Clase 5 @September 22, 2022

1ra hora

- Repasamos los filtros que hicimos la clase pasada
- Terminamos con filtro de excepciones
- Repaso Mock
- Ventajas de LINQ

2da hora

- Prueba en clase
- Preguntas de obligatorio

1- 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 OrderController : ControllerBase{
}
```

Repaso Mocks

Qué son los mocks?

Son objetos que nos permiten configurar como se comportan los métodos de una clase o librería.
Básicamente le decimos al mock que dado un escenario, se comporta de determinada manera. Ejemplo:

```
mock.Setup(o => o.Create(It.IsAny<Order>())).Returns(order);
```

```
mock.Setup(m => m.Create(null)).Throws(new ArgumentException());
```

Cuándo mockear?

Cuando la unidad de código que queremos testear tiene dependencias externas que no queremos definir en cada test. Básicamente no entran en el SUT (*System under test*).

Por ejemplo nuestro Web Api utiliza la business logic, cada vez que testeamos un método de la api no queremos tener que utilizar una clase de la BusinessLogic y a su vez definir una clase del DataAccess para pasarle al Data Access, dado que no entra en la unidad que queremos testear.
Si no mockearamos los tests, estos pasarían a tender a fallar por cambios externos a los del código que queremos testear. Ejemplo: un cambio en la business logic genera una falla en los tests de la web api.

Qué mockear?

Debemos mockear solo los métodos que se utilizan directamente en la unidad de código que vamos a testear.

Ejemplo:

```
[HttpPost]
0 references
public IActionResult Add([FromBody] Order order)
{
    try
    {
        Order createdOrder = _orderService.Create(order);
        return CreatedAtRoute("AddOrder", new OrderBasicInfoModel()
        {
            Id = createdOrder.Id,
            DeliveryDateTime = createdOrder.DeliveryDateTima
        });
    }
    catch (ArgumentException ex)
    {
        return BadRequest();
    }
}
```

Qué flujos de ejecución salen de este código?
1. El orderService.Create retornar una orden de forma correcta y el Post retorna un `CreatedAtRoute`
2. El orderService.Create lanza una excepción y nos vamos al catch del Post.

Cómo validamos que lo mockeado se esté ejecutando?

VerifyAll → verifica que todo lo mockeado sea llamado

```
mock.VerifyAll();
```

Verify → verifica que un método en particular sea llamado

```
mock.Verify(o => o.Create(It.IsAny<Order>()), Times.Once());
```

Linq

Gracias a LINQ podemos evitar las iteraciones a mano, esto quiere decir que no es necesario hacer uso de foreach, for y while.

Filtrar ordenes por Año y rango de precios

- En la businessLogic van a crear el predicado/expresion de la siguiente manera:

```
//Lo que estamos escribiendo aca es un metodo como cualquier otro, solo que lo estamos asignando a la variable expression.
//El order es el parametro de la funcion, a esta funcion le paso un order de tipo Order
//address y price son una variable/parametro que esta en la funcion que crea esta expresion
//La expresion tiene este formato de que recibe una orden y retorna un booleano porque queremos hacer filtrado
Expression<Func<Order, bool>> expression = order => order.DeliveryDateTime.Year == year && order.price >= priceLowerBound && order.pri
```

- En el dataAccess se va utilizar el metodo .Where(Expression<Func<T, bool>> predicate) de LINQ. Este metodo es el que va a transformar EF Core a una sentencia SQL al proveedor de base de datos SQL Server.

```
IEnumerable<Order> orders = _context.Orders.Where(expression);
```

No sería lo más correcto hacer:

```
public IQueryable<Order> GetFilteredOrders(int year ,int priceLowerBound, int priceUpperBound)
  {
    IList<Order> ordersToReturn = new List<Order>();
```

```
    IQueryable<Order> allOrders = _repository.GetAll();

    foreach(Order order in allOrders)
    {
      if(order.DeliveryDateTime.Year == year && order.price >= priceLowerBound && order.price >= priceUpperBound)
      {
        ordersToReturn.Add(movie);
      }
    }

    return ordersToReturn;
  }
```

Traer solo algunas columnas en especifico

- En BusinessLogic

```
public IQueryable<Order> GetAllNames()
  {
    Expression<Func<Order, string>> expression = order => order.Name;

    IQueryable<Order> orders = _orderRepository.GetAllNames(expression);

    return orders;
  }
```

- En DataAccess

```
public IQueryable<Order> GetAllNames(Expression<Func<Order, string>> expression)
{
  IQueryable<Order> orders = _context.Orders.Select(expression);

  return orders;
}
```

No sería lo más correcto hacer:

```
public IQueryable<Order> GetAllNames()
  {
    IList<Order> ordersToReturn = new List<Order>();
    IEnumerable<Order> orders = _ordersRepository.GetAll();

    foreach(Order order in orders)
    {
      Order orderToReturn = new Order
                            {
                              Name = order.Name
                            };

      ordersToReturn.Add(orderToReturn);
    }

    return ordersToReturn;
  }
```