🤯

Practico 1 - @September 1, 2022

Nombre completo:

Numero de estudiante:

Se pide:

A partir de los ejemplo visto en clase construir una aplicación con las capas dataAccess. domain, businessLogic y WebAPI para poder manejar los usuarios de una empresa, esto.

Para esto tener en cuenta que un usuario debe tener los atributos: nombre, apellido, edad y dirección.

Para el manejo del sistema se desea implementar las operaciones, **agregar usuario (add), obtener todos los usuarios (getAll) y borrar un usuario (delete) .**

Entregar en aulas

- Código de la solución o link a un repo de github
- Colección de postman con los endpoints; post, delete y get correspondientes a las funcionalidades pedidas

---

Para este ejercicio puede servir tener en cuenta los siguientes pasos vistos en clase:

1- Vamos a ver una solución completa con todas las capas para los que vamos a crear los proyectos:

- Domain → Class Library
- BusinessLogic → Class Library
- IBusinessLogic → Class Library (opcional)
- DataAccess → Class Library
- IDataAccess → Class Library (opcional)
- WebAPI → ASP.NET Core Web Application type web API

2- En el proyecto Domain comenzamos creando la clase Order que es el dominio que nuestro sistema va a manejar

```
public class Order
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set;}
    public int PurchaseNumber { get; set;}
    public int Price { get; set;}
    public DateTime DeliveryDateTime { get; set; }
}
```

3- Creamos la clase contextDB dentro del paquete DataAccess

Para esto antes instalamos los paquetes:

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.SqlServer

```
public class ContextDb: DbContext
    {
        public DbSet<Order> Orders {get; set;}

        protected override void OnModelCreating (ModelBuilder modelBuilder)
        {

        }
        public ContextDb(DbContextOptions options) : base(options) {}

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
        }

    }
```

4- En el proyecto IDataAccess creamos el repositorio que se encargá de explicitar los metodos que se van a usar para interactuar con el context que contiene llos dbset de las representaciones de las tablas.

Nosotros ahora solo usarmos dos de estas funciones el add y get pero dejamos todas para que ya les quede de ejemplo la firma.

```
public interface IOrderRepository {

    void Add (Order entity);
    void Remove (Order entity);
    void Update(Order entity);
    IQueryable<Order> GetAll();
}
```

5- En DataAccess creamos la implementacion de la interfaz anterior que es la que se encarga de interactuar con el contextDb (nuestra puerta a la bdd) para hacer las operaciones CRUD sobre este.

```
public class OrderRepository: IOrderRepository
{
    private ContextDb _contextDb;

    public OrderRepository(ContextDb contextDb)
    {
        _contextDb = contextDb;
    }

    public void Add(Order entity)
    {
        _contextDb.Orders.Add(entity);
        _contextDb.SaveChanges();
    }
```

```
    public void Remove(Order entity)
    {
        throw new NotImplementedException();
    }

    public void Update(Order entity)
    {
        throw new NotImplementedException();
    }

    public IQueryable<Order> GetAll()
    {
       return _contextDb.Orders;

    }
}
```

6- Creamos ahora en IBusinessLogic la interface IOrderService

```
public interface IOrderService
{
    Order Get (int id);
    Order Create (Order order);
    IQueryable<Order> GetAll();
    void Delete(int id);
    void Update(int id, Order entity) ;
}
```

7- Creamos ahora en businessLogic la implementación de la clase anterior que sera OrderService

```
public class OrderService : IOrderService
{
    private IOrderRepository repository;
    public OrderService(IOrderRepository orderRepository)
    {
        repository = orderRepository;
    }

    public Order Get(int id)
    {
        throw new NotImplementedException();
    }

    public Order Create(Order order)
    {
         repository.Add(order);
         return order;
    }

    public IQueryable<Order> GetAll()
    {
        return repository.GetAll();

    }

    public void Delete(int id)
    {
        throw new NotImplementedException();
    }

    public void Update(int id, Order entity)
```

```
    {
        throw new NotImplementedException();
    }
}
```

8- Antes de comenzar a hacer el controller debemos hacer algunas modificaciones en las clases Program y agregar la clase Startup

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder => { webBuilder.UseStartup<Startup>(); });
}
```

```
public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddCors(options =>
            {
                options.AddPolicy("AllowEverything",builder => builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod());
            });
            services.AddControllers();
            services.AddScoped<IOrderRepository, OrderRepository>();
            services.AddScoped<IOrderService, OrderService>();
            string directory = System.IO.Directory.GetCurrentDirectory();
            IConfigurationRoot configuration = new ConfigurationBuilder()
                .SetBasePath(directory)
                .AddJsonFile("appsettings.json")
                .Build();
            var connectionString = configuration.GetConnectionString("DBClase2");
            services.AddDbContext<ContextDb>(options =>
                options.UseSqlServer(connectionString));

        }


        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();
```

```
            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints => { endpoints.MapControllers(); });
        }
    }
```

9- Finalmente crearemos el controller

```
[ApiController]
[Route("api/orders")]
public class OrderController: ControllerBase {

    private IOrderService _orderService;

    public OrderController(IOrderService orderService){
        _orderService = orderService;
    }

    [HttpGet]
    public IActionResult Get()
    {
        IQueryable<Order> orders = _orderService.GetAll();
        return Ok (orders);
    }


   [HttpPost]
   public IActionResult Add([FromBody] Order order)
   {
       Order orderCreated = _orderService.Create(order);
       return Created("", orderCreated);
   }


}
```

9- Instalamos el paquete EntityFrameworkCore.Design en el paquete DataAccess y en WebAPI para poder hacer las migrations, para ejecutar estos comandos nos paramos en DataAccess y referenciamos a webAPI que es donde iniciamos nuestra aplicación y construimos el contextDB.

```
dotnet ef migrations add MyMigration --startup-project "../WebAPI"
dotnet ef database update --startup-project "../WebAPI"
```

10- Corremos la solución y probamos los endpoints en postman.