(order).State = EntityState.Modified;

await \_context.SaveChangesAsync();

return order;

}

* + **Обновление заказа:**
    - Проверяется, совпадает ли идентификатор id с идентификатором объекта order.
    - Проверяется существование заказа в базе данных.
    - Метод **Entry** устанавливает состояние сущности как Modified, чтобы обновить запись в базе данных.
  + Исключения (FaultException) бросаются при несоответствии идентификаторов или отсутствии записи.

1. **DeleteOrder**

public async Task<bool> DeleteOrder(int id)

{

var order = await \_context.Orders.FindAsync(id);

if (order == null)

throw new FaultException("Order not found");

\_context.Orders.Remove(order);

await \_context.SaveChangesAsync();

return true;

}

* + **Удаление заказа:**
    - Находит заказ по идентификатору.
    - Удаляет запись с помощью метода **Remove**.
    - Возвращает true, если удаление успешно.

**Особенности**

1. **Паттерн Асинхронности**
   * Все методы используют асинхронные версии Entity Framework Core (FindAsync, ToListAsync, SaveChangesAsync), что позволяет эффективно обрабатывать запросы без блокировки потоков.
2. **SOAP-специфичные Исключения**
   * Вместо стандартных исключений (ArgumentException, InvalidOperationException) используются **FaultException**, которые специфичны для SOAP и передают информацию об ошибке клиенту.
3. **Инкапсуляция Бизнес-Логики**
   * Все операции с базой данных инкапсулированы в сервисе OrderService, что делает его удобным для повторного использования и тестирования.

**Как работает в системе**

* Этот класс подключается в Program.cs:

builder.Services.AddScoped<IOrderService, OrderService>();

* + При каждом запросе создается новый экземпляр OrderService с подключением к базе данных.
* Через SoapCore создается SOAP-эндпоинт, позволяющий клиентам вызывать методы OrderService.