Tópicos Abordados

- Paginação de Resultados
- Relacionamento
    - Um para Um

Paginação de Resultados

A **paginação de resultados em uma API ASP.NET (ou qualquer API)** é **fundamental** tanto para a performance do servidor quanto para a experiência do cliente que consome a API. Vou explicar os principais pontos:

Motivos para usar paginação em APIs

1. **Performance do servidor**
    - Sem paginação, uma consulta que retorna milhares de registros vai consumir muita memória, CPU e tempo de processamento.
    - Isso pode sobrecarregar o **Entity Framework** e até travar a aplicação em cenários de alta concorrência.
2. **Tempo de resposta**
    - APIs que retornam dados massivos ficam lentas e podem até estourar o **timeout** do cliente.
    - Com paginação, você retorna apenas um subconjunto (ex.: 20 itens), garantindo respostas rápidas.
3. **Redução de tráfego de rede**
    - Se sua API mandar **10.000 registros em JSON**, o payload pode ter vários MB.
    - Isso prejudica dispositivos móveis e conexões lentas.
    - Paginar mantém o **payload leve e eficiente**.
4. **Melhor experiência para o cliente (frontend)**
    - Frontends (React, Angular, Vue, Mobile, etc.) conseguem carregar dados aos poucos (scroll infinito, botões "Próximo/Anterior").
    - Isso melhora a usabilidade e evita "travamentos" de UI.
5. **Escalabilidade**
    - Com paginação, sua API suporta mais usuários concorrentes.
    - Sem ela, alguns poucos requests pesados podem derrubar o sistema.
6. **Controle e ordenação**
    - Paginação geralmente vem junto com **ordenação e filtros**.
    - Isso permite ao cliente **navegar e consultar dados de forma organizada**.
7. **Compatibilidade com padrões de mercado**

- Quase todas as APIs públicas (GitHub, Twitter, Google) usam paginação (`?page=1&limit=20`).
- Isso se tornou um **padrão esperado** para desenvolvedores.

Tipos de paginação

- Offset-Based Pagination
  Pula posições com skip (offset)
- Cursor-Based-Pagination
  É mais usando para paginações infinitas

Offset Based Pagination

Criando modelo na pasta Models, que servirá para formatar a saída de dados do nosso modelo.
Ele esta usando generics `<T>`, para assumir o tipo do objeto que for passado

```
public class PageResultModel<T>
{
    public required T Data { get; set; }

    public int Deslocamento { get; set; }
    public int RegistroRetornado { get; set; }
    public int TotalRegistros { get; set; }
}
```

Modificando a interface

```
 Task<PageResultModel<IEnumerable<ClienteEntity>>> ObterTodosAsync(int
Deslocamento = 0, int RegistroRetornado = 3);
```

Alterando o método do repositório para buscar as iformações

```
public async Task<PageResultModel<IEnumerable<ClienteEntity>>>
ObterTodosAsync(int Deslocamento = 0, int RegistroRetornado = 3)
{
    if (Deslocamento < 0) Deslocamento = 0;
    if (RegistroRetornado <= 0) RegistroRetornado = 3;

    var totalRegistros = await _context.Cliente.CountAsync();

    var result = await _context
        .Cliente
        .OrderBy(x => x.Id)
        .Skip(Deslocamento)
        .Take(RegistroRetornado)
```

```
        .ToListAsync();

    return new PageResultModel<IEnumerable<ClienteEntity>> {
        Data = result,
        Deslocamento = Deslocamento,
        RegistroRetornado = RegistroRetornado,
        TotalRegistros = totalRegistros
    };
}
```

Ajustando a controller para receber os parmetros de Deslocamento e Registros Retornados

```
public async Task<IActionResult> Get(int Deslocamento, int
RegistroRetornado)
{
    var result = await _clienteRepository.ObterTodosAsync(Deslocamento,
RegistroRetornado);

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

    ........
}
```

Relacionamento

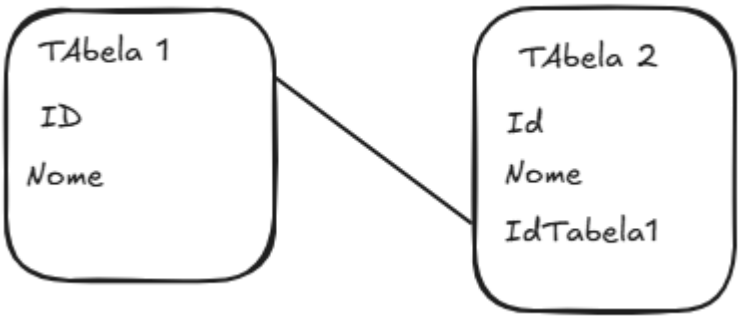
Um para Um (1:1)

Um **relacionamento 1 para 1 (One-to-One)** no **Entity Framework** é quando **uma entidade está associada a no máximo uma outra entidade**. Ou seja, cada registro em uma tabela só pode ter **um** registro relacionado em outra tabela, e vice-versa.

Exemplo do mundo real:

- Uma **Pessoa** pode ter **apenas um Passaporte**.
- Um **Cliente** pode ter **apenas um Endereço de Cobrança**.

No **banco de dados**, isso geralmente é representado com uma **chave primária compartilhada** ou com uma **chave estrangeira exclusiva**.



Modificar a Endidade Cliente

```
[Table("tb_cliente")]
public class ClienteEntity
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Campo nome é obrigatorio")]
    [StringLength(150, ErrorMessage = "Campo nao pode ter mais que 150
caracteres")]
    public string Nome { get; set; } = string.Empty;
    public string Email { get; set; } = string.Empty;

    //Modificar o campo EstadoId por Estado

    public EstadoId { get; set; }

    //Por

    public EstadoEntity Estado { get; set; }
}
```

Adicionar os campos na tabela estado

A notação `[System.Text.Json.Serialization.JsonIgnore]` é utilizada para ignora o campo de relacionamento no retorno dos dados, evitando falhar na serialização.

```
[ForeignKey(nameof(ClienteEntity))]
public int ClienteId { get; set; }

[System.Text.Json.Serialization.JsonIgnore]
public ClienteEntity Cliente { get; set; }
```

Endidade Estado com JsonIgnore

```
[Table("tb_estado")]
public class EstadoEntity
{
    [Key]
    public int Id { get; set; }
    public string Nome { get; set; } = string.Empty;

    [ForeignKey(nameof(ClienteEntity))]
    public int ClienteId { get; set; }

    [System.Text.Json.Serialization.JsonIgnore] //Ignora a propriedade no
bind
    public ClienteEntity Cliente { get; set; }

}
```

Ajustando o mapeamento para receber o EstatoEntity

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

Modificando DTO de entrada

Alterando a coluna do DTO Estado de `Int` para `String`

```
public record ClienteDto(string Nome, string Email, string EstadoName);
```

Ajustando o método de listagem de dados

Para buscar os dados deve incluir `.Include(x => x.Estado)` para buscar os dados relacionados

```
public async Task<PageResultModel<IEnumerable<ClienteEntity>>>
ObterTodosAsync(int Deslocamento = 0, int RegistroRetornado = 3)
```

```
{
    if (Deslocamento < 0) Deslocamento = 0;
    if (RegistroRetornado <= 0) RegistroRetornado = 3;

    var totalRegistros = await _context.Cliente.CountAsync();

    var result = await _context
        .Cliente
        .Include(x => x.Estado) //Busca os relacionamentos
        .OrderBy(x => x.Id)
        .Skip(Deslocamento)
        .Take(RegistroRetornado)
        .ToListAsync();

    ....
}
```

Ajustando o método de Editar

Para salvar as informações de estado usando o relacionamento 1:1 é necessário adicionar uma validação no campo Estado para verificar se ele não esta nulo

```
public async Task<ClienteEntity?> EditarAsync(int Id, ClienteEntity
entity)
{
    var result = await _context
        .Cliente
        .Include(x => x.Estado)
        .FirstOrDefaultAsync(x => x.Id == Id);

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

        if (result.Estado is null)
        {
            result.Estado = new EstadoEntity
            {
                Nome = entity.Estado.Nome,
            };
        }
        else
        {
            result.Estado.Nome = entity.Estado.Nome;
        }

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

```
            return result;
        }

        return null;
    }
```