Crear cuenta de usuario

Antes estaba de moda N capas

* Datos de entrada:
* Correo
* Password
* Confirmación de password
* Nombre y Apellido

El sistema debe validar los datos

El sistema debe registrar los datos

El sistema debe devolver una respuesta afirmativa.

Si los datos son incorrectos se detiene el flujo e informa al usuario los errores encontados.

Validación: Correo, pasword y confirmación son obligatorios.

NCapas

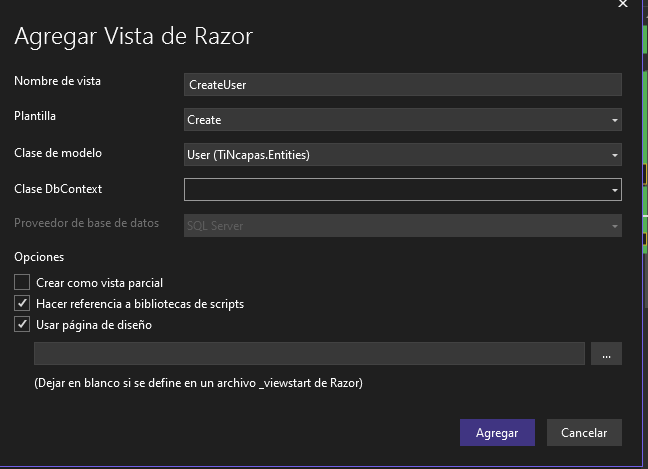
Dice que no trabaja con inyección de dependencias.

1. Crea capa de Entidad o dominio
2. Luego capa de acceso a datos. Depende de entidades.

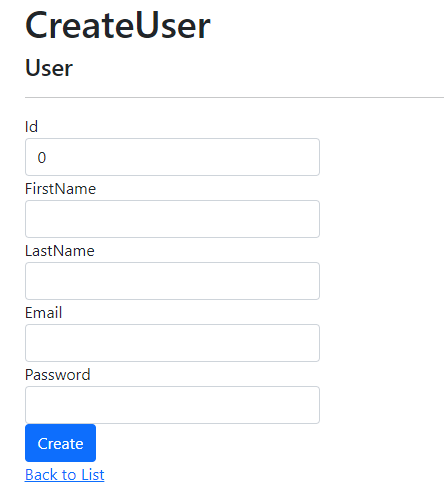
Hace falta una referencia a las entidades, dado que estoy trabajando con EF:UserManagerDbContext, que herede de DbContext.

1. Agrega capa de negocios. Depende de DAL..
2. Como es proyecto MVC, están la vista y el controlador. Este depende del bussiness (Servicios).

Desde DAL hace la migración: add-migration InitialCreate (también debe estar como proyecto de inicio). Luego update-database



Click derecho en el controlador, agregar vista, y se pone esto:



Así se ve, hacen falta las validaciones. Desde la capa MVC, agrega carpeta viewModels y ahí si tira los dataAnottations.

TIRA MAGIA

public class UserViewModel

{

public string FirstName { get; set; }

public string LastName { get; set; }

[Required(ErrorMessage = "El correo es requerido")]

[EmailAddress(ErrorMessage = "El formato de correo no es válido")]

public string Email { get; set; }

[Required(ErrorMessage = "La contraseña es requerida")]

[DataType(DataType.Password)]

public string Password { get; set; }

[DataType(DataType.Password)]

[Compare("Password",ErrorMessage ="La contraseña y la confirmación no coinciden")]

public string ConfirmPassword { get; set; }

}

Luego, el controlador, ya no usa las entidades, sino el userViewModel:

PASA DE ESTO:

[HttpGet]

public IActionResult Create()

{

var User=new User();

return View(User);

}

[HttpPost]

public IActionResult Create(User user)

{

userService.CreateUser(user);

return View(User);

}

A ESTO:

if (ModelState.IsValid) {}

Eso significa que se superan las validaciones de los data anottations.

HttpGet]

public IActionResult Create()

{

var UserViewModel=new UserViewModel();

return View(UserViewModel);

}

[HttpPost]

public IActionResult Create(UserViewModel userViewModel)

{

if (ModelState.IsValid)

{

User user = new User()

{

Email = userViewModel.Email,

FirstName = userViewModel.FirstName,

LastName = userViewModel.LastName,

Password = userViewModel.Password

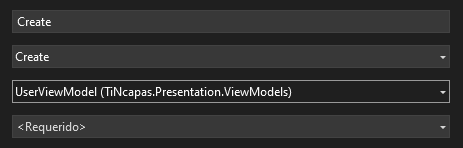
};

UserService.CreateUser(user);

}

return View(userViewModel);

}



Ahí ya funca y valida. Aunque el validador de email no anda joya. Puse .Co en vez de .com y me lo tomó.

Desventajas del modelo

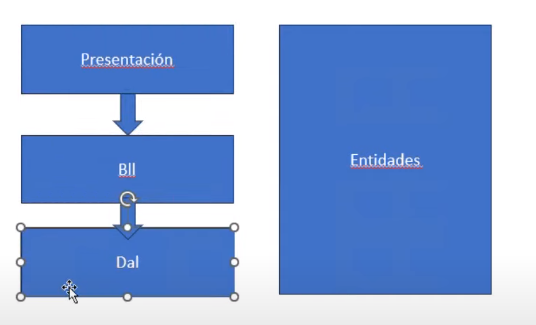
Es muy difícil de testear.

No hay inyección de dependencias.

Está muy acoplado: Si se toca algo, se deben tocar muchas clases y capas mas.

Hay que modificar distintas capas para realizar un simple cambio de la presentación.

Lo mismo si se cambia la BBDD.

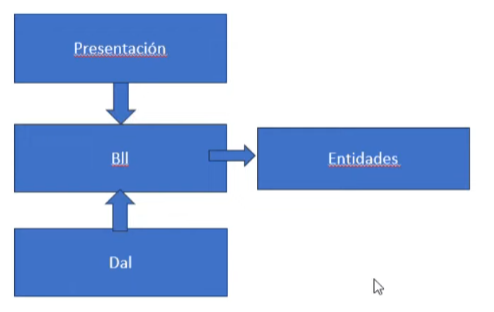


Así lo representa. Todas ellas pueden hacer referencia a las entidades.

Dice que DAL es la mas importante, si cambia ella, deben cambiar todas las demás.

Arquitectura limpia

Lo mas importante es la capa bussiness (servicios). Se le pegotean las entidades.



Toma su proyecto y lo cambia

Modifica las referencias:

BLL hace referencia a entidades.

DAL hace referencia a BLL (inversión de dependencias)

Presentacion hace referencia a BLL.

Inversión de dependencias:

No se debe depender de implementaciones sino de abstracciones.

Eso es una implementación: UserManagerDbContext context;

Primero se crean las abstracciones y luego las implementaciones cuando se hace clean architecture.

En este caso, las abstracciones las pone en Entidades. No lo da como regla general, también podría ponerse en BLL, así como en un proyecto nuevo.

La interfaz dice qué quieres, no cómo lo quieres.

public interface IUserRepository

{

User CreateUser(User user);

}

El Bussiness pasa de :

UserManagerDbContext context;

//Acá se ve que no hay inyección de dependencias:

//No llega nada por parámetro y debe crearse el contexto aquí.

public UserService()

{

context = new UserManagerDbContext();

}

public void Dispose()

{

context.Dispose();

}

A:

readonly IUserRepository repositorio;

//Acá se ve que no hay inyección de dependencias:

//No llega nada por parámetro y debe crearse el contexto aquí.

public UserService(IUserRepository repository)

{

repositorio = repository;

}

public void CreateUser(User user)

{

repositorio.CreateUser(user);

}

Crea UserRepository en DAL.

public class UserRepository : IUserRepository,IDisposable

{

readonly UserManagerDbContext context;

public UserRepository()

{

context = new UserManagerDbContext();

}

public User CreateUser(User user)

{

context.Add(user);

context.SaveChanges();

return user;

}

public void Dispose()

{

context.Dispose();

}

}

Esto al controlador:

readonly UserService UserService;

public UserController(UserService userService )

{

UserService = userService;

}

Pero no es así, ya que es el framework quien crea una instancia del usercontroller, es el quien debe pasar el servicio.

OYE FRAMEWORK, PASAME EL USERSERVICE.

Agrega un nuevo proyecyo IoC Inversion of control con clase DependencyContainer. Este depende de BLL y DAL.

public static class DependencyContainer

{

public static IServiceCollection AddUserManagerServices(this IServiceCollection services)

{

services.AddScoped<UserService>();

services.AddScoped<IUserRepository,UserRepository>();

return services;

}

}

Agrega otro dependencyContainer para la DAL.

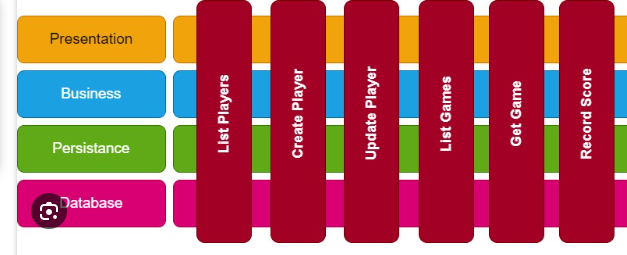
En el program se agrega una referencia a IoC

Tras todo el quilombete este. En program queda solo esto, y no cada uno de los servicios que se agregarían.

builder.Services.AddUserManagerServices();

Jimmy Bogard - vertical slices

Tiene una propuesta distinta a la clean architecture de Robert C Martin



Sería algo así. El énfasis está en las funciones, cada una tendría su propia arquitectura, por así decir.