Министерство образования и науки Российской Федерации

Российский государственный университет нефти и газа

(национальный исследовательский университет)

им. И.М. Губкина

Кафедра информатики

Отчет по лабораторной работе №2

дисциплины ***Разработка мобильных и WEB приложений***

«Разработка серверной части приложения»

Выполнил:

студент группы АА-21-07

Сюзев Максим Владиславович

Проверила:

Казакова Анастасия Семеновна

2024 г.

Описание проекта

Это учебное веб-приложение на ASP.NET core с использованием EntityFrameworkCore для сохранения/удаления студентов, преподавателей, задача, групп, дисциплин через API. Архитектура используемая в нашем API (clear)

Стек технологий

- C# 10
- ASP.NET core
- EntityFrameworkCore
- Postrgres
- Docker

Основные возможности

- Создание, просмотр, редактирование и удаление.
- REST API для работы с бд
- Статическая подача контента

Теория и структура ASP .Net core проекта

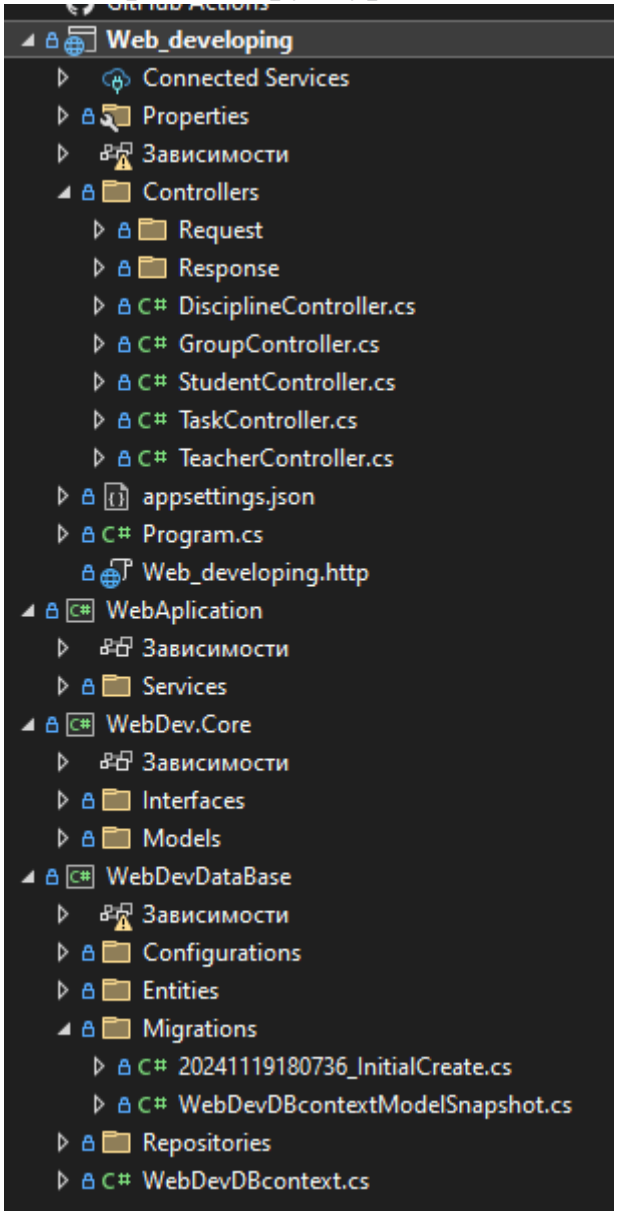
ASP.NET Core — свободно-распространяемый кроссплатформенный фреймворк для создания веб-приложений на платформе .NET с открытым исходным кодом. Данная платформа разрабатывается компанией Майкрософт совместно с сообществом и имеет большую производительность по сравнению с ASP.NET. Имеет модульную структуру и совместима с такими операционными системами как Windows, Linux и macOS.

Основные компоненты ASP .Net **проекта:**

1. Проект — верхний уровень структуры, содержащий настройки и конфигурацию веб-приложения.
2. Приложения (Application) — отдельные сервисы внутри проекта, каждый из которых отвечает за конкретную функциональность и может быть использован в других проектах.

3. Ядро – необходимо для представления моделей с которыми работаешь наше API и представления интерфейсов реализованных в контексте базы данных и сервисах.
4. Контекст базы данных – необходим для работы с базой данных. Так же был использован принцип CodeFirst поэтому на основании этого проекта были созданы миграции и инициализированы таблицы в бд.

Стандартная структура ASP .Net core проекта:



Описание основных файлов и директорий

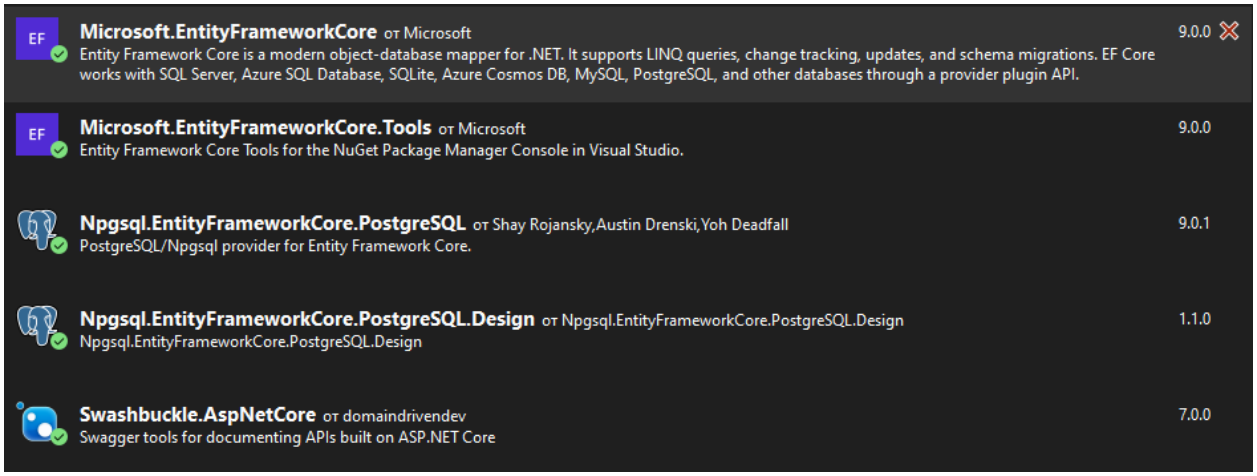
- **Program.cs** — файл необходимый для регистрации наших контроллеров и построения связи между нашими интерфейсами и реализующими их классами. Так же именно в нем builder производит соединение нашего API с используемой бд

- **Контроллеры** — файлы необходимы для создания контроллеров, которые в дальнейшем будем использовать frontend
- **Services** – файлы содержащие в себе всю бизнес логику нашего приложения для корректной работы.
- **Модели** — Классы объекты которого использует наше API для работы.
- **Configurations** — Файлы описывающие конфигурацию каждой таблицы в нашей базе данных
- **Сущности** — классы объекты, которых создаются при загрузке данных из бд. Внутри себя не имеют логики. В отличие от моделей.
- **Repositories**— классы содержащие в себе логику взаимодействия нашего API с базой данных.

Этапы создания

Шаг 1: Настройка проекта ASP .Net core

1.1. Установим NuGet пакеты для работы с базой данных:



1.2. Создадим connectionstring для соединения с базой данных

```
"ConnectionStrings": {
  "WebDevDBcontext": "Username=maxim;Password=maxim;Host=localhost;Port=5432;Database=web_db"
}
```

1.3. Создадим docker compose файл для создания сервера постгрес:

```yaml
services:
  db:
    image: postgres:latest
    container_name: web_cnotainer
    environment:
      POSTGRES_DB: web_db
      POSTGRES_USER: maxim
      POSTGRES_PASSWORD: maxim
    ports:
      - "5432:5432"
    volumes:
      - pg_data:/var/lib/postgresql/data

volumes:
  pg_data:
```

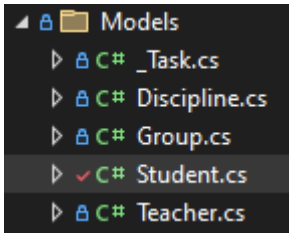
1.4. Запустим наш Docker Compose файл командой docker compose up

```
C:\Users\cuzev>cd C:\Users\cuzev\OneDrive\Рабочий стол\gitrepository\Web_Dev_Rep\Web_Dev_rep\Web_development_db

C:\Users\cuzev\OneDrive\Рабочий стол\gitrepository\Web_Dev_Rep\Web_Dev_rep\Web_development_db>docker compose up
[+] Running 1/0
 ⠿ Container web_cnotainer  Created                                                                    0.0s
Attaching to web_cnotainer
web_cnotainer  |
web_cnotainer  | PostgreSQL Database directory appears to contain a database; Skipping initialization
web_cnotainer  |
web_cnotainer  | 2024-11-19 18:56:49.033 UTC [1] LOG:  starting PostgreSQL 17.0 (Debian 17.0-1.pgdg120+1) on x86_64-pc-l
inux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
web_cnotainer  | 2024-11-19 18:56:49.033 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
web_cnotainer  | 2024-11-19 18:56:49.033 UTC [1] LOG:  listening on IPv6 address "::", port 5432
web_cnotainer  | 2024-11-19 18:56:49.044 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
web_cnotainer  | 2024-11-19 18:56:49.059 UTC [30] LOG:  database system was shut down at 2024-11-19 18:35:27 UTC
web_cnotainer  | 2024-11-19 18:56:49.072 UTC [1] LOG:  database system is ready to accept connections
```

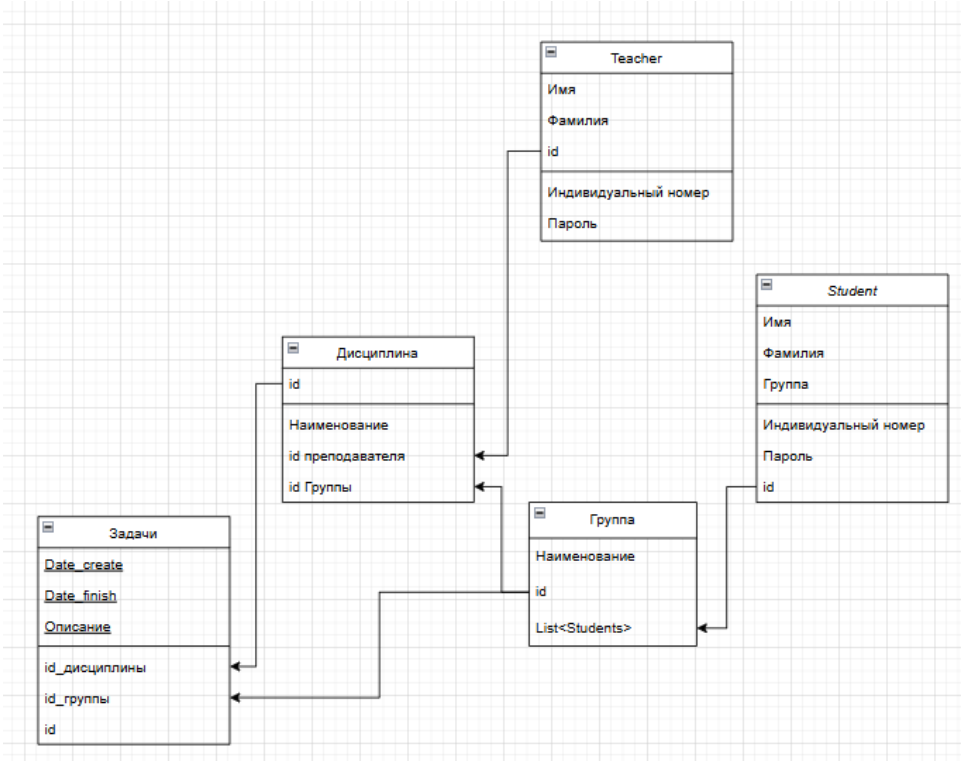
Шаг 2: Создание моделей используемых нашим API

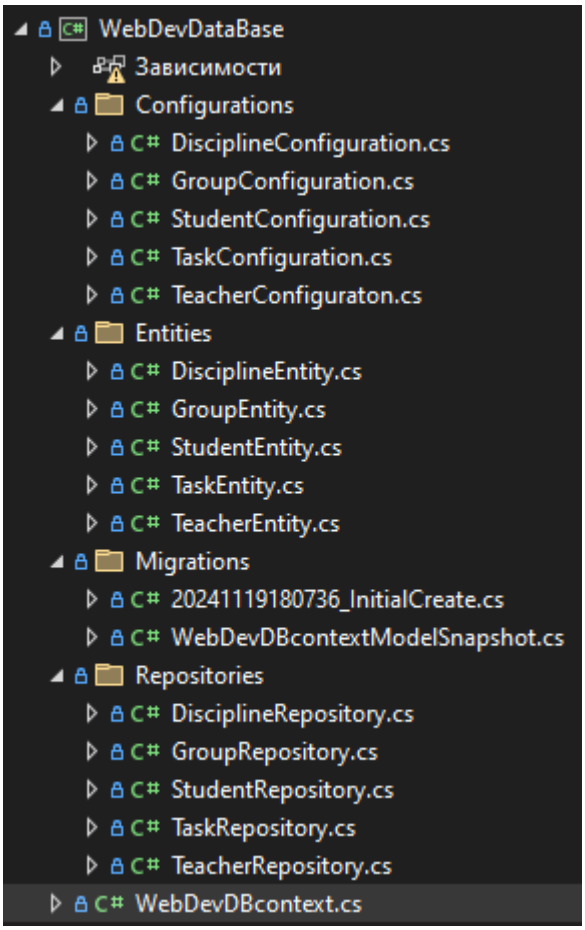
2.1. Создаем класс Student (повторим для всех необходимых моделей)

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WebDev.Core.Models
{
    public class Student
    {
        public Guid id { get; }
        public string Name { get; } = string.Empty;
        public string SecondName { get; } = string.Empty;
        public string GroupName { get; } = string.Empty;
        public ulong IndividualNumber { get; }
        public string Password { get; } = string.Empty;

        private Student(Guid id,string Name
            ,string SecondName,string GroupName
            ,ulong IndividualNumber,string Password)
        {
            this.id = id;
            this.Name = Name;
            this.SecondName = SecondName;
            this.GroupName = GroupName;
            this.IndividualNumber = IndividualNumber;
            this.Password = Password;
        }

        public static (Student student, string Error) CreateStudent (Guid id,string Name
            , string SecondName, string GroupName
            , ulong IndividualNumber, string Password)
        {
            var Error = string.Empty;

            if (string.IsNullOrEmpty(Name))
            {
                Error = "Ошибка введите имя";
            }

            var student = new Student(id, Name, SecondName, GroupName, IndividualNumber, Password);

            return (student, Error);
        }

    }
}
```



2.2. Создадим Контекст нашей базы данных (по разработанной нами архитектуре)

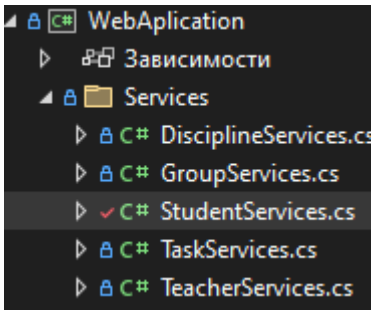





```
using Microsoft.EntityFrameworkCore;
using WebDevDataBase.Entities;

namespace WebDevDataBase
{
    public class WebDevDBcontext:DbContext
    {
        public WebDevDBcontext(DbContextOptions<WebDevDBcontext> options)
            : base(options)
        {
        }
        public DbSet<StudentEntity> Students { get; set; }
        public DbSet<TeacherEntity> Teachers { get; set; }
        public DbSet<TaskEntity> Tasks { get; set; }
        public DbSet<GroupEntity> Groups { get; set; }
        public DbSet<DisciplineEntity> Disciplines { get; set; }
    }
}
```

Шаг 3: Создадим сервисы описывающие бизнес логику нашего API и работающие с нашими моделями

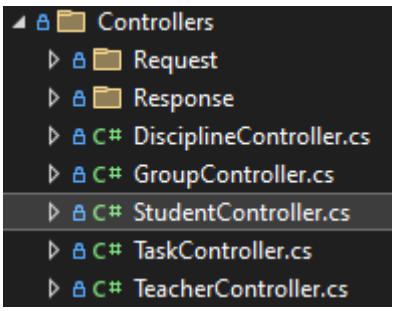
```
using System.Xml.Linq;
using WebDev.Core.Interfaces;
using WebDev.Core.Models;

namespace WebAplication.Services
{
    public class StudentServices : IStudentServices
    {
        private readonly IStudentRepository _StudentRepository;

        public StudentServices(IStudentRepository StudentRepository)
        {
            _StudentRepository = StudentRepository;
        }

        public async Task<List<Student>> GetAllStudents()
        {
            return await _StudentRepository.GetStudents();
        }

        public async Task<Guid> CreateStudent(Student student)
        {
            return await _StudentRepository.CreateStudent(student);
        }

        public async Task<Guid> DeleteStudent(Guid id)
        {
            return await _StudentRepository.DeleteStudent(id);
        }

        public async Task<Guid> UpdateStudent(Guid id, string Name, string SecondName, ulong IndividualNumber, string GroupName, string GroupNumber, string Password)
        {
            return await _StudentRepository.UpdateStudent(id, Name, SecondName, IndividualNumber, GroupName, GroupNumber, Password);
        }
    }
}
```



Шаг 4: создадим контроллеры для взаимодействия с нашим API через swagger

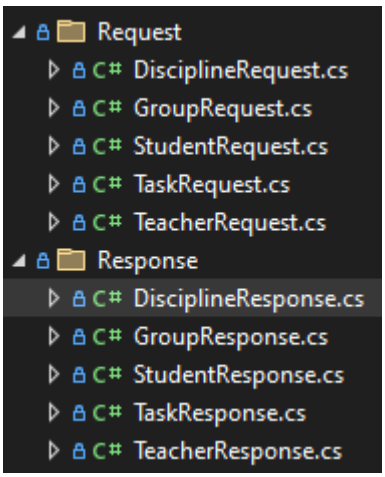
Создание контроллеров необходимо, так как, по сути, это единственная часть программы способная контактировать с внешними приложениями. Контроллер обеспечивает «связь» между пользователем и системой. Контролирует и направляет данные от пользователя к системе и наоборот. Использует модель и представление для реализации необходимого действия.

```
using Microsoft.AspNetCore.Mvc;
using Web_developing.Controllers.Request;
using Web_developing.Controllers.Response;
using WebAplication.Services;
using WebDev.Core.Models;


namespace Web_developing.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class StudentController:ControllerBase
    {
        private readonly IStudentServices _studentService;

        public StudentController(IStudentServices studentService)
        {
            _studentService = studentService;
        }

        [HttpGet]
        public async Task<ActionResult<List<StudentResponse>>> GetStudents()
        {
            var students = await _studentService.GetAllStudents();
            var response = students.Select(b => new StudentResponse(b.id, b.Name, b.SecondName,b.GroupName,b.IndividualNumber, b.Password));
            return Ok(response);
        }

        [HttpPost]
        public async Task<ActionResult<Guid>> CreateStudents([FromBody] StudentRequest request)
        {
            var (student, Error) = Student.CreateStudent(
                Guid.NewGuid(),
                request.Name,
                request.SecondName,
                request.GroupName,
                request.IndividualNumber,
                request.Password);
            if (!string.IsNullOrEmpty(Error))
            {
                return BadRequest(Error);
            }
            await _studentService.CreateStudent(student);
            return Ok(student.id);
```



Шаг 5: Написание Response и Request

Напишем record классы (разница с обычными в том, что record классы неизменны (Immutable)). Они необходимы нам, чтобы мы могли четко понимать какого формата запрос будет приходить нашему API с Frontenda и какой формат ответа будет получать наш FrontEnd.




```
namespace Web_developing.Controllers.Response
{
    public record DisciplineResponse
    (
        Guid id,
        string Name,
        Guid idTeacher,
        Guid idGroup);
}
```