Sprint 1 Introduction

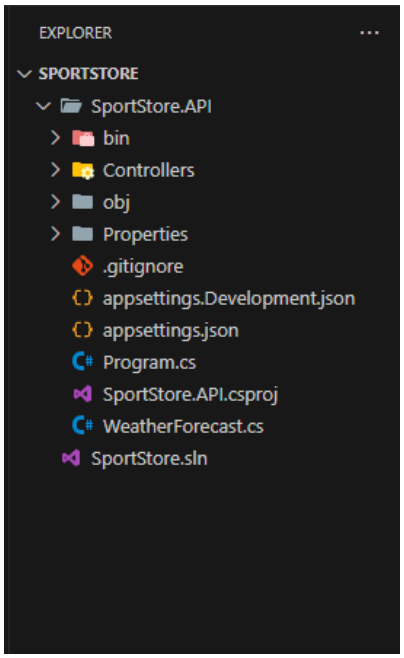
Проекта API. Тип архитектуры: All-In

Реализовать базовую функциональность API

Использование dotnet CLI

- создайте папку SportStore и перейдите в нее.
- посмотрите с помощью команды `dotnet new list` список доступных проектов и создайте проект `webapi` с именем `SportStore.API`
- добавьте файл решения, находясь в папке рабочей директории, командой `dotnet new sln`
- добавьте в решение проект API - `dotnet sln add SportStore.API`
- добавьте файл gitignore (на уровне рабочей директории) - `dotnet new gitignore`
- откройте начальную архитектуру проекта в Visual Code командой `code .`

В результате вид обозревателя должен получиться такой:



Для проверки работоспособности приложения запустите API

```
dotnet run --project SportStore.API .
```

У вас по конечной точке http://localhost:5290/weatherforecast должен выводится результат в формате json.

Примечание: номер порта может быть другим.

- добавьте в решение файл readme.md
- настройка Visual Code(exlude obj and bin, prefix _)
- настройка среды разработки (Visual Code, Visual Studio, Rider)
- фиксация изменений в git в master
- от мастер создать ветку `git branch all-in` и перейти в нее `git checkout all-in` . Далее работа будет вестись в этой ветке.

Разработка домена приложения. Модель пользователя

Создайте в проекте `SportStore.API` папку `Entities` , в которой создайте класс **User**

```
public class User{

    public Guid Id {get; set;}
    public string Name {get ;set;}

}
```

Замечание: тип Guid будет пока использоваться для локальной разработки без использования базы данных

Интерфейсы

Создайте папку Interfaces и поместите следующий класс

```
public interface IUserRepository
{
   User CreateUser(User user);
   List<User> GetUsers();
   User EditUser(User user, Guid id);
   bool DeleteUser(Guid id);
   User FindUserById(Guid id);
}
```

Паттерн репозиторий

Создайте папку `Repositories` и поместите там следующий класс, который будет имлементировать (реализовывать) интерфейс IUserRepository.

```
public class UserRepository : IUserRepository
{
    public IList<User> Users { get; set; } = new List<User>();

    public User CreateUser(User user)
    {
       user.Id = Guid.NewGuid();
       Users.Add(user);
       return user;
    }

    public bool DeleteUser(Guid id)
    {
        var result = FindUserById(id);
        Users.Remove(result);
        return true;
    }

    /// <summary>
    /// Редактирование пользователя
    /// </summary>
    /// <param name="user"></param>
    /// <param name="id"></param>
    /// <returns></returns>
    public User EditUser(User user, Guid id)
    {
       var result = FindUserById(id);
       // update
       result.Name = user.Name;
       return result;
    }

    public User FindUserById(Guid id)
    {
        var result = Users.Where(u => u.Id == id).FirstOrDefault();

       if(result == null){
         throw new Exception($"Нет пользователя с id = {id}");
       }

       return result;
    }

    public List<User> GetUsers()
    {
       return (List<User>)Users;
    }
}
```

Реализация CRUD в UserRepository

Задание: реализуйте методы, которые будут составлять CRUD операции для User

Unit Test

Для реализации unit-тестирования функциональности методов репозитория создадим проект:

```
dotnet new xunit -o SportStore.Tests
```

Создайте класс `UserRepositoryTests`

```
public class UserRepositoryTests
{
    private readonly UserRepository _userRepository;
    public UserRepositoryTests()
    {
        _userRepository = new UserRepository();
    }

    [Fact]
    public void CreateUser_ShouldReturnNewUserWithGeneratedId()
    {
        // Arrange
        var newUser = new User { Name = "Test User" };
        // Act
        var createdUser = _userRepository.CreateUser(newUser);
        // Assert
        Assert.NotNull(createdUser);
        Assert.NotEqual(Guid.Empty, createdUser.Id);
        Assert.Equal(newUser.Name, createdUser.Name);
    }

    [Fact]
    public void DeleteUser_ShouldReturnTrueAndRemoveUser()
    {
        // Arrange
        var userRepository = new UserRepository();
        var testUser = new User { Id = Guid.NewGuid(), Name = "Test User" };
        userRepository.Users.Add(testUser);

        // Act
        bool result = userRepository.DeleteUser(testUser.Id);

        // Assert
        Assert.True(result);
        Assert.Empty(userRepository.Users);
    }

    [Fact]
    public void EditUser_ShouldUpdateExistingUser()
    {
        // Arrange
        var userRepository = new UserRepository();
        var originalUser = new User { Id = Guid.NewGuid(), Name = "Original User" };
        userRepository.Users.Add(originalUser);

        // Act
        var editedUser = new User { Id = originalUser.Id, Name = "Edited User" };
        var result = userRepository.EditUser(editedUser, originalUser.Id);

        // Assert
        Assert.NotNull(result);
        Assert.Equal("Edited User", result.Name);
        Assert.Single(userRepository.Users);
    }

    [Fact]
    public void FindUserById_ShouldReturnUserByValidId()
    {
        // Arrange
```

```
        var userRepository = new UserRepository();
        var testUser = new User { Id = Guid.NewGuid(), Name = "Test User" };
        userRepository.Users.Add(testUser);

        // Act
        var foundUser = userRepository.FindUserById(testUser.Id);

        // Assert
        Assert.NotNull(foundUser);
        Assert.Equal(testUser.Id, foundUser.Id);
        Assert.Equal(testUser.Name, foundUser.Name);
    }

    [Fact]
    public void FindUserById_ShouldThrowExceptionForInvalidId()
    {
        // Arrange
        var userRepository = new UserRepository();

        // Act & Assert
        Assert.Throws<Exception>(() => userRepository.FindUserById(Guid.NewGuid()));
    }

    [Fact]
    public void GetUsers_ShouldReturnAllUsers()
    {
        // Arrange
        var userRepository = new UserRepository();
        var testUser1 = new User { Id = Guid.NewGuid(), Name = "User 1" };
        var testUser2 = new User { Id = Guid.NewGuid(), Name = "User 2" };
        userRepository.Users.Add(testUser1);
        userRepository.Users.Add(testUser2);

        // Act
        var users = userRepository.GetUsers();

        // Assert
        Assert.NotNull(users);
        Assert.Equal(2, users.Count);
        Assert.Contains(testUser1, users);
        Assert.Contains(testUser2, users);
    }

    [Fact]
    public void FindUserById_ShouldThrowExceptionForNonExistentId()
    {
        // Arrange
        var userRepository = new UserRepository();

        // Act & Assert
        Assert.Throws<Exception>(() => userRepository.FindUserById(Guid.NewGuid()));
    }
}
```

- запуск всех тестов `dotnet test`
- просмотр все доступных тестов `dotnet test --list-tests`
- запуск конкретного списка по фильтру

```
dotnet test dotnet test --filter "FullyQualifiedName=xunit.UserRepositoryTests.FindUserById_ShouldThrowExceptionForNonExistentId"
```

Создание UsersCotroller для управления пользователями

```
[ApiController]
[Route("[controller]")]
public class UserController : ControllerBase
{
    private readonly IUserRepository _repo;
    public UserController(IUserRepository repo)
    {
       _repo = repo;
    }

    [HttpPost]
    public ActionResult CreateUser(User user){

        var validator = new FluentValidator();
        var result = validator.Validate(user);
        if(!result.IsValid){
            throw new Exception($"{result.Errors.First().ErrorMessage}");
        }

        Ok(_repo.CreateUser(user))
    }

    [HttpGet]
    public ActionResult GetUser(){
        return Ok(_repo.GetUsers());
    }


    [HttpPut]
    public ActionResult UpdateUser(User user){
       return Ok(_repo.EditUser(user, user.Id));
    }


    [HttpGet("{id}")]
    public ActionResult GetUserById(Guid id){
       return Ok(_repo.FindUserById(id));
    }


    [HttpDelete]
    public ActionResult DeleteUser(Guid id){
        return Ok(_repo.DeleteUser(id));
    }

}
```

Задание: при запросе post на создание нового ресурса обычно принято отвечать кодом 201. Примените метод `Created` для возрата ответа типа `ActionResult`

DI

Для контроллер UsersController запрашивает в своем конструкторе

```
    private readonly IUserRepository _repo;
    public UserController(IUserRepository repo)
    {
       _repo = repo;
    }
```

реализацию интерфейса `IUserRepository` , который ему должен предоставить DI (Dependency Injection) - контейнер внедрения зависимости встроенный во фреймворк ASP Core. Для этого надо зарегистрировать сервис в коллекции сервимов в проекте API.

```
builder.Services.AddSingleton<IUserRepository, UserRepository>();
```

Validation

DataAnnotation

При отравке пост запросов надо проверять модель данных на соответствие валидности. Для этого применяются инструменты: встроенное средства проверки `DataAnnotation` и пакет `FluentValidation` .

Атрибут для проверки минимального длины имени

```
  [MinLength(5,ErrorMessage = "Минимальное длина имени 5")]
  public string Name {get ;set;} = string.Empty;
```

Для создания собственного атрибута валидации DataAnnotation создайте папку `Validations` и в ней создайте класс `UserValidator`

```
public class MaxLengthAttribute : ValidationAttribute
{
    private readonly int _maxLength;

    public MaxLengthAttribute(int maxLength) : base($"Name max {maxLength} ")
    {
        _maxLength = maxLength;
    }

    public override bool IsValid(object? value)
    {
        return ((String)value!).Length <= _maxLength;
    }
}
```

Таким образом модель будет выглядеть следующим образом:

```
public class User
{
    public Guid Id {get ;set;} = Guid.NewGuid();

    [MinLength(5,ErrorMessage = "Минимальное длина имени 5")]
    [SportStore.API.Validations.MaxLength(10)]
    public string Name {get ;set;} = string.Empty;
}
```

FluentValidation

Установите пакет `FluentValidation`

Создайте в папке Validation новый класс.

```
    public class FluentValidator : AbstractValidator<User>
    {
        public FluentValidator()
        {
            RuleFor(u => u.Name).Must(StartsWithCapitalLetter).WithMessage("Имя пользователя должно начинаться с заглавной буквы");
        }

        private bool StartsWithCapitalLetter(string username)
        {
            return char.IsUpper(username[0]);
        }
    }
```

Для применения валидатора к конечной точки

```
        var validator = new FluentValidator();
        var result = validator.Validate(user);
        if(!result.IsValid){
            throw new Exception($"{result.Errors.First().ErrorMessage}");
        }
```

Задание: проверьте метод контроллера создания пользователя, чтобы имя пользователя начиналось с заглавной буквы.

Postman(Swagger,request.http) для тестирования API

- Способ 1
  Протестируйте работу API на примере управления пользователями с помощью встроенного средства Swagger по адресу http://localhost:5290/swagger
- Способ 2. Postman
- Способ 3. Запросы .http

Создайте в корнейвой директории папку requests в которой создайте файл с расширением http. Например, getusers.http

```
GET http://localhost:5290/User
```

postuser.http

```
POST http://localhost:5290/User
Content-Type: application/json

{
  "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "name": "Newuser123213131"
}
```

Проверка запросов осуществляется с помощью VS Code.

Задание: у пользователя должна быть роль. Создайте модель для роли пользователя, интерфейс, репозиторий, контроллер, валидации, напишите unit-тесты для репозитории.

Асинхронность. Работа с Task

Entity Framework Core

Установить пакеты:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Tools
- Microsoft.EntityFrameworkCore.Design

При установке пакетов надо соблюдать версионность относительно версии .net. В данном приложении применяется `net7.0`

В результате добавления пакетов `project` файл `SportStore.API` будет выглядеть следующим образом:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="7.0.5" />
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="7.0.20" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="7.0.20">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="7.0.20">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.4.0" />
  </ItemGroup>

</Project>
```

Создайте папку `Data` и добавьте класс `SportStoreContext`

```
public class SportStoreContext : DbContext
{
    public DbSet<User> Users { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseNpgsql("Host=localhost;Port=5432;Database=SportStoreCourse;Username=;Password=");
    }
}
```

В параметрах подключения к базе данных поставьте свой UserName и Password

Миграции

В рабочей директории создайте первую миграцию.

```
dotnet ef migtations add Initial -s SportStore.API -p SportStore.API
```

Далее, выполните эту миграцию. То есть EF создаст реальные таблицы в базе данных PostgreSQL.

```
dotnet ef update database -s SportStore.APi -p SportStore.API
```

Замечание: -s - это стартовый проект, -p - это текущий проект. Либо, можно зайти в проект `SportStore.API` явно и не прописывать данные параметры.

Результат:

- branch:spring1:Introduction
- pullRequest