Tópicos Abordados

- Swagger Annontation
- Hate Limit
- Compressão de Dados
- Hateaos
- Repository Pattern
- Utilizando Dtos
- Criando Mapeamentos

Swagger Annontation

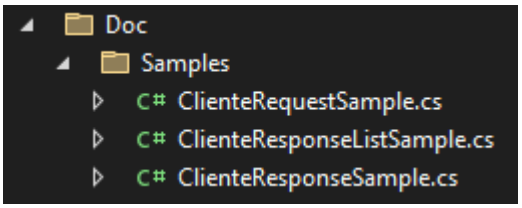
Adicionando Pacote de Exemplos

```
dotnet add package Swashbuckle.AspNetCore.Filters
```

Configurando a `Program.cs`

```
using Swashbuckle.AspNetCore.Filters;

builder.Services.AddSwaggerGen(c =>
{
    c.EnableAnnotations();
    c.ExampleFilters();    // habilita os exemplos do pacote
});

// registra as classes de exemplo
builder.Services.AddSwaggerExamplesFromAssemblyOf<Program>();

var app = builder.Build();

app.UseSwagger();
app.UseSwaggerUI();
```

Criando Diretório



Criando as classes com exemplos

```
public class ClienteRequestSample : IExamplesProvider<ClienteEntity>
{
    public ClienteEntity GetExamples()
    {
        return new ClienteEntity
        {
            Id = 6,
            Nome = "Joao Silta",
            Email = "joao.silva@outlook.com",
            EstadoId = 5,
        };
    }
}
```

```
public class ClienteResponseSample : IExamplesProvider<ClienteEntity>

{
    public ClienteEntity GetExamples()
    {
        return new ClienteEntity
        {
            Id = 1,
            Nome = "Marcos Oliveira",
            Email = "marcos.oliveira@outlook.com",
            EstadoId = 5,
        };
    }
}
```

```
public class ClienteResponseListSample :
IExamplesProvider<IEnumerable<ClienteEntity>>
{
    public IEnumerable<ClienteEntity> GetExamples()
    {
        return new List<ClienteEntity>
        {
            new ClienteEntity {
                Id = 1,
                Nome = "Marcos Oliveira",
                Email = "marcos.oliveira@outlook.com",
                EstadoId = 5,
            },
            new ClienteEntity {
                Id = 2,
                Nome = "Alex Ferreira",
                Email = "alex.ferreira@outlook.com",
                EstadoId = 5,
```

```
            }
        };
    }
}
```

Implementando nos métodos da controller

```
[HttpGet]
[SwaggerOperation(
   Summary = "Lista clientes",
   Description = "Retorna a lista completa de clientes cadastrados."
)]
[SwaggerResponse(statusCode: 200, description: "Lista retornada com
sucesso", type: typeof(IEnumerable<ClienteEntity>))]
[SwaggerResponse(statusCode: 201, description: "Não possui dados para o
cliente")]
[SwaggerResponseExample(statusCode: 200,
typeof(ClienteResponseListSample))]
public async Task<IActionResult> Get()

{
    ...
}
```

```
[HttpPost]
[SwaggerRequestExample(typeof(ClienteEntity),
typeof(ClienteRequestSample))]
[SwaggerResponse(statusCode: 200, description: "Cliente salvo como
sucesso", typeof(ClienteEntity))]
[SwaggerResponseExample(statusCode: 200, typeof(ClienteResponseSample))]
public IActionResult Post(ClienteDto entity)

{
    ....
}
```

Hate Limit

Usar **Rate Limit em uma API ASP.NET** é essencial para **segurança, estabilidade e controle de uso**, evitando abusos, protegendo o servidor e garantindo que todos os usuários tenham acesso justo aos recursos.

Criando as policies

Vamos adicionar a namespace `System.Threading.RateLimiting` no arquivo de configuração Program.cs para ciar as policies e limitar a quantidade de acessos na api.

Adicione as configurações do RateLimiter:

```
using System.Threading.RateLimiting;

builder.Services.AddRateLimiter(options => {
    options.AddFixedWindowLimiter(policyName: "rateLimitePolicy", opt => {
        opt.PermitLimit = 5;
        opt.Window = TimeSpan.FromSeconds(10);
        opt.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        opt.QueueLimit = 2;
    });

    options.AddFixedWindowLimiter(policyName: "rateLimitePolicy2", opt =>
{
        opt.PermitLimit = 3;
        opt.Window = TimeSpan.FromSeconds(5);
        opt.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        opt.QueueLimit = 2;
    });
    options.RejectionStatusCode = StatusCodes.Status429TooManyRequests;
});
```

> **Obs:** É possível criar mais de uma politica de Rate Limiter

O que ele esta fazendo

- O cliente só pode fazer **5 requisições a cada 10 segundos**.
- Se ultrapassar, recebe **429 Too Many Requests**.

Habilitando na aplicação

Adicione na Program.cs a seguinte linha para habilitar o Rate Limmiter

```
app.UseRateLimiter();
```

Exemplo:

```
var app = builder.Build();


if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseAuthorization();
```

```
app.UseRateLimiter(); //Habilitando Rate Limiter

app.MapControllers();

app.Run();
```

Configurando controller

Vamos adicionar a annotation para habilitar o Rate Limiter no metodo Get da API

```
using Microsoft.AspNetCore.RateLimiting;

[HttpGet]
[EnableRateLimiting("rateLimitePolicy")] // Incluindo a Plolicy de Rate
Limit
public async Task<IActionResult> Get()
{

}
```

Compressão de Dados

A compressão de respostas em uma API tem como objetivo reduzir o tamanho dos dados enviados do servidor para o cliente. Isso traz diversos benefícios:

1. **Redução de tráfego na rede**: respostas menores significam menos consumo de banda.
2. **Melhoria no tempo de resposta percebido**: clientes recebem os dados mais rapidamente.
3. **Escalabilidade**: menos dados trafegando permite que mais requisições sejam atendidas sem aumentar significativamente o uso de recursos.
4. **Compatibilidade com navegadores e clientes modernos**: Gzip e Brotli são amplamente suportados.
5. **Custo-benefício**: troca-se processamento do servidor (para comprimir) por economia de largura de banda e melhor experiência do usuário.

Configurando Compressão

Adicione a namespace `Microsoft.AspNetCore.ResponseCompression` no arquivo Program.cs, para adicionar os pacotes de compressão de resposta.

```
using Microsoft.AspNetCore.ResponseCompression;
```

Adicionando configuração

Ainda no arquivo Program.cs adicione as seguinte configurações:

```
builder.Services.AddResponseCompression(options => {
    //options.EnableForHttps = true;
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
});
```

Configurando tipo de compressão

Cada algoritmo de compressão (Brotli e Gzip) pode operar em diferentes **níveis de intensidade**, que impactam em duas dimensões:

- **Tamanho final da resposta** (quanto menor, melhor para o cliente).
- **Uso de CPU do servidor** (quanto maior a compressão, mais processamento é necessário).

```
builder.Services.Configure<BrotliCompressionProviderOptions>(options =>
{
    options.Level = System.IO.Compression.CompressionLevel.Fastest;
});

builder.Services.Configure<GzipCompressionProviderOptions>(options => {
    options.Level = System.IO.Compression.CompressionLevel.Fastest;
});
```

No caso acima, foi configurado o nível `Fastest`, que prioriza **velocidade de resposta** em vez de máxima compactação. Isso é importante em cenários de API porque:

1. **APIs respondem muitas vezes com payloads pequenos/medianos**, onde a diferença entre máxima e rápida compressão é mínima em bytes, mas o custo em CPU pode ser significativo.
2. **Escalabilidade**: reduzir o uso de CPU permite que o servidor atenda muito mais requisições simultâneas.
3. **Equilíbrio ideal**: em geral, `Fastest` gera economia de rede suficiente sem comprometer o desempenho do servidor.

Habilitando a compressão

Adicione na Program.cs as seguinte propriedade `app.UseResponseCompression()`

```
var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

```
app.UseAuthorization();
app.UseResponseCompression();

app.MapControllers();

app.Run();
```

Hateaos

HATEOAS (Hypermedia as the Engine of Application State) é um princípio do estilo arquitetural REST que define que as respostas da API devem conter não apenas os dados solicitados, mas também links de navegação que orientam o cliente sobre as ações disponíveis para aquele recurso. Dessa forma, ao buscar um cliente, por exemplo, a resposta pode incluir links para visualizar, atualizar ou excluir o registro, tornando a API autoexplicativa e permitindo que o consumidor descubra endpoints por meio da própria navegação. Esse recurso reduz a dependência de documentação externa, facilita a evolução da API sem quebrar clientes existentes e torna a interação mais flexível, navegável e auto-documentada.

Formatando a saída com Hateos

No método **Get** que utilizamos para retornar todos os clientes vamos formatar para orientar o usuário com a navegação dos links

```
[HttpGet]
public async Task<IActionResult> Get()
{
    var clientes = await _context.Cliente.ToListAsync();

    if(!result.Any())
        return NoContent();

    var Id = result.FirstOrDefault()?.Id ?? 0;

    var hateoas = new
    {
        data = result,
        links = new {
           self = Url.Action(nameof(Get), "Cliente", null),
           getById = Url.Action(nameof(Get), "Cliente", new { id = Id}),
           put = Url.Action(nameof(Put), "Cliente", new { id = Id }),
           delete = Url.Action(nameof(Delete), "Cliente", new { id = Id
}),
        }
    };
```

```
    return Ok(hateoas);
}
```

Aplicando em cada registro o Hateaos

Podemos também orientar o usuário com uma navegação mais em cada registro, formatando a saída e incluindo os links.

```
[HttpGet]
public async Task<IActionResult> Get()
{
    var clientes = await _context.Cliente.ToListAsync();

    if(!result.Any())
        return NoContent();

    var hateoas = new
    {
        data = result.Select(c => new
        {
            c.Id,
            c.Nome,
            c.Email,
            links = new
            {
                self = Url.Action(nameof(Get), "Cliente", new { id = c.Id
}, Request.Scheme),
                update = Url.Action(nameof(Put), "Cliente", new { id =
c.Id }, Request.Scheme),
                delete = Url.Action(nameof(Delete), "Cliente", new { id =
c.Id }, Request.Scheme)
            }
        }),
        links = new
        {
            self = Url.Action(nameof(Get), "Cliente", null,
Request.Scheme),
            create = Url.Action(nameof(Post), "Cliente", null,
Request.Scheme)
        }
    };

    return Ok(hateoas);
}
```

Adicionando o `Request.Scheme` temos uma saída formatada com os links formatados

```
{
  "data": [
    {
      "id": 1,
      "nome": "João",
      "email": "joao@email.com",
      "links": {
        "self": "https://localhost:5001/api/Cliente/1",
        "update": "https://localhost:5001/api/Cliente/1",
        "delete": "https://localhost:5001/api/Cliente/1"
      }
    },
    {
      "id": 2,
      "nome": "Maria",
      "email": "maria@email.com",
      "links": {
        "self": "https://localhost:5001/api/Cliente/2",
        "update": "https://localhost:5001/api/Cliente/2",
        "delete": "https://localhost:5001/api/Cliente/2"
      }
    }
  ],
  "links": {
    "self": "https://localhost:5001/api/Cliente",
    "create": "https://localhost:5001/api/Cliente"
  }
}
```


Repository Pattern

É um padrão de projeto que **isola a lógica de acesso a dados** (queries SQL, Entity Framework, Dapper, MongoDB etc.) da lógica de negócios da aplicação.
Ele atua como um **intermediário** entre o **Domain/Application** e a **fonte de dados**.

Benefícios de Usar Repository Pattern

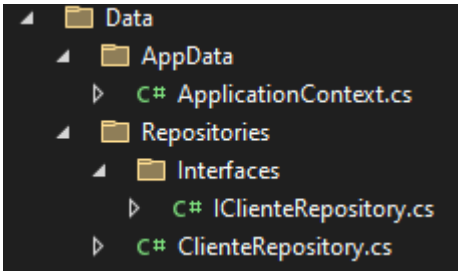
1. **Separação de responsabilidades (SRP - Single Responsibility Principle)**
   - A camada de aplicação/serviço não precisa conhecer como os dados são persistidos.
   - O repositório concentra tudo relacionado a persistência.
2. **Facilidade de manutenção**
   - Se amanhã você trocar o banco de dados (ex.: de SQL Server para MongoDB), você só altera a camada de repositórios, sem mudar a lógica de negócio.
3. **Testabilidade**
   - Você pode **mockar** os repositórios em testes unitários sem precisar de banco real.

- Isso facilita TDD.

4. **Reutilização de código**
   - Centraliza operações comuns como `Add`, `Update`, `GetById`, `Remove`, evitando duplicação.
5. **Camada de abstração**
   - Evita que o Entity Framework (ou outro ORM) fique espalhado pelo código da aplicação.
   - Se amanhã você quiser trocar EF por Dapper, o impacto será mínimo.
6. **Organização e clareza**
   - Deixa claro quais operações estão disponíveis sobre uma entidade.
   - Facilita a leitura e a manutenção por equipes grandes.

Organizando o Diretório

Adicionando novo diretório dentro da pasta Data, para armazenar os arquivos do repositório.



Criando a Interface

```
public interface IClienteRepository
{
    Task<IEnumerable<ClienteEntity>> ObterTodosAsync();
    Task<ClienteEntity?> ObterUmAsync(int Id);
    Task<ClienteEntity?> AdicionarAsync(ClienteEntity entity);
    Task<ClienteEntity?> EditarAsync(int Id, ClienteEntity entity);
    Task<ClienteEntity?> DeletarAsync(int Id);
}
```

Criando arquivo concreto do Repositorio

```
public class ClienteRepository : IClienteRepository
{
    public Task<ClienteEntity?> AdicionarAsync(ClienteEntity entity)
    {
        throw new NotImplementedException();
    }

    public Task<ClienteEntity?> DeletarAsync(int Id)
```

```
    {
        throw new NotImplementedException();
    }

    public Task<ClienteEntity?> EditarAsync(int Id, ClienteEntity entity)
    {
        throw new NotImplementedException();
    }

    public Task<IEnumerable<ClienteEntity>> ObterTodosAsync()
    {
        throw new NotImplementedException();
    }

    public Task<ClienteEntity?> ObterUmAsync(int Id)
    {
        throw new NotImplementedException();
    }
}
```

Configurando Context

```
public class ClienteRepository : IClienteRepository
{
    private readonly ApplicationContext _context;

    public ClienteRepository(ApplicationContext context)
    {
        _context = context;
    }
}
```

Refatorando os métodos

Vamos trazer a lógica de banco de dados que esta no controller para o nosso arquivo ClienteRepository.cs

```
public async Task<ClienteEntity?> AdicionarAsync(ClienteEntity entity)
{
    _context.Cliente.Add(entity);
    _context.SaveChanges();

    return entity;
}

public async Task<ClienteEntity?> DeletarAsync(int Id)
{
```

```
    var result = await _context.Cliente.FindAsync(Id);

    if (result is not null)
    {
        _context.Remove(result);
        _context.SaveChanges();

        return result;
    }
    return null;
}



public async Task<ClienteEntity?> EditarAsync(int Id, ClienteEntity
entity)
{
    var result = await _context.Cliente.FindAsync(Id);

    if (result is not null)
    {
        result.Nome = entity.Nome;
        result.Email = entity.Email;
        result.EstadoId = entity.EstadoId;

        _context.Update(result);
        _context.SaveChanges();

        return result;
    }
    return null;
}



public async Task<IEnumerable<ClienteEntity>> ObterTodosAsync()
{
    var result = await _context.Cliente.ToListAsync();
    return result;
}

public async Task<ClienteEntity?> ObterUmAsync(int Id)
{
    var result = await _context.Cliente.FindAsync(Id);
    return result;
}
```

Refatorando a controller

Configurando o Repository

```
public class ClienteController : ControllerBase
{
    private readonly IClienteRepository _clienteRepository;

    public ClienteController(IClienteRepository clienteRepository)
    {
        _clienteRepository = clienteRepository;
    }
}
```

Usando Repository nos métodos

- Buscar todos os clientes

```
[HttpGet]

[SwaggerOperation(
   Summary = "Lista clientes",
   Description = "Retorna a lista completa de clientes cadastrados."
)]

[SwaggerResponse(statusCode: 200, description: "Lista retornada com
sucesso", type: typeof(IEnumerable<ClienteEntity>))]
[SwaggerResponse(statusCode: 201, description: "Não possui dados para o
cliente")]
[EnableRateLimiting("rateLimitePolicy")]
public async Task<IActionResult> Get()
{
    var result = await _clienteRepository.ObterTodosAsync();

    if(!result.Any())
        return NoContent();

    var Id = result.FirstOrDefault()?.Id ?? 0;

    var hateos = new
    {
        data = result,
        links = new {
            self = Url.Action(nameof(Get), "Cliente", null),
            getById = Url.Action(nameof(Get), "Cliente", new { id = Id}),
            put = Url.Action(nameof(Put), "Cliente", new { id = Id }),
            delete = Url.Action(nameof(Delete), "Cliente", new { id = Id
}),
        }
    };
```

```
    return Ok(hateoas);
}
```

- Obter um cliente

```
[HttpGet("{id}")]
[SwaggerOperation(
    Summary = "Obtém cliente por ID",
    Description = "Retorna o cliente correspondente ao ID informado."
)]
[SwaggerResponse(statusCode: 200, description: "Cliente encontrado", type:
typeof(ClienteEntity))]
[SwaggerResponse(statusCode: 404, description: "Cliente não encontrado")]
public async Task<IActionResult> Get(int id)
{
    var result = await _clienteRepository.ObterUmAsync(id);

    if (result is null)
        return NotFound();

    var hateaos = new
    {
        data = result,
        links = new
        {
            self = Url.Action(nameof(Get), "Cliente", new { id }),
            get = Url.Action(nameof(Get), "Cliente", null),
            put = Url.Action(nameof(Put), "Cliente", new { id }),
            delete = Url.Action(nameof(Delete), "Cliente", new { id }),
        }
    };

    return Ok(hateos);
}
```

Configurando a DI (Injeção de Dependência)

```
builder.Services.AddTransient<IClienteRepository, ClienteRepository>();
```

Dtos

DTOs (**Data Transfer Objects**) são usados em aplicações, especialmente em **APIs ASP.NET** com boas práticas de arquitetura (ex: **Clean Architecture**, **DDD**, etc.), para **transportar dados entre camadas** de forma organizada, segura e desacoplada.

Quais as vantagens

1. Separação de Camadas

- As entidades de domínio (ex: `ClienteEntity`) representam **a regra de negócio**.
- As DTOs representam **os dados que viajam pela API**.
  Isso evita expor diretamente suas entidades internas, protegendo o domínio.

2. Segurança

- Sem DTOs, a API poderia expor campos sensíveis (ex: `SenhaHash`, `TokenInterno`) ao serializar uma entidade.
- Com DTOs, você controla **exatamente o que vai para o cliente**.

3. Controle de Entrada e Validação

- DTOs servem para definir **o que a API espera receber**.
- Exemplo: no `ClienteCreateDto`, você pode pedir só `Nome` e `Email`, sem permitir que o usuário mande `Id` ou `DataCriacao`.

4. Facilidade de Evolução

- DTOs permitem **versões diferentes** da API sem alterar o domínio.
  Exemplo: `ClienteDtoV1` e `ClienteDtoV2`.

5. Performance

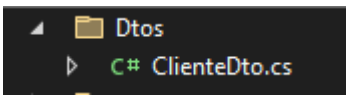
- DTOs podem reduzir payloads, transmitindo **apenas os dados necessários**.
- Isso evita trafegar objetos grandes com dados inúteis.

6. Padronização e Clareza

- Quando outra equipe consome sua API, a documentação fica clara:
  “Esse é o contrato, esses são os dados de entrada e saída.”

Organizando Diretório

Crie uma nova pasta no projeto com nome Dtos para armazenar os arquivos de Dtos



Criando a DTO

Crie o arquivo `ClienteDto.cs` para ser utilizado nos metodos Post e Put da Controller `ClienteController.cs`.

Vamos utilizar o `Record`, que é uma forma de declarar classes imutáveis e com **igualdade por valor** (value-based equality). Ele foi pensado para facilitar a criação de **objetos de**

domínio, **DTOs** e estruturas de dados onde a comparação deve ser feita pelo **conteúdo** e não pela referência.

```
public record ClienteDto(string Nome, string Email, int EstadoId);
```

Utilizando nos metodos de entrada

Altere o parametro de entrada para utilizar a `ClienteDto.cs`.

De:

```
[HttpPost]
public IActionResult Post(ClienteEntity entity)
{
    ....
}
```

Para:

```
[HttpPost]
public IActionResult Post(ClienteDto entity)
{
    ....
}
```

Mappers

Criar uma **classe Mapper** para converter de **DTO → Entidade** (e vice-versa) é uma **boa prática de arquitetura**.

Por que usar Mapper?

1. **Separação de responsabilidades (SRP - SOLID)**
   - O DTO (Data Transfer Object) existe para transportar dados entre a API e o cliente.
   - A Entidade representa a regra de negócio no domínio.
   - Se misturarmos os dois, a camada de domínio pode acabar dependendo de detalhes da API.
   - O Mapper centraliza a conversão e mantém cada camada focada na sua responsabilidade.
2. **Evita código duplicado**
   - Sem mapper, cada controller ou service teria que repetir código para converter DTO em entidade.
   - Com mapper, centralizamos a lógica de transformação em um só lugar.
3. **Maior manutenibilidade**

- Se mudar o DTO ou a Entidade, você ajusta apenas no Mapper.
- Sem isso, teria que procurar e alterar em todos os pontos do sistema onde há conversão.

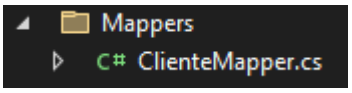
4. **Facilita testes**
   - Você consegue testar a lógica de mapeamento separadamente.
   - Garante que sempre haverá consistência ao transformar dados.
5. **Clareza no fluxo de dados**
   - Fica explícito quando você está lidando com dados de entrada/saída (DTO) e quando está lidando com objetos do domínio (Entidade).

Usando **extension methods como mapper**, utilizamos dois princípios do SOLID:

1. **SRP** (Single Responsibility Principle) → separa responsabilidades.
2. **OCP** (Open/Closed Principle) → estende classes sem modificá-las.

Configurando o Diretório

Cria um novo diretório para armazenar os arquivos de mapper do projeto.



Criando o arquivo `ClienteMapper.cs` e vamos utilizar o conceito de `Extension Methods` para extender o comportamento da classe `ClienteDto`

```
public static class ClienteMapper
{
    public static ClienteEntity ToClienteEntity(this ClienteDto obj)
    {
        return new ClienteEntity {
            Nome = obj.Nome,
            Email = obj.Email,
            EstadoId = obj.EstadoId,
        };
    }
}
```

Aplicando na controller POST e PUT

O repositório espera uma classe `ClienteEntity` no parâmetro de entrada, utilizando a método de extensão, vamos converter o tipo de `ClienteEntity` para `ClienteDto`.

- Método Post

```
[HttpPost]
public IActionResult Post(ClienteDto entity)
```

```
{
    try
    {
        var result
=_clienteRepository.AdicionarAsync(entity.ToClienteEntity());

        return Ok(result);
    }
    catch (Exception)
    {
        return BadRequest();
    }
}
```

- Método Put

```
[HttpPut("{id}")]
public async Task<IActionResult> Put(int id, ClienteDto entity)
{
    var result = await _clienteRepository.EditarAsync(id,
entity.ToClienteEntity());

    if (result is null)
        return NotFound();

    return Ok(entity);
}
```