

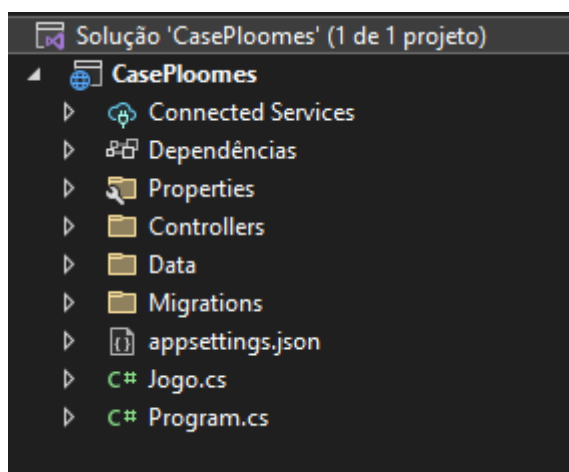
# TESTE PRÁTICO C# - PLOOMES

## Criação da aplicação em ASP.NET Web Api

Para cumprir o desafio proposto, criaremos uma Aplicação Web ASP.NET utilizando o .NET 6. Com as novas funcionalidades, temos o Swagger já configurado no Program.cs e configurado para que possamos utilizá-lo normalmente.

## Estrutura do Projeto

O projeto será iniciado com sua classe de Program.cs contendo as configurações para rodar projeto, juntamente com o arquivo de appsettings.json contendo a string de conexão para o banco de dados Sql Server, entre outras informações padronizadas. Na pasta “Properties” estão todas as configurações do projetos e suas extensões como definição de porta, perfil do swagger e configuração do ambiente de desenvolvimento.



## Criando Classe Jogo

A ideia parte de um princípio básico e simples, uma classe jogo contendo apenas sua identificação(Id), título e a empresa responsável pelo desenvolvimento do jogo.

```
8 referências
public class Jogo
{
    1 referência
    public int Id { get; set; }
    1 referência
    public string? Titulo { get; set; }
    1 referência
    public string? Produtora { get; set; }
}
```

## Desenvolvendo o Controller

Após a criação da Classe Jogo.cs podemos partir para o desenvolvimento dos métodos na classe de controladores. Começando com o método Get, foi criado um objeto com a finalidade de testar se já estava aparecendo na interface de aplicação e realizando sua determinada ação. Com isso, foi possível induzir o valores dos atributos para identificar se o valor estava retornando corretamente.

```
[HttpGet]
0 referências
public async Task<ActionResult<List<Jogo>>> Get()
{
    var jogos = new List<Jogo>
    {
        new Jogo { Id = 1, Titulo = "God of War", Produtora = "Santa Monica" }
    };
    return Ok(jogos);
}
```

Ao concluir o teste inicial é possível dar prosseguimento para desenvolver os próximos métodos para consultar através do código e incluir um jogo de sua escolha. Esse estilo de código é por conta de que não estamos trabalhando com Sql nesse primeiro momento, então estamos utilizando objetos diretamente instanciados para definição de valores. Para exemplificar nosso método Get com Id temos:

```
[HttpGet("{id}")]
0 referências
public async Task<ActionResult<Jogo>> Get(int id)
{
    var jogo = jogos.Find(f => f.Id == id);
    if (jogo == null)
        return BadRequest("Jogo não Encontrado.");
    return Ok(jogo);
}
```







```
[HttpPost]
0 referências
public async Task<ActionResult<List<Jogo>>> AdicionaJogo(Jogo jogo)
{
    jogos.Add(jogo);
    return Ok(jogos);
}
```

Caso o retorno não venha da forma como esperado, será retornada uma mensagem de “Bad Request”.

## Pacotes necessários

Com a criação dos métodos e validação dos cenários de teste, será necessário realizar o *download* de alguns pacotes para perfeito funcionamento da nossa aplicação.

São eles o EntityFrameworkCore, EntityFrameworkcoreDesign e SqlServer:

 <b>Microsoft.EntityFrameworkCore</b> por Microsoft	6.0.6
 Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL, and other databases through a provider plugin API.	
 <b>Microsoft.EntityFrameworkCore.Design</b> por Microsoft	6.0.6
 Shared design-time components for Entity Framework Core tools.	
 <b>Microsoft.EntityFrameworkCore.SqlServer</b> por Microsoft	6.0.6
 Microsoft SQL Server database provider for Entity Framework Core.	

Instalados podemos criar nossa pasta de Migrations onde será criado nossas migrations com base no contexto que inserimos juntamente com a classe de DataContext na pasta Data.

Para prosseguir com o desenvolvimento do banco de dados será necessário instalarmos a ferramenta do EntityFramework no terminal.

Para instalar por completo o EF, pode-se utilizar o comando:

```
dotnet tool install --global dotnet-ef
```

Feita a instalação, executamos o seguinte comando que será responsável por criar a Migrations dentro da pasta da solução:

```
dotnet ef migrations add CreateInitial
```

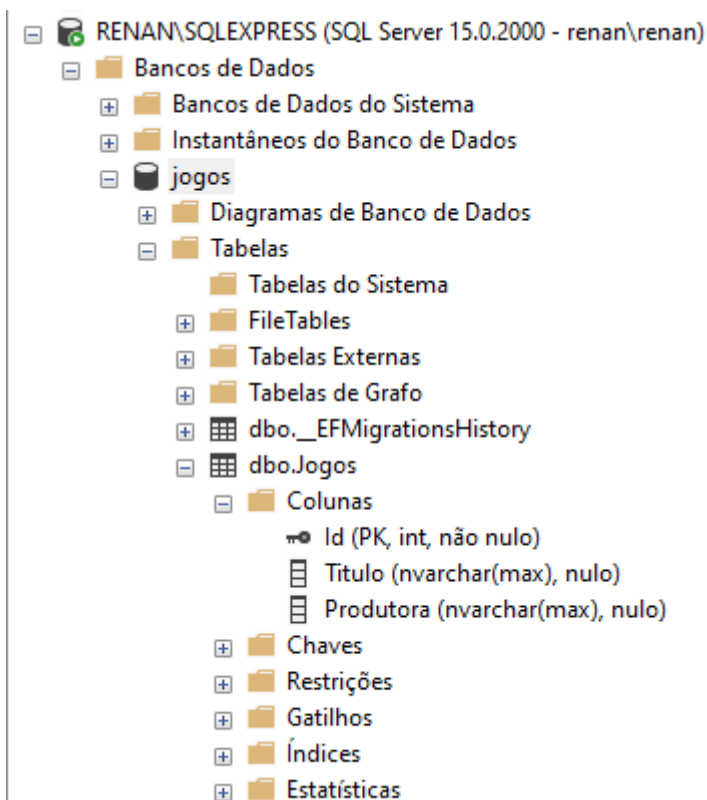
Desta forma, o arquivo foi gerado e juntamente com ele, toda nossa estrutura do banco de dados, agora temos conteúdo para utilizar.

[Arquivo da Migrations Criado]

```
1 referência
public partial class CreateInitial : Migration
{
    0 referências
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Jogos",
            columns: table => new
            {
                Id = table.Column<int>(type: "int", nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                Titulo = table.Column<string>(type: "nvarchar(max)", nullable: true),
                Produtora = table.Column<string>(type: "nvarchar(max)", nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Jogos", x => x.Id);
            });
    }
}

0 referências
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "Jogos");
}
}
```

[Estrutura Real do Banco de Dados]



Retornando ao projeto, temos que utilizar o connectionString do banco de dados no appsettings.json para conectar corretamente.

```
"ConnectionStrings": {  
  "DefaultConnection" : "server=localhost\\sqlexpress;database=jogos;"  
},  
"Logging": {  
  "LogLevel": {  
    "Default": "Information",  
    "Microsoft.AspNetCore": "Warning"  
  }  
},  
"AllowedHosts": "*"
```

Feito isso, voltamos ao arquivo principal para introduzir a conexão e integração entre o banco e aplicação.

```
1 global using CasePloomes.Data;  
2 using Microsoft.EntityFrameworkCore;  
3  
4 var builder = WebApplication.CreateBuilder(args);  
5  
6 // Add services to the container.  
7  
8 builder.Services.AddControllers();  
9 builder.Services.AddDbContext<DataContext>(options =>  
10 {  
11     options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));  
12 });  
13 // Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle  
14 builder.Services.AddEndpointsApiExplorer();  
15 builder.Services.AddSwaggerGen();
```

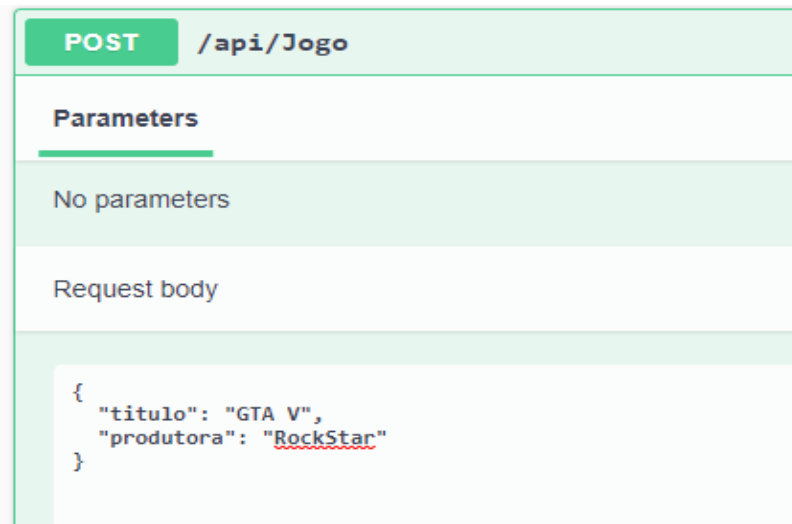
Com os conteúdos disponibilizados podemos utilizar dos mesmos para alterar os métodos passando os parâmetros corretamente.

```
[HttpPost]  
0 referências  
public async Task<ActionResult<List<Jogo>>> AdicionaJogo(Jogo jogo)  
{  
    _context.Jogos.Add(jogo);  
  
    try  
    {  
        await _context.SaveChangesAsync();  
        return Ok(await _context.Jogos.ToListAsync());  
    }  
    catch (Exception)  
    {  
        return BadRequest();  
    }  
}
```

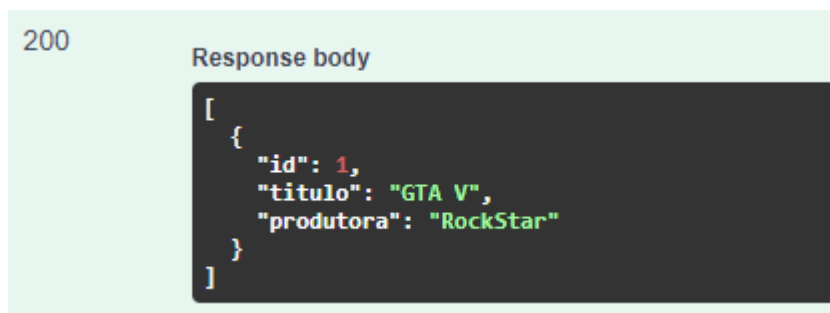
Pronto, estamos prontos para executar a aplicação =D

## Executando a Aplicação

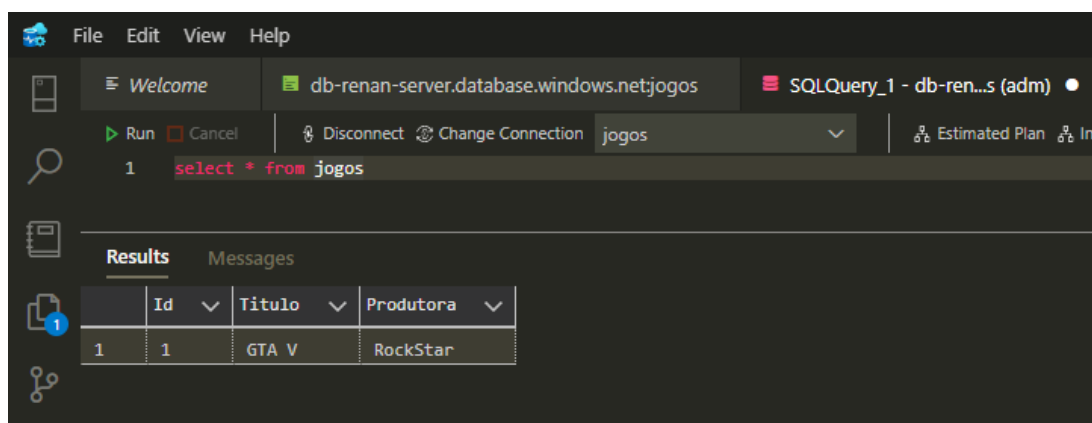
Para adicionarmos um jogo, selecionamos o método Post na página web. Em seguida, colocamos o nome do game e o de sua produtora.



Ao clicar em Execute, o valor será retornado em caso de sucesso como apresenta a imagem abaixo.



Ele já está armazenado em nossa base de dados.



E para consultá-lo com o método Get utilizando seu Id é bem simples;

Name	Description
<b>id</b> * required	
integer(\$int32)	2
(path)	

Selecione como valor o ID do jogo em que você quer pesquisar.

Code	Details
200	<div>Response body</div> <pre>{   "id": 2,   "titulo": "League of Legends",   "produtora": "Riot Games" }</pre>

Aplicação configurada com Azure e commitada no GitHub.