# ساخت مدل

|  |  |  |  |
| --- | --- | --- | --- |
|  | -- | | |
| شماره سند: | شناسه سند | |  |
| طـبقه‌بندی: | عادی/حساس/محرمانه | |  |
| تــاریــخ: | روز/ماه/سال |  | |

v

ساخت کلاس User در Domain

Domain > Entites > Users > User.cs

Entity به معنای موجودیت‌هایی است که در بانک اطلاعاتی شما وجود دارند. این موجودیت‌ها ممکن است مربوط به جداول در بانک اطلاعاتی باشند. در Entity Framework Core، هر موجودیت به‌صورت یک کلاس تعریف می‌شود که ویژگی‌های متناظر با ستون‌های جدول را دارد. همچنین، ارتباطات بین جداول در بانک اطلاعاتی نیز می‌توانند با استفاده از خصوصیت‌های مربوطه در موجودیت‌ها تعریف شوند. Entity به‌عنوان مدل داده‌ای استفاده می‌شود که اطلاعات مورد نیاز برنامه را نشان می‌دهد.

نمونه ای از کلاس User در زیر آمده است:

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Domain.Entites.User

{

public class User

{

#region Properties

[Key]

public int UserId { get; set; }

[Display(Name = "ایمیل")]

[Required(ErrorMessage = "لطفا {0} را وارد کنید")]

[MaxLength(350, ErrorMessage = "حداکثر کاراکتر مجاز {0} می باشد")]

public string Email { get; set; }

[Display(Name = "کلمه عبور")]

[Required(ErrorMessage = "لطفا {0} را وارد کنید")]

[MaxLength(350, ErrorMessage = "حداکثر کاراکتر مجاز {0} می باشد")]

public string Password { get; set; }

[Display(Name = "نام کاربری")]

[Required(ErrorMessage = "لطفا {0} را وارد کنید")]

[MaxLength(350, ErrorMessage = "حداکثر کاراکتر مجاز {0} می باشد")]

public string UserName { get; set; }

[Display(Name = "شماره موبایل")]

[MaxLength(11, ErrorMessage = "حداکثر کاراکتر مجاز {0} می باشد")]

public string? Mobile { get; set; }

[Display(Name = "آواتار")]

[MaxLength(50, ErrorMessage = "حداکثر کاراکتر مجاز {0} می باشد")]

public string Avatar { get; set; }

[Display(Name = "ایمیل فعال باشد؟")]

public bool IsEmailActive { get; set; }

[Display(Name = "تاریخ ثبت نام")]

public DateTime RegisterDate { get; set; }

#endregion

}

}

# ساخت Context

ساخت کلاس DatingAppContext در Data

Data > Context > DatingAppContext.cs

Context یا همان DbContext در Entity Framework Core، یک کلاس است که به‌عنوان واسطه بین برنامه و بانک اطلاعاتی عمل می‌کند. این کلاس شامل تمام موجودیت‌ها (Entities) و جداول مربوطه در بانک اطلاعاتی است. Context شامل متدهایی است که به برنامه امکان ایجاد، حذف، بروزرسانی و استعلام داده‌ها در بانک اطلاعاتی را می‌دهد. همچنین، این کلاس مسئول برقراری ارتباط با بانک اطلاعاتی، مدیریت تراکنش‌ها و مدیریت تغییرات روی داده‌ها است.

به‌طور خلاصه، مفهوم Entity در ASP.NET Core به موجودیت‌های داده‌ای مربوط به برنامه و بانک اطلاعاتی اشاره دارد، در حالی که Context مسئول مدیریت و ارتباط با بانک اطلاعاتی است. با استفاده از Entity Framework Core، شما می‌توانید موجودیت‌ها را تعریف کنید و با استفاده از Context، عملیات مربوط به داده‌ها را انجام دهید.

نمونه از کلاس Context در زیر آمده است:

using Domain.Entites.User;

using Microsoft.EntityFrameworkCore;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Data.Context

{

public class DatingAppContext:DbContext

{

#region Constructor

public DatingAppContext(DbContextOptions<DatingAppContext> options):base(options)

{

}

#endregion

#region Users

public DbSet<User> Users { get; set; }

#endregion

}

}

# ساخت Migration

با اجرای دستور زیر در Package Manager Console یک Migration ایجاد می کنیم که اطلاعات Entity در پایگاه داده ایجاد شود.

توجه داشته باشید که حتما قبل از اجرای Migration کانکشن اتصال به پایگاه داده را ایجاد کرده باشید.

Add-Migration AddUserTable

Update-Database

# ساخت Repository

ساخت Interface با نام IUserRepository در Domain

Domain > Interfaces > IUserRepository

Repository یک الگوی طراحی است که به منظور جدا کردن لایه دسترسی به داده‌ها از لایه‌های بالاتر در برنامه استفاده می‌شود. هدف اصلی Repository، فراهم کردن روشی استاندارد برای عملیات مربوط به داده‌ها از جمله خواندن، نوشتن، به روزرسانی و حذف آنها است.

Repository میان لایه‌های بالاتر برنامه (مانند لایه سرویس و لایه کنترل کننده) و لایه دسترسی به داده‌ها (مانند لایه دسترسی به پایگاه داده) واسطی را ایجاد می‌کند. این واسط (Interface) معمولاً شامل متدهایی است که عملیات CRUD (ساخت، خواندن، به روزرسانی و حذف) را برای داده‌ها تعریف می‌کند.

با استفاده از Repository، لایه‌های بالاتر برنامه نیازی به دانستن جزئیات پیاده‌سازی دسترسی به داده‌ها ندارند. آنها فقط با رابط (Interface) Repository در ارتباط هستند و می‌توانند از طریق آن عملیات مورد نیاز خود را انجام دهند. این مزیت اصلی استفاده از Repository در معماری ASP.NET Core است که باعث جدا کردن مسئولیت‌ها و افزایش قابلیت تست و قابلیت استبدال پذیری در برنامه می‌شود.

using Domain.Entites.User;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Domain.Interfaces

{

public interface IUserRepository

{

#region User

Task<IEnumerable<User>> GetAllUsersAsync();

Task<User?> GetUserByUserIdAsync(int userId);

#endregion

}

}

سپس یک کلاس Repository با نام UserRepository که از IUserRepository اثربری می کند در مسیر زیر ایجاد می کنیم.

Data > Repositories > UserRepository > UserRepository

using Data.Context;

using Domain.Entites.User;

using Domain.Interfaces;

using Microsoft.EntityFrameworkCore;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Data.Repositories

{

public class UserRepository : IUserRepository

{

#region Constructor

private readonly DatingAppContext \_context;

public UserRepository(DatingAppContext context)

{

\_context = context;

}

#endregion

#region User

public async Task<IEnumerable<User>> GetAllUsersAsync()

=> await \_context.Users.ToListAsync();

public async Task<User?> GetUserByUserIdAsync(int userId)

=> await \_context.Users.FirstOrDefaultAsync(x => x.UserId == userId);

}

}

# ساخت Repository در سرویس

مشابه همان Interface را در لایه سرویس هم می سازیم

Application > Services > Interfaces > IUserService

using Domain.DTOs.Account;

using Domain.Entites.User;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Application.Services.Interfaces

{

public interface IUserService

{

#region User

Task<IEnumerable<User>> GetAllUsersAsync();

Task<User?> GetUserByUserIdAsync(int userId);

#endregion

}

}

و سپس یک کلاس همانند کلاس Repository با نام UserService که از IUserService ارثبری می کند در مسیر زیر می سازیم.

Application > Services > Implemetation > UserService

using Application.Services.Interfaces;

using Domain.Entites.User;

using Domain.Interfaces;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Application.Services.Implementation

{

public class UserService : IUserService

{

#region Constructor

private readonly IUserRepository \_userRepository;

public UserService(IUserRepository userRepository)

{

\_userRepository = userRepository;

}

#endregion

#region User

public async Task<IEnumerable<User>> GetAllUsersAsync()

=> await \_userRepository.GetAllUsersAsync();

public async Task<User?> GetUserByUserIdAsync(int userId)

=> await \_userRepository.GetUserByUserIdAsync(userId);

#endregion

}

}

# تزریق وابستگی ها

یک کلاس با نام DependencyContainer در مسیر زیر ایجاد می کنیم.

IOC > Dependencies > DependencyContainer

در این کلاس تمام سروریس ها و ریپوزیتوری هایی که ساخته ایم را در این کلاس ثبت می کنیم.

using Application.Services.Implementation;

using Application.Services.Interfaces;

using Data.Repositories;

using Domain.Interfaces;

using Microsoft.Extensions.DependencyInjection;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace IOC.Dependencies

{

public static class DependencyContainer

{

public static void RegisterServices(this IServiceCollection services)

{

#region Service

services.AddScoped<IUserService, UserService>();

#endregion

#region Repository

services.AddScoped<IUserRepository, UserRepository>();

#endregion

}

}

}

# معرفی سروریس ها

برای معرفی سروریس ها در Program.cs کد زیر را در مسیر زیر قرار می دهیم.

Presentations > Program.cs

#region Add Services

builder.Services.RegisterServices();

#endregion

# ساخت کنترلر API

یک کنترلر با نوع API و با ویژگی read/write action ایجاد می کنیم.

using Application.Services.Interfaces;

using Domain.Entites.User;

using Microsoft.AspNetCore.Mvc;

// For more information on enabling Web API for empty projects, visit https://go.microsoft.com/fwlink/?LinkID=397860

namespace DatingApp.Api.Controllers

{

public class UserController : BaseSiteController

{

#region Constructor

private readonly IUserService \_userService;

public UserController(IUserService userService)

{

\_userService = userService;

}

#endregion

// GET: api/<UserController>

[HttpGet]

public async Task<IEnumerable<User>> Get()

{

return await \_userService.GetAllUsersAsync();

}

// GET api/<UserController>/5

[HttpGet("{id}")]

public async Task<User?> Get(int id)

{

return await \_userService.GetUserByUserIdAsync(id);

}

// POST api/<UserController>

[HttpPost]

public void Post([FromBody] string value)

{

}

// PUT api/<UserController>/5

[HttpPut("{id}")]

public void Put(int id, [FromBody] string value)

{

}

// DELETE api/<UserController>/5

[HttpDelete("{id}")]

public void Delete(int id)

{

}

}

}

در این کد متد Get به دو شکل استفاده شده است، تمام یوزر ها و یک یوزر خاص با ID مشخص.