Tarea - Semana 5

Desarrollo de Software 3

Agustin De Luca

1. Aspecto del Código en los Controladores (Sin MLA)

En un proyecto que carece de una estructura de capas definida, los controladores inevitablemente se convierten en el centro donde convergen múltiples responsabilidades, excediendo ampliamente su función principal de gestionar solicitudes HTTP y coordinar respuestas. Se produciría una mezcla de responsabilidades significativa dentro del código del controlador. Este incluiría lógica de presentación (implícita en la forma en que se devuelven los datos o se crean respuestas como CreatedAtAction), lógica de negocio (como la regla de no borrar Tópicos con Ideas asociadas), y lógica de acceso a datos (interacción directa con Entity Framework Core). Todo esto coexistiría con el manejo inherente de la solicitud y respuesta HTTP.

Esta concentración de tareas llevaría a una notable falta de modularidad y claridad. Los controladores se volverían extensos y densos, haciendo muy difícil discernir dónde termina una responsabilidad y comienza la siguiente. La reutilización de fragmentos de lógica de negocio (como la validación de borrado) o de acceso a datos en otros contextos se volvería impracticable sin duplicar código o crear dependencias no deseadas entre controladores.

Un ejemplo concreto de esto se puede observar en el siguiente TopicController

```
using IdeaVotingSystem.Models;

using IdeaVotingSystem.Dtos;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace IdeaVotingSystem.Controllers;
```

```
[Route("api/topics")]

[ApiController]

public class TopicController : ControllerBase

{

    private readonly AppDbContext _db;



    public TopicController(AppDbContext db)

    {

        _db = db;

    }



    [HttpGet]

    public async Task<ActionResult<IEnumerable<Topic>>> GetAll()

    {

       return await _db.Topics.ToListAsync();

    }



    [HttpGet("{id}")]

    public async Task<ActionResult<Topic>> GetById(int id)

    {

         var topic = await _db.Topics.FindAsync(id);

          return topic != null ? topic : NotFound();
```

```
}

[HttpPost]
public async Task<IActionResult> Create(Topic topic)
{
    topic.CreatedAt = DateTime.UtcNow;
    topic.UpdatedAt = DateTime.UtcNow;

    // Lógica de acceso a datos
    _db.Topics.Add(topic);
    await _db.SaveChangesAsync();

    // Lógica de presentación/respuesta HTTP
    return CreatedAtAction(nameof(GetById), new { id = topic.Id }, new
    {
        topic.Id, // Selección explícita de campos para la respuesta
        topic.Title,
        topic.Description,
        topic.CreatedAt,
        topic.UpdatedAt
    });
}
```

```
    [HttpPut("{id}")]
    public async Task<IActionResult> Update(int id, [FromBody]
UpdateTopicDto updateDto)
    {
        // Lógica de acceso a datos
        var existingTopic = await _db.Topics.FindAsync(id);

        if (existingTopic == null)
        {
            return NotFound(); // Respuesta HTTP
        }
        // Lógica de mapeo/negocio (actualización de campos)
        existingTopic.Title = updateDto.Title;
        existingTopic.Description = updateDto.Description;
        existingTopic.UpdatedAt = DateTime.UtcNow;

        try
        {
            // Lógica de acceso a datos y manejo de errores específicos de
EF Core
            await _db.SaveChangesAsync();
```

```
        }
        catch (DbUpdateConcurrencyException)
        {
            // Más lógica de acceso a datos para verificar existencia
            if (!await _db.Topics.AnyAsync(t => t.Id == id))
            {
                return NotFound();
            }
            else
            {
                throw; // Re-lanzar excepción específica de la capa de
datos
            }
        }
        // Respuesta HTTP
        return NoContent();
    }


    [HttpDelete("{id}")]
    public async Task<IActionResult> Delete(int id)
    {
        // Lógica de acceso a datos
```

```
        var topicToDelete = await _db.Topics.FindAsync(id);



        if (topicToDelete == null)

        {

            return NotFound(); // Respuesta HTTP

        }

        var hasIdeas = await _db.Ideas.AnyAsync(i => i.TopicId == id);

        if (hasIdeas)

        {

            // Respuesta HTTP específica basada en regla de negocio

            return Conflict($"Cannot delete Topic '{topicToDelete.Title}'
(Id: {id}) because it has associated ideas.");

        }



        // Lógica de acceso a datos

        _db.Topics.Remove(topicToDelete);

        await _db.SaveChangesAsync();

        return NoContent();

    }

}
```

Este código muestra claramente cómo el TopicController maneja directamente

AppDbContext para todas las operaciones CRUD, incluye lógica de negocio como la validación en Delete y la asignación de fechas en Create/Update, y construye las respuestas HTTP. Todo está concentrado en una sola clase.

2. Violación de Principios de Código Limpio

La ausencia de una arquitectura en capas, como se evidencia en el TopicController anterior, conduce directamente a la transgresión de principios esenciales de Clean Code, deteriorando la calidad general del software. El Principio de Responsabilidad Única (SRP) se ve claramente violado. El TopicController asume múltiples responsabilidades: gestionar las rutas y solicitudes HTTP, interactuar directamente con la base de datos a través de AppDbContext, implementar reglas de negocio (como la validación de borrado en Delete), mapear entre DTOs y modelos (en Update), y formatear las respuestas HTTP. Una modificación en cualquiera de estas áreas (cambio en la estructura de la base de datos, alteración de una regla de negocio, cambio en el formato de respuesta JSON) requeriría modificar el TopicController.

Además, el Principio de Abierto/Cerrado (OCP), que estipula que el software debe estar abierto a la extensión pero cerrado a la modificación, se vuelve difícil de cumplir. La adición de funcionalidad transversal, como logging detallado antes y después de cada operación de base de datos o la implementación de un sistema de caché para las consultas Get, requeriría modificar cada uno de los métodos (GetAll, GetById, Create, Update, Delete) del controlador. Con una MLA, se podría decorar un repositorio o servicio sin tocar el controlador.

Aunque el Principio de Sustitución de Liskov (LSP) se relaciona más directamente con la herencia, la falta de abstracciones (interfaces) para el acceso a datos o la lógica de negocio dificulta la sustitución. El controlador depende directamente de la clase concreta AppDbContext. Sustituirla por una implementación diferente (quizás una que use Dapper o una base de datos NoSQL, o incluso una versión en memoria para pruebas) requeriría cambiar la dependencia del controlador y probablemente gran parte del código interno de los métodos que usan _db. Si dependiera de una interfaz ITopicRepository, la sustitución sería transparente para el controlador.

El Principio de Inversión de Dependencia (DIP) también se ve comprometido. El TopicController (un módulo de alto nivel que maneja la interacción con el usuario/cliente) depende directamente de AppDbContext (un detalle de bajo nivel

sobre cómo se persisten los datos). DIP postula que ambos deberían depender de abstracciones. En este caso, la dependencia es directa y concreta, creando un fuerte acoplamiento entre la capa de presentación/API y la capa de acceso a datos.

Métodos como Update o Delete mezclan la recuperación de datos, validaciones, lógica de negocio, actualizaciones y manejo de errores específicos de la base de datos, haciéndolos más largos y complejos de entender. A medida que la aplicación crezca, esta mezcla hará que la Mantenibilidad sea Difícil. Corregir un bug en la lógica de acceso a datos o modificar una regla de negocio podría generar efectos secundarios inesperados o bugs.

3. Impacto en la Testabilidad

La falta de separación de responsabilidades y el fuerte acoplamiento presentes en el TopicController convierten las pruebas efectivas en una tarea extremadamente complicada. Las pruebas unitarias se ven particularmente afectadas por la dificultad de aislamiento. Resulta casi imposible probar unitariamente la lógica de negocio dentro del método Delete (la comprobación if (hasIdeas)) sin involucrar una base de datos real o simulada. Con la estructura actual, se requeriría configurar un mock complejo para AppDbContext, DbSet<Topic>, DbSet<Idea> y el comportamiento de métodos como FindAsync y AnyAsync. Esto termina probando más la configuración del mock o la interacción con EF Core que la lógica de negocio en sí misma. Probar la lógica de mapeo en Update también requiere simular primero la recuperación exitosa del existingTopic desde la base de datos.

Las pruebas de integración, por su parte, aunque necesarias, se vuelven más gruesas. Al probar el método Delete, por ejemplo, no solo se prueba la integración del controlador con la base de datos, sino también la regla de negocio incrustada. Se necesita un entorno de pruebas con una base de datos real (o una base de datos en memoria configurada para EF Core) que contenga datos específicos para cubrir los casos de éxito (borrado permitido) y de conflicto (borrado denegado por ideas existentes). Diagnosticar fallos puede ser más complejo al no estar claro si el problema reside en la interacción con la BD o en la lógica de negocio.

En cuanto a las pruebas End-to-End (E2E), aunque por naturaleza evalúan el sistema completo, se vuelven más frágiles. Un cambio interno en cómo se implementa la consulta AnyAsync en el método Delete, aunque funcionalmente equivalente, podría potencialmente romper una prueba E2E si esta dependía de algún efecto secundario

o tiempo de respuesta específico. Más importante aún, cuando una prueba E2E falla (por ejemplo, un DELETE devuelve 500 Internal Server Error en lugar de 204 No Content o 409 Conflict), rastrear la causa raíz es más difícil. El problema podría estar en la deserialización del request, la lógica de búsqueda, la regla de negocio, la operación de borrado en la base de datos, o incluso en la serialización de la respuesta, todo dentro del mismo método del controlador.

4. Problemas al Cambiar la Tecnología de Base de Datos

Intentar cambiar la tecnología de base de datos subyacente en un proyecto estructurado como el del TopicController (por ejemplo, migrar de EF Core con SQL Server a usar Dapper con PostgreSQL, o a MongoDB) representa un desafío monumental, plagado de riesgos y costes elevados. El problema fundamental reside en el código disperso y fuertemente acoplado. La lógica de acceso a datos, implementada usando directamente AppDbContext y métodos específicos de EF Core (ToListAsync, FindAsync, Add, SaveChangesAsync, AnyAsync, Remove), está embebida dentro de cada método del controlador que interactúa con la persistencia.

Esto implica un esfuerzo de refactorización masivo. No se trata solo de cambiar la inyección en el constructor; cada línea de código que usa _db tendría que ser reemplazada por el código equivalente usando la nueva tecnología de acceso a datos. Esto incluye reescribir consultas, cambiar la forma en que se manejan las transacciones o unidades de trabajo (si aplica), y adaptar el manejo de errores (por ejemplo, DbUpdateConcurrencyException es específico de EF Core).

Este proceso de cambio invasivo y distribuido conlleva un alto riesgo de introducir errores. Modificar la lógica de acceso a datos en tantos lugares diferentes aumenta significativamente la probabilidad de cometer errores, introducir regresiones o cambiar sutilmente el comportamiento de la aplicación. Validar que todas las operaciones CRUD y las lógicas asociadas funcionen correctamente después de tal cambio requiere un esfuerzo de pruebas exhaustivo y meticuloso.