



## Aula 06 - Operações CRUD com o Entity Framework

Agora que já programamos os comandos de mapeamento e criação das tabelas no banco de dados, podemos programar a *controller* PersonagensController para que os métodos salvem os dados enviados (via client de API como o postman, o Swagger ou Talend API) nas tabelas criadas na base de dados. Quem viabilizará esta operação é a classe DataContext que ao realizar o mapeamento das classes para as tabelas do banco permitirá ações diretamente no banco de dados.

1. Criação da classe **PersonagensController**, dentro da pasta *Controllers*

```
using Microsoft.AspNetCore.Mvc;

namespace RpgApi.Controllers
{
    [ApiController]
    [Route("[Controller]")]
    // 0 references
    public class PersonagensController : ControllerBase
    {
        //Programação seguinte será aqui
    } //Fim da classe do tipo controller
}
```

2. Dentro da classe *controller* crie um atributo global que será do tipo *DataContext* com o nome de **\_context** (1), ele exigirá o using de *RpgApi.Data*. Crie também o construtor para inicializar o atributo **\_context**, que receberá os dados via parâmetro chamado de *context* (2)

```
public class PersonagensController : ControllerBase
{
    //Programação de toda a classe ficará aqui abaixo
    // 1 reference
    1 private readonly DataContext _context; //Declaração do atributo

    // 0 references
    2 public PersonagensController(DataContext context)
    {
        //Inicialização do atributo a partir de um parâmetro
        _context = context;
    }
} //Fim da classe do tipo Controller
```

- Trecho (2): Construtor → Tem o mesmo nome da classe e é executado quando a classe é executada em memória. O parâmetro *context* virá criado da classe *Program.cs* com o caminho do banco. Chamamos esse conceito de injeção de dependência.



3. Crie o método de nome **GetSingle** seja feita uma busca no contexto do Banco de Dados para retornar um personagem de acordo com o Id. Usings necessários: System.Threading.Tasks;RpgApi.Models; Microsoft.EntityFrameworkCore e System

```
[HttpGet("{id}")]
0 references
public async Task<IActionResult> GetSingle(int id)
{
    try
    {
        Personagem p = await _context.Personagens
            .FirstOrDefaultAsync(pBusca => pBusca.Id == id);
        return Ok(p);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

- Observe que temos um bloco try/catch onde tudo que tem que ser programado ficará dentro do try e se algum erro acontecer o Catch vai capturar a mensagem dele e devolver ao requisitante.
4. Crie um método de rota GetAll chamado Get. Exigirá o using System.Collections.Generic

```
[HttpGet("GetAll")]
0 references
public async Task<IActionResult> Get()
{
    try
    {
        List<Personagem> lista = await _context.Personagens.ToListAsync();
        return Ok(lista);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



5. Criação do método do tipo Post chama de **Add**. O comando *SaveChangesAsync* confirmará a inserção dos dados. Temos também uma validação dos pontos de vida que lança uma exceção. Observe que no retorno do método estamos exibindo Id que o banco de dados atribuirá ao personagem

```
[HttpPost]
0 references
public async Task<IActionResult> Add(Personagem novoPersonagem)
{
    try
    {
        if (novoPersonagem.PontosVida > 100)
        {
            throw new Exception("Pontos de Vida não pode ser maior que 100");
        }
        await _context.Personagens.AddAsync(novoPersonagem);
        await _context.SaveChangesAsync();

        return Ok(novoPersonagem.Id);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

6. Crie um método do tipo Put com o nome de Update. Observe que guardamos a quantidade de linhas afetadas para retornar no resultado da requisição.

```
[HttpPut]
0 references
public async Task<IActionResult> Update(Personagem novoPersonagem)
{
    try
    {
        if (novoPersonagem.PontosVida > 100)
        {
            throw new Exception("Pontos de Vida não pode ser maior que 100");
        }
        _context.Personagens.Update(novoPersonagem);
        int linhasAfetadas = await _context.SaveChangesAsync();

        return Ok(linhasAfetadas);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



7. Crie um método do tipo Delete com o nome **Delete**. Observe que como só temos o id sendo passado, primeiro vamos na base encontrar o objeto que desejamos remover e passamos este objeto para o contexto excluir. Após isso, retornamos da base quantas linhas foram afetadas, que será só uma, pois estamos usando a chave primária.

```
[HttpDelete("{id}")]
0 references
public async Task<IActionResult> Delete(int id)
{
    try
    {
        Personagem pRemover = await _context.Personagens
            .FirstOrDefaultAsync(p => p.Id == id);

        _context.Personagens.Remove(pRemover);
        int linhasAfetadas = await _context.SaveChangesAsync();

        return Ok(linhasAfetadas);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

8. Crie uma controller chamada UsuariosController.cs realizando os passos semelhantes as etapas 1 e 2 (configuração básica da controller e declaração/inicialização do atributo \_context).
9. Crie um método para obter o usuário através do username e e-mail. Será necessário os usings: *System.Threading.Tasks; RpgApi.Models; RpgApi.Data; Microsoft.EntityFrameworkCore;*

```
[HttpPost("GetUser")]
0 references
public async Task<IActionResult> Get(Usuario u)
{
    try
    {
        Usuario uRetornado = await _context.Usuarios
            .FirstOrDefaultAsync(x => x.Username == u.Username && u.Email == u.Email);

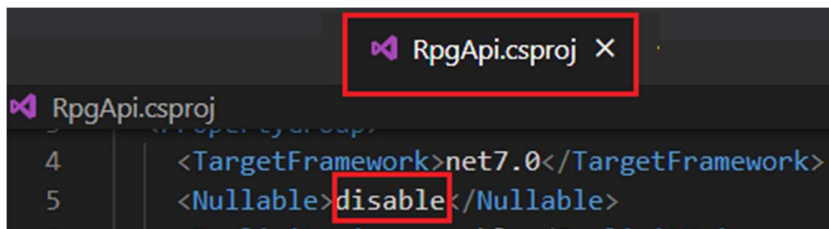
        if(uRetornado == null)
            throw new Exception("Usuário não encontrado");

        return Ok(uRetornado);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



10. Siga as instruções a seguir antes de realizar os testes

- Salvar todos os arquivos, Certificar que o SQL está ativo nos serviços do Windows, rodar o comando “*dotnet ef database update*” para recriar as tabelas.
- Alterar para disable a configuração de nulos no arquivo .csproj.



- Execute a API e realize os testes com o *Postman* para Listar todos, listar por Id, deletar pelo Id, adicionar um personagem e atualizar, sempre conferindo as atualizações nas tabelas no banco de dados.
- **Informação:** Caso queira alterar a base de dados usada, basta alterar a informação contida na classe **Program.cs**, conforme o trecho sinalizado, por uma chave existente no arquivo appSettings.json, que contenha um caminho de banco de dados válido. Fica como informação para quando utilizarmos base de dados que tenham strings de conexão diferentes.

```
builder.Services.AddDbContext<DataContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("ConexaoLocal"));
});
```