Padrões de Desenvolvimento e Boas Práticas ZCustodia - Backend .NET

Sumário

1. Introdução

Este documento foi criado para consolidar os padrões e boas práticas que devem ser seguidos no desenvolvimento de aplicações .NET utilizando C#, com base em princípios sólidos de arquitetura, clareza de código e manutenção a longo prazo. O objetivo é padronizar a forma como controladores, serviços e DTOs são implementados, garantindo qualidade e consistência entre os projetos.

2. Arquitetura e Responsabilidades

A arquitetura adotada segue o padrão Domain-Driven Design (DDD), com divisão clara de responsabilidades entre as camadas: Presentation, Application, Domain e Infrastructure.

Responsabilidades principais:

- • **Controller:** recebe requisições HTTP, chama serviços e retorna respostas padronizadas.
- • **Service:** contém a lógica de negócio e orquestra operações entre repositórios e domínios.
- • **Domain:** define as entidades e regras fundamentais de negócio.
- • **Infrastructure:** lida com persistência, comunicação externa e acesso a dados.

A lógica de negócio **nunca deve estar na Controller**. O foco da Controller é apenas receber e devolver respostas, enquanto as regras ficam encapsuladas na camada de Service, seguindo o princípio de separação de responsabilidades.

3. Padrão de Controller

As Controllers devem utilizar retornos padronizados e lidar com exceções esperadas e inesperadas de forma consistente. Sempre que possível, utilize atributos `[ProducesResponseType]` para documentar os possíveis retornos da ação.

```
[HttpGet("{id}")]
[ProducesResponseType(typeof(TbModulo), (int)HttpStatusCode.OK)]
[ProducesResponseType(typeof(ErroOutputDto), (int)HttpStatusCode.NotFound)]
[ProducesResponseType(typeof(ErroOutputDto), (int)HttpStatusCode.InternalServerError)]
public async Task<IActionResult> ObterModuloPorId([FromRoute, Required] int id)
{
   try
   {
      return Ok(await _moduloService.ObterModuloPorId(id));
   }
   catch (NotFoundException ex)
   {
      return NotFound(new ErroOutputDto(ex));
   }
   catch (Exception ex)
   {
      return StatusCode((int)HttpStatusCode.InternalServerError, new ErroOutputDto(ex));
   }
}
```

4. Padrão de Retorno para Erros

Os erros retornados pelas APIs devem utilizar o objeto `ErroOutputDto`, garantindo estrutura e rastreabilidade. Esse padrão padroniza mensagens de erro tanto para desenvolvedores quanto para o front-end.

```
public class ErroOutputDto
{
   public HttpStatusCode? CodigoStatus { get; set; }
   public string? Mensagem { get; set; }
   public string? StackTrace { get; set; }
   public List<dynamic>? ListaErros { get; set; }
}
```

A propriedade `ListaErros` pode ser utilizada para validar campos específicos em requisições de entrada, por exemplo, erros de validação de modelo.

5. Padrão para GETs Paginados

Quando um endpoint retorna grandes volumes de dados, utilize `PaginadoOutputDto<T>` para estruturar a resposta. Esse padrão facilita a paginação no front-end e garante padronização entre diferentes APIs.

```
public class PaginadoOutputDto<T>
{
   public int TotalRegistros { get; set; }
   public int TotalPaginas { get; set; }
   public int PaginaAtual { get; set; }
   public int TamanhoPagina { get; set; }
   public List<T>? Resultados { get; set; }

   public PaginadoOutputDto() { }

   public PaginadoOutputDto(int totalRegistros, int totalPaginas, int paginaAtual, int
tamanhoPagina, List<T> resultados)
   {
      TotalRegistros = totalRegistros;
      TotalPaginas = totalPaginas;
      PaginaAtual = paginaAtual;
      TamanhoPagina = tamanhoPagina;
      Resultados = resultados;
   }
}
```

6. Boas Práticas de Implementação

- • Utilize injeção de dependência para todos os serviços e repositórios.
- • Nunca acesse diretamente o banco de dados a partir da Controller.
- • Sempre trate exceções conhecidas com retornos adequados (400, 404, 500).
- • Utilize DTOs específicos para entrada e saída (Input/Output DTOs).
- • Utilize AutoMapper para conversões entre entidades e DTOs.

7. Tratamento de Exceções

O tratamento de exceções deve seguir um padrão previsível. Exceções específicas devem ser capturadas e traduzidas em mensagens claras, enquanto exceções genéricas devem ser logadas e retornadas com um status 500.

8. Conclusão

O cumprimento destes padrões garante que o código seja legível, padronizado e fácil de manter. A aplicação de DDD e SOLID contribui diretamente para a escalabilidade e qualidade da solução.