📝

Clase 4 @September 15, 2022

- Test con Data Access
- Filters

Test sobre DataAccess

Ahora pasaremos a probar DataAccess. Nuevamente vamos a tener que crear un proyecto de prueba pero con el nombre DataAccess.Tests.

1. *dotnet new mstest -n DataAccess.Tests*
   dotnet sln add DataAccess.Tests
   cd DataAccess.Tests
   dotnet add package Microsoft.EntityFrameworkCore
   dotnet add package Microsoft.EntityFrameworkCore.InMemory
   dotnet add reference ../DataAccess
   dotnet add reference ../Domain

¿Que tipo de prueba queremos hacer?

Antes de continuar analisemos que tipo de pruebas queremos hacer.

La dependencia de nuestros repositorios es de DbContext, una clase que no esta a nuestra alcance. Pero muchas veces cuando pensamos probar esta capa queremos ver un impacto en una base de datos de prueba o en una base de datos en memoria

Hasta ahora creamos el proyecto, lo agregamos a la solución y le agregamos la referencia de proyectos que va a necesitar.

2. Creemos una clase que se llama OrderRespositoryTest.
3. Nuestra clase OrderRespositoryTest quedaria asi inicialmente:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DataAccess.Test;

[TestClass]
public class OrderRespositoryTest
{
    [TestMethod]
    public void TestGetAllOrdersOk()
    {
    }
}
```

4. Creamos las ordenes y el contexto

```
using System;
using Domain;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DataAccess.Test;

[TestClass]
public class OrderRespositoryTest
{
    [TestMethod]
    public void TestGetAllOrdersOk()
    {
        List<Order> ordersToReturn = new List<Order>()
        {
            new Order()
            {
                Id = 1,
                Name = "Papas fritas",
                Address = "Juan Gomez 3878",
                PutchaseNumber = 1,
                Price = 100,
                DeliveryDateTima = DateTime.Now.AddHours(1)
            },
            new Order()
            {
                Id = 2,
                Name = "Milanesa",
                Address = "Juan Gomez 3878",
                PutchaseNumber = 1,
                Price = 150,
                DeliveryDateTima = DateTime.Now.AddHours(1)
            }
        };
        var options = new DbContextOptionsBuilder<ContextDb>()
                    .UseInMemoryDatabase(databaseName: "TacosDB").Options;
        var context = new ContextDb(options);
        ordersToReturn.ForEach(m => context.Orders.Add(m));
        context.SaveChanges();
        var repository = new OrderRepository(context);
    }
}
```

Analicemos este parte del código:

1. Primero creamos la lista de ordenes que se van a obtener desde la base de datos en memoria.
2. Luego se tiene que crear la configuración de nuestro proveedor de base de datos para nuestro contexto.
3. Creamos nuestro contexto y le pasamos la configuración recien creada.
4. Una vez que tenemos nuestra configuración necesitamos agregarle las ordenes para que luego las retorne el contexto.
5. Una vez que se agregaron las ordenes hay que guardar esos cambios.

6. Una vez que tenemos el contexto se lo tenemos que pasar a nuestro repositorio que es la dependencia que necesita y es lo que queremos probar.

5. Llamamos el GetAll del repositorio

```
using System;
using Domain;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DataAccess.Test;

[TestClass]
public class OrderRespositoryTest
{
    [TestMethod]
    public void TestGetAllOrdersOk()
    {
        List<Order> ordersToReturn = new List<Order>()
        {
            new Order()
            {
                Id = 1,
                Name = "Papas fritas",
                Address = "Juan Gomez 3878",
                PutchaseNumber = 1,
                Price = 100,
                DeliveryDateTima = DateTime.Now.AddHours(1)
            },
            new Order()
            {
                Id = 2,
                Name = "Milanesa",
                Address = "Juan Gomez 3878",
                PutchaseNumber = 1,
                Price = 150,
                DeliveryDateTima = DateTime.Now.AddHours(1)
            }
        };
        var options = new DbContextOptionsBuilder<ContextDb>()
                    .UseInMemoryDatabase(databaseName: "TacosDB").Options;
        var context = new ContextDb(options);
      ordersToReturn.ForEach(m => context.Add(m));
      context.SaveChanges();
      var repository = new OrderRepository(context);

      var result = repository.GetAll();
    }
}
```

6. Asserts

```
using System.Linq;
using Domain;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
```

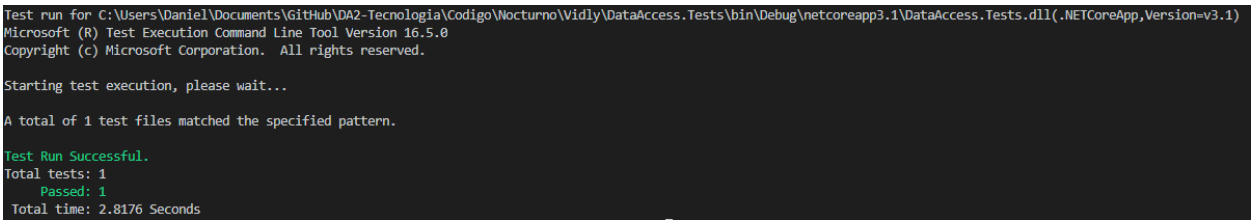
```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DataAccess.Test;

[TestClass]
public class OrderRespositoryTest
{
    [TestMethod]
    public void TestGetAllOrdersOk()
    {
        List<Order> ordersToReturn = new List<Order>()
        {
            new Order()
            {
                Id = 1,
                Name = "Papas fritas",
                Address = "Juan Gomez 3878",
                PutchaseNumber = 1,
                Price = 100,
                DeliveryDateTima = DateTime.Now.AddHours(1)
            },
            new Order()
            {
                Id = 2,
                Name = "Milanesa",
                Address = "Juan Gomez 3878",
                PutchaseNumber = 1,
                Price = 150,
                DeliveryDateTima = DateTime.Now.AddHours(1)
            }
        };
        var options = new DbContextOptionsBuilder<ContextDb>()
                    .UseInMemoryDatabase(databaseName: "TacosDB").Options;
        var context = new ContextDb(options);
      ordersToReturn.ForEach(m => context.Add(m));
      context.SaveChanges();
      var repository = new OrderRepository(context);

      var result = repository.GetAll();

      Assert.IsTrue(ordersToReturn.SequenceEqual(result));
    }
}
```

La parte de assert tampoco tiene mucha magia. Para que esta parte funcione hay que redefinir el equals en la clase Order.

```
using System;
namespace Domain;

public class Order: IEqualityComparer<Order>
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Address  { get; set; }
    public int PutchaseNumber { get; set; }
    public int Price { get; set; }
    public DateTime DeliveryDateTima { get; set; }

    public bool Equals(Order x, Order y)
    {
```

```
      if (ReferenceEquals(x, y)) return true;
      if (ReferenceEquals(x, null)) return false;
      if (ReferenceEquals(y, null)) return false;
      if (x.GetType() != y.GetType()) return false;
      return x.Id == y.Id && x.Name == y.Name;
    }

    public int GetHashCode(Order obj)
    {
      return HashCode.Combine(obj.Id, obj.Name);
    }
}
```

7. Ejecutemos los tests



Mejorando la prueba

Si nosotros quisieramos crear un unico contexto y usarlo en las diferentes pruebas, deberiamos de inicializarlo y borrarlo para lograr que las pruebas sean independientes. Esto lo logramos de la siguiente manera:

```
[TestClass]
public class OrderRepositoryTest
{
  private DbContext context;
  private DbContextOptions options;

  [TestInitialize]
  public void Setup()
  {
      this.options = new DbContextOptionsBuilder<DBContext>().UseInMemoryDatabase(databaseName: "TacosDB").Options;
      this.context = new DBContext(this.options);
  }

  [TestCleanup]
  public void TestCleanup()
  {
      this.context.Database.EnsureDeleted();
  }
}
```

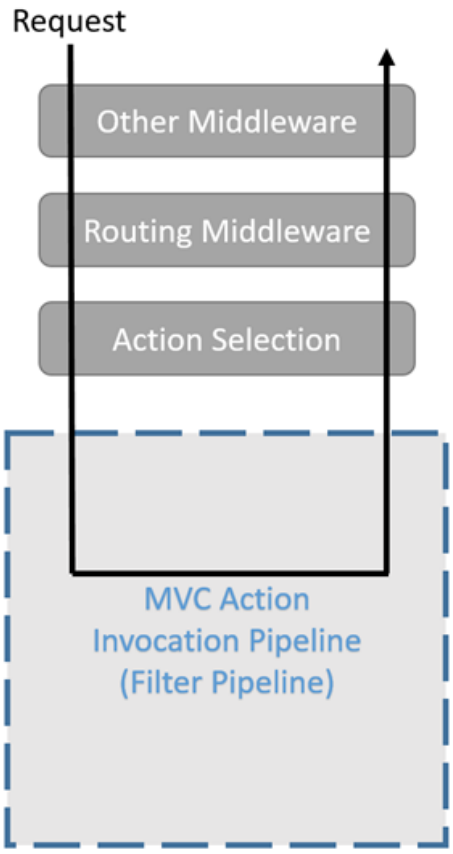
Esto lo que hace es crear el contexto antes de la prueba y hacerle un clean despues de usarlo. Con esto logramos que las pruebas sean independientes entre ellas.

Filtros

Cuando hacemos una request y se ejecuta el método del controller pasan muchas cosas en el medio.

Routing middleware → se encarga de mappear la ruta con la clase

Action Selection → corresponde con este metodo



Lo que podemos hacer es agregar pasos en el medio, y hacer ya sea una autenticacion o un control de excepciones. Hay dos formas de hacer esto, un middleware mas a bajo nivel que se ejcuta cuando llega la request y cuando sale la response, y mas a alto nivel estan los filtros, para esto ultimo Microsft ya nos brinda ciertos filtros.

¿Que son entonces los filtros?

Los filtros o *filters* nos permiten ejecutar código antes o después de determinadas fases en el procesamiento de una solicitud HTTP. Es decir podes interferir esta solicitud antes o después de que llego a nuestro *Controller*.

¿Para qué utilizarlo?

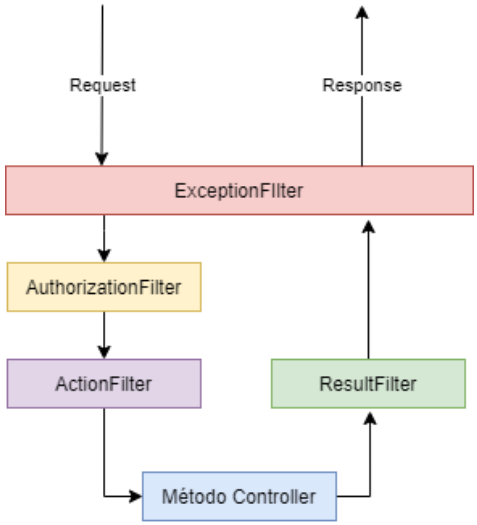
Podemos crear filtros personalizados para diferentes cuestiones, como por ejemplo el control de errores, almacenamiento *cache* , configuración, autorización y registro, entre otras. ¿Qué ganamos con esto? Evitamos la duplicación de código en aquellas instancias donde se deben aplicar los mismos procedimientos para muchos métodos del *Controller y abstraemos esas partes de codigo a un lugar mas global.*

Tipos de filtro

Cada tipo de filtro es ejecutado en una fase diferente de la solicitud, es decir tienen un órden de ejecución según su responsabilidad

- Authorization Filters → Se utiliza para aplicar la política de autorización y seguridad
- Action Filters → Para hacer una accion antes y despues de la invocacion del metodo
- Result Filters → Para hacer una accion antes y despues del resultado
- Exception filters → Para manejar excepciones a nivel global, sirve para limpiar el codigo de los controller porque se quitan todos los try catch.

Orden de los filtros



Ahora vamos a ver la implementación de tres de ellos que son:

- Action Filter
- Authorization Filter
- Exception filter

Para esto vamos a seguir los siguientes pasos

1- Creamos la clase user

```
public class User
{
    public int Id { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
    public string Token { get; set; }
    public RoleType Role { get; set; }
}
```

2- Creamos el enum de rol

```
public enum RoleType
{
    Admin=0,
    Owner=1
}
```

3- Agregamos la tabla users al context

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

4- Creamos el repositorio de usuario

```
public interface IUserRepository
{
    void Add(User entity);
    List<User> GetAll();
    User Find(Predicate<User> condition);
    User Update(User oldObject, User updatedObject);
}
```

```
public class UserRepository: IUserRepository
{
    private ContextDb _context;

    public UserRepository(ContextDb context)
    {
        _context = context;
    }

    public void Add(User entity)
    {
        _context.Users.Add(entity);
        _context.SaveChanges();
    }

    public List<User> GetAll()
    {
        throw new NotImplementedException();
    }

    public User Find(Predicate<User> condition)
    {
        List<User> adminsitrators = _context.Users.ToList();
        foreach (var user in adminsitrators)
        {
            var condResult = condition(user);
            if (condResult)
                return user;
        }
        throw new KeyNotFoundException();
    }

    private User FindDto(Predicate<User> condition)
    {
        List<User> adminsitrators = _context.Users.ToList();
        foreach (var user in adminsitrators)
        {
            var condResult = condition(user);
            if (condResult)
                return user;
        }
        throw new KeyNotFoundException();
    }


    public User Update(User oldObject, User updatedObject)
    {
        _context.Users.Update(updatedObject);
        _context.SaveChanges();
        return updatedObject;
    }
}
```

5 - Creamos el IUserService

```
public interface IUserService
{
  User AddUser(User user);
```

```
  User GetUsersById(int userId);
  User UpdateUser(User newUser, int userId);
  string Login(string email, string password);
  User GetUserByToken(string toString);
}
```

Creamos el UserService

```
public class UserService : IUserService
    {
        private IUserRepository _repository;
        private IGuidService _guidService;

        public UserService(IUserRepository vRepository, IGuidService vGuid)
        {
            _repository = vRepository;
            _guidService = vGuid;
        }

        public User AddUser(User admin)
        {
             _repository.Add(admin);
             return admin;
        }

        public User GetUsersById(int userId)
        {
            try
            {
                return _repository.Find(x => x.Id == userId);
            }
            catch (KeyNotFoundException)
            {
                throw new KeyNotFoundException();
            }
        }

        public User UpdateUser(User newUser, int userId)
        {
            User oldUser = GetUsersById(userId);
            return _repository.Update(oldUser, newUser);
        }

        public string Login(string email, string password)
        {
            try
            {
                User user = _repository.Find(
                    x => x.Email == email && x.Password == password);
                user.Token = _guidService.NewGuid().ToString();
                UpdateUser(user, user.Id);
                return user.Token;
            }
            catch (KeyNotFoundException)
            {
                throw new KeyNotFoundException();
            }
        }

        public User GetUserByToken(string token)
        {
            try
            {
                return _repository.Find(x => x.Token == token);
```

```
            }
            catch (KeyNotFoundException)
            {
                throw new KeyNotFoundException();
            }
        }


    }
```

6- Aqui usaremos Guid para generar el token

Esta herramienta denominada **GUID** es un tipo **de** datos binario **de** SQL Server **de** 16 bytes que es globalmente único en tablas, bases **de** datos y servidores. El término **GUID** significa Globally Unique Identifier y se usa indistintamente con UNIQUEIDENTIFIER.

```
public interface IGuidService
{
   public Guid NewGuid();
}
```

```
public class GuidService:IGuidService
{
    public Guid NewGuid() { return Guid.NewGuid(); }

}
```

7- Vamos a crear dos models que vamos a usar para el filtro

```
public class ErrorDto
{
    public object Content { get; set; }
    public bool IsSuccess { get; set; }
    public int Code { get; set; }
    public string ErrorMessage { get; set; }
}
```

```
public class LoginDto
{
    public string Email { get; set; }
    public string Password { get; set; }
    public string Token { get; set; }
}
```

8- Creamos los dos nuevos controllers, uno para el login y otro para registrar a los usuarios

```
[ApiController]
[Route("api/Users")]
public class UserController : ControllerBase
{
    private IUserService _userService;

    public UserController(IUserService userService)
    {
        _userService = userService;
    }


    [HttpPost]
    public IActionResult Add([FromBody] User user)
    {
        _userService.AddUser(user);
        return Created("", user);
    }
}
```

```
[ApiController]
[Route("api/Login")]
public class LoginController: ControllerBase
{
    private readonly IUserService _userService;

    public LoginController(IUserService userService)
    {
        _userService = userService;
    }

    [HttpPost]
    public IActionResult Login(LoginDto loginDto)
    {
        loginDto.Token = _userService.Login(loginDto.Email, loginDto.Password);
        return Ok(loginDto);
    }
}
```

8- Creamos el filtro de autenticación

El filtro de autenticación: se utiliza con el fin de autenticar y crear políticas de seguridad para nuestro aplicación web. Se ejecutan antes que cualquier otro filtro y permiten evitar llegar al controller en caso de no cumplir con las politicas de seguridad. Estos estan comprendidos en la interface IAuthorizationFilter

El método OnAuthorization se utiliza para escribir el código para que el filtro pueda autorizar la solicitud.

El parámetro AuthorizationFilterContext context, recibe los datos del contexto que describen la solicitud. Este objeto contiene una propiedad llamada REsult del tipo IActionResult que se utiliza para alterar la respuesta en el caso de ser necesario. Por ejemplo, si no esta autorizado se responde un `401: Unauthorized`. Esto evita que se llame al *controller* y el resto de los filtros.

```
public class FilterAuthentication: Attribute, IAuthorizationFilter
{
    private IUserService? userService;

    public FilterAuthentication(IUserService userService)
    {
        this.userService = userService;
    }

    public void OnAuthorization(AuthorizationFilterContext context)
    {
        this.userService = context.HttpContext.RequestServices.GetService<IUserService>();
        Exception exception = new AuthenticationException("You are not allowed to do this action");
        ErrorDto error=new ErrorDto()
        {
            IsSuccess = false,
            ErrorMessage = exception.Message,
            Content = exception.Message,
            Code = 401
        };

        StringValues token;
        context.HttpContext.Request.Headers.TryGetValue("Authorization", out token);
        if (token.Count == 0 || token == "")
        {
            context.Result = new ObjectResult(error)
            {
                StatusCode = error.Code
            };
        }
        else
        {
            try
            {
                userService.GetUserByToken(token);
            }
            catch (KeyNotFoundException)
            {
                context.Result = new ObjectResult(error)
                {
                    StatusCode = error.Code
                };
            }
        }
    }
}
```

9- Cremoa el filtro de accion

El filtro de Accion: contiene la lógica que se ejecuta antes y después de que se ejecute una acción del controlador. se puede usar un filtro de acción, por ejemplo, para modificar los datos de vista que devuelve una acción del controlador. O como en este caso vamos a ver como usarlo para controlar los roles de los usuarios.

La interfaz **IActionFilter** contiene dos metodos que nos permiten implementar este filtro:

- **OnActionExecuting**: se llama a este método antes de ejecutar una acción del controlador.
- **OnActionExecuted**: se llama a este método después de ejecutar una acción del controlador.

```
public class ProtectFilter : Attribute, IActionFilter
{

    private IUserService? _userService;
    private readonly RoleType _role;
    public ProtectFilter(RoleType role)
    {
        this._role = role;
    }

    public void OnActionExecuted(ActionExecutedContext context) { }

    public void OnActionExecuting(ActionExecutingContext context)
    {
        this._userService = context.HttpContext.RequestServices.GetService<IUserService>();
        StringValues token;
        context.HttpContext.Request.Headers.TryGetValue("Authorization", out token);
        Exception exception = new AuthenticationException("You are not allowed to do this action");
        ErrorDto error=new ErrorDto()
        {
            IsSuccess = false,
            ErrorMessage = exception.Message,
            Content = exception.Message,
            Code = 401
        };
        try
        {
            User user = _userService?.GetUserByToken(token);
            if (user != null && user.Role != _role)
            {
                context.Result = new ObjectResult(error)
                {
                    StatusCode = error.Code
                };
            }
        }
        catch (KeyNotFoundException)
        {
            context.Result = new ObjectResult(error)
            {
                StatusCode = error.Code
            };
        }
    }

}
```

10- Agregamos la inyección de dependencias de las nuevas clases

```
services.AddScoped<IUserRepository, UserRepository>();
services.AddScoped<IUserService, UserService>();
services.AddScoped<IGuidService, GuidService>();
services.AddScoped<FilterAuthentication>();
```

11- Agregamos el filtro a los controller o métodos donde queremos que se aplique

```
[HttpPost]
[ServiceFilter(typeof(FilterAuthentication))]
[ProtectFilter(RoleType.Admin)]
public IActionResult Add([FromBody] Order order)
{
}
```

12- Para probar esto creamos la migracion

```
dotnet ef migrations add migration2 --startup-project "../WebAPI"
dotnet ef database update --startup-project "../WebAPI"
```

13- Agregamos una excepcion en el add order para poder probar el exception filter

```
public void Create(Order order)
    {
        if (order.Name == "")
            throw new FormatException();
        _repository.Add(order);
    }
```

2- Creamos el filtro de excepciones

El filtro de excepciones: Los filtros de excepción son el último tipo de filtro que se va a ejecutar. Puede usar un filtro de excepciones para controlar los errores generados por las acciones del controlador o los resultados de la acción del controlador. También puede usar filtros de excepción para registrar errores.

La interfaz **IExceptionFilter** trae el metodo

- **OnException** el cual se ejecuta al ocurrir un error ocacionado por las funciones llamdas por el controlador

```
public class ExceptionFilter : Attribute, IExceptionFilter
{
     public void OnException(ExceptionContext context)
        {
            List<Type> errors401 = new List<Type>()
            {
            };
            List<Type> errors404 = new List<Type>()
            {
                typeof(KeyNotFoundException)
            };
            List<Type> errors409 = new List<Type>()
            {

            };
            List<Type> errors422 = new List<Type>()
            {
```

```
                typeof(FormatException)
            };

            ErrorDto response = new ErrorDto()
            {
                IsSuccess = false,
                ErrorMessage = context.Exception.Message
            };

            Type errorType = context.Exception.GetType();
            if (errors401.Contains(errorType))
            {
                response.Content = context.Exception.Message;
                response.Code = 401;
            }
            else if (errors404.Contains(errorType))
            {
                response.Content = context.Exception.Message;
                response.Code = 404;
            }
            else if (errors409.Contains(errorType))
            {
                response.Content = context.Exception.Message;
                response.Code = 409;
            }
            else if (errors422.Contains(errorType))
            {
                response.Content = context.Exception.Message;
                response.Code = 422;
            }
            else
            {
                response.Content = context.Exception.Message;
                response.Code = 500;
                Console.WriteLine(context.Exception);
            }

            context.Result = new ObjectResult(response)
            {
                StatusCode = response.Code
            };
        }
    }
```

3- Agregamos el filtor a los controller donde queremos que se aplique

```
[ApiController]
[ExceptionFilter]
[Route("api/Orders")]
public class UserController : ControllerBase
{
}
```