**PERSISTINDO DADOS COM ENTITY FRAMEWORK CORE**

**BD USADO: SQL SERVER**

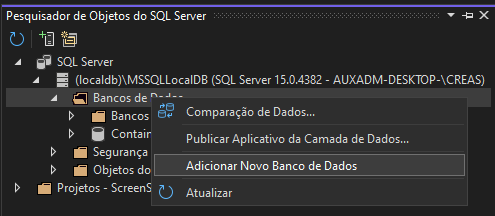
Considerando o projeto ScreenSound já funcional: <https://github.com/alura-cursos/3506-csharpWeb-screensound-curso1>

Pacote necessário:

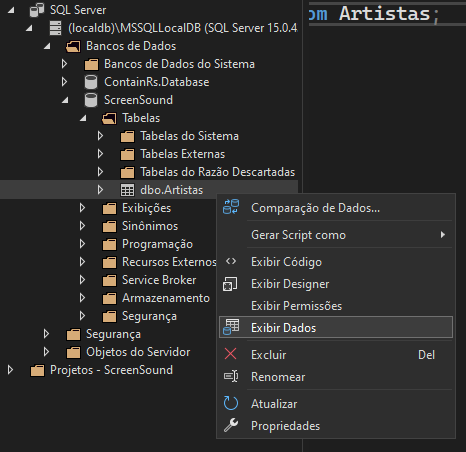
**Microsoft.EntityFrameworkCore.SqlServer - versão 7.0.14**

**CRIAR UMA TABELA ARTISTAS USANDO SQLSERVER**

Exibir -> Pesquisador de objetos do SQLServer



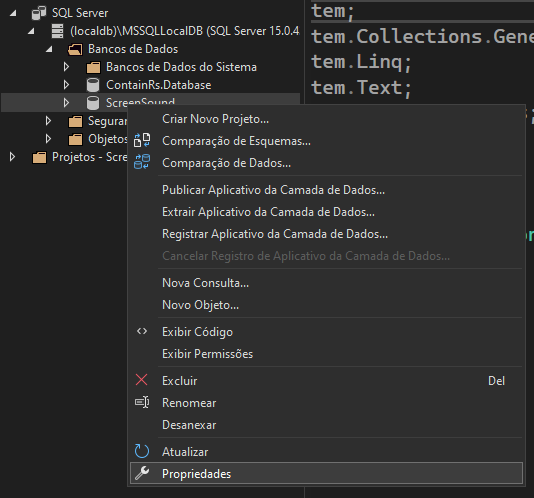
**EXIBIR E ADICIONAR DADOS DE UMA TABELA ATRAVÉS DO PESQUISADOR DE OBJETOS DO SQL SERVER**



**CONECTAR PROGRAMA COM BASE DE DADOS USANDO SqlConnection**

Baixar biblioteca **Microsoft.Data.SqlClient - v.7.0.14**

Criar uma pasta Banco com classe Connection, na qual consta uma variável string connectionString, que tem o caminho para o bando de dados criado anteriormente (esse caminho pode ser consultado nas propriedades, no campo “**cadeia de conexão**”) dentre outras propriedades.



E um método chamado ObterConexão, cujo tipo de retorno é SqlConnection

No program

try

{

using var connection = new Connection().ObterConexao();

connection.Open();

Console.WriteLine(connection.State);

}

catch(Exception e)

{

Console.WriteLine(e.Message);

}

return;

**LER DADOS DE UMA TABELA NO BD E MAPEAR PARA OS CAMPOS DE UM OBJETO**

Isso foi feito usado **SqlCommand e SqlDataReader, classes do ADO.NET**, dentro de um método Listar, na classe Connection,

public IEnumerable<Artista> Listar()

{

var lista = new List<Artista>();

using var connection = ObterConexao();

connection.Open();

string sql = "SELECT \* FROM Artistas";

SqlCommand command = new SqlCommand(sql, connection);

using SqlDataReader dataReader = command.ExecuteReader();

while (dataReader.Read())//ENQUANTO HOUVER LINHAS NA TABELA DO BD

{

//["Nome"] - CAMPO NO BD

string nomeArtista = Convert.ToString(dataReader["Nome"]);

string bioArtista = Convert.ToString(dataReader["Bio"]);

int idArtista = Convert.ToInt32(dataReader["Id"]);

Artista artista = new Artista(nomeArtista, bioArtista)

{

Id = idArtista

};

lista.Add(artista);

}

return lista;

}

No program

try

{

var connection = new Connection();

var listaArtistas = connection.Listar();

foreach(var artista in listaArtistas)

{

Console.WriteLine(artista);

}

}

catch(Exception e)

{

Console.WriteLine(e.Message);

}

**ADICIONAR DADOS NUMA TABELA EXISTENTE A PARTIR DE UM OBJETO**

Primeiramente, para organizar o código, foi usado o conceito de **DAL - Data Acess Layer**, que determina que deva existir uma classe que realiza todas as operação de uma data entidade do BD, ou seja, uma classe ArtistaDAL que guarda todas as operações relacionadas à artista.

Banco->ArtistaDal

public IEnumerable<Artista> Listar()

{

var lista = new List<Artista>();

using var connection = new Connection().ObterConexao();

connection.Open();

string sql = "SELECT \* FROM Artistas";

SqlCommand command = new SqlCommand(sql, connection);

using SqlDataReader dataReader = command.ExecuteReader();

while (dataReader.Read())//ENQUANTO HOUVER LINHAS NA TABELA DO BD

{

//["Nome"] - CAMPO NO BD

string nomeArtista = Convert.ToString(dataReader["Nome"]);

string bioArtista = Convert.ToString(dataReader["Bio"]);

int idArtista = Convert.ToInt32(dataReader["Id"]);

Artista artista = new Artista(nomeArtista, bioArtista)

{

Id = idArtista

};

lista.Add(artista);

}

return lista;

}

public void Adicionar(Artista artista)

{

//CONEXÃO COM BD

using var connection = new Connection().ObterConexao();

connection.Open();

string sql = "INSERT INTO Artistas " +

"(Nome, FotoPerfil, Bio) VALUES" +

"(@nome, @perfilPadrao, @bio)";

SqlCommand command = new SqlCommand(sql, connection);

command.Parameters.AddWithValue("@nome", artista.Nome);

command.Parameters.AddWithValue("@perfilPadrao", artista.FotoPerfil);

command.Parameters.AddWithValue("@bio", artista.Bio);

//EXECUÇÃO DO REGISTRO NA BD

//CUJO RETORNO É UM INTEIRO QUE DIZ QUANTAS LINHAS FORAM REGISTRADAS

int retorno = command.ExecuteNonQuery();

Console.WriteLine($"Registros adicionados: {retorno}");

}

**UTILIZANDO ENTITY FRAMEWORK NO ADO.NET**

Entity é um ORM - Object-Relational Mapping

Instalar no projeto o pacote



Classe Connection renomeada e com um novo método

internal class ScreenSoundContext : DbContext

{

private string connectionString = "Data Source=(localdb)\\" +

"MSSQLLocalDB;Initial Catalog=ScreenSound;" +

"Integrated Security=True;Encrypt=False;" +

"Trust Server Certificate=False;Application Intent=ReadWrite;" +

"Multi Subnet Failover=False";

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(connectionString);

}

}

E as devidas mudanças em Program e nas classes de menu.

Feito também a adição da tabela Musicas no BD, e criado também seu DAL, nota-se que os arquivos musicasDAL e artistasDAL são bem parecidos, permitindo nesse caso o uso do Generics, criando assim uma classe abstrata DAL<T> que com o uso de Generics determina a assinatura de seus métodos abstratos, para que sejam implementados nas classes ArtistaDAL e MusicaDAL

Banco-> DAL.cs

internal abstract class DAL<T>

{

public abstract IEnumerable<T> Listar();

public abstract void Adicionar(T objeto);

public abstract void Atualizar(T objeto);

public abstract void Deletar(T objeto);

}

Banco-> ArtistaDAL.cs

internal class ArtistaDAL : DAL<Artista>

.

.

public override IEnumerable<Artista> Listar()

{

.

.

Banco-> MusicaDAL.cs

internal class MusicaDAL : DAL<Musica>

.

.

public override IEnumerable<Musica> Listar()

{

.

.

Mesmo considerando que já houve bastante reaproveitamento de código, a implementação de cada método em ArtistaDAL e MusicaDAL ainda tem bastante repetição. Essas repetições podem ser repassadas pra classe DAL com o uso de Generics

Considerando alguns detalhes importantes:

Para que cada método de DAL seja capaz de identificar com qual tabela ele irá trabalhar nas classes ArtistaDAL e MusicasDAL, na sua assinatura temos que expliciar que o tipo genérico T usado é uma classe

internal abstract class DAL<T> where T : class

E em cada método, no momento de usar uma classe, utilizar o método Set

No método Listar, por exemplo

public IEnumerable<T> Listar()

{

return context.Set<T>().ToList();

}

**return context.Set<T>().ToList();**

Quando usado pela classe ArtistaDAL, que implementa a classe abstrata DAL, esse comando se torna

return context.Artistas.ToList();

Considerando que das classes ArtistaDAL e MusicaDAL, resta somente o método RecuperarPorNome, podemos levá-lo para a classe DAL

public T? RecuperarPor(Func<T, bool> condicao)

{

return context.Set<T>().FirstOrDefault(condicao);

}

O método é estruturado dessa forma, usando Func nos seus parâmetros para que ele possa receber uma função lambda no seu uso, ou seja, seu parâmetro será uma condição cujo valor vai ser definido por uma função lambda.

Um exemplo de seu uso vai ser na classe MenuRegistrarMusica

var artistaRecuperado = artistaDAL.RecuperarPor(

artista =>artista.Nome.Equals(nomeDoArtista));

A variável artistaRecuperado assume o valor do objeto cujo campo Artista.Nome seja igual ao valor passado em nomeDoArtista.

**MIGRATIONS**

Necessário pacotes

**Microsoft.Entity.FrameworkCore.Design -v 7.0.14**

**Microsoft.Entity.FrameworkCore.Tools - v 7.0.14**

Console->Gerenciador de Pacotes Nuget->Console do gerenciador de pacotes

Add-Migration ProjetoInicial

**INSERINDO MANUALMENTE DADOS NUMA TABELA A PARTIR DA CLASSE GERADA PELA MIGRATION**

Primeiramente, para iniciaR um BD vazio, mudar o campo InitialCatalog da variável connectionString em Banco->ScreenSoundContext, com um nome de uma nova base de dados para a aplicação.

Initial Catalog=ScreenSoundV0;

No console-> Update-Database ProjetoInicial

Abrir uma nova migration para inserção manual dos dados

No console-> Add-Migration PopularTabela

No método Up ->

migrationBuilder.InsertData("Artistas",

new string[] { "Nome", "Bio", "FotoPerfil" },

new object[] { "Djavan", "Djavan bio.", "https://cdn.pixabay.com/photo/2016/08/08/09/17/avatar-1577909\_1280.png" });

Para cada objeto que será inserido na tabela.

No método Down ->

migrationBuilder.Sql("DELETE FROM Artistas");

No console-> Update-Database

Assim os dados da migration foram inseridos na tabela Artistas.

**ALTERANDO UMA TABELA A PARTIR DA MUDANÇA NA CLASSE MUSICA**

Adicionar o atributo AnoLancamento à classe Musica

No console-> **Add-Migration AdicionarColunaAnoLancamento**

A migration gerada já traz as mudanças necessárias para o BD

No console-> **Update-Database**

**RELACIONAR MUSICA E ARTISTA**

Na classe Musica->Adicionar propriedade Artista

public Artista? Artista { get; set; }

Na classe Artista->Mudar o campo lista de musicas para uma coleção do tipo Musicas

public ICollection<Musica> Musicas = new List<Musica>();

Construir uma migration para essas mudanças

No console->

**Add-Migration RelacionarArtistaMusica**

**Update-Database**

Com a estrutura das tabelas pronta e relacionadas, os dados da coluna ArtistaId podem ser preenchidos pelo explorador de objetos do SQLServer.

**Lembrando que, as mudanças feitas diretamente pelo pesquisador de objetos do SQLServer não são registradas nas migrations, logo, precisam ser refeitas caso o projeto seja utilizado em outra máquina que uso um banco de dados local.**

**MUDANDO O MENU PARA REFLETIR A RELAÇÃO ENTRE ARTISTAS E MÚSICAS**

Pacote necessário

Microsoft.EntityFrameworkCore.Proxies

Mudar o método OnConfiguring da classe ScreenSoundContext para usar o pacote Proxies

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder

.UseSqlServer(connectionString)

.UseLazyLoadingProxies();

}

Propriedades que relacionam os modelos, em Artista e Musica precisam ser virtual para que possam ser usadas pelo Proxies, além das próprias classes públicas.

Artista

public virtual ICollection<Musica> Musicas { get; set; } = new List<Musica>();

Musica

public virtual Artista? Artista { get; set; }

Mudar classe MenuRegistarMusica para efetivar as mudanças

.

.

if (artistaRecuperado is not null)

{

Console.Write("Agora digite o título da música: ");

string tituloDaMusica = Console.ReadLine()!;

Console.Write("Agora digite o ano de lançamento da música: ");

int anoLancamento = int.Parse(Console.ReadLine()!);

artistaRecuperado.AdicionarMusica(new Musica(tituloDaMusica)

{

AnoLancamento = anoLancamento

});

Console.WriteLine($"A música {tituloDaMusica} de {nomeDoArtista} foi registrada com sucesso!");

artistaDAL.Atualizar(artistaRecuperado);//UPDATE NO BD

.

.

Além de um detalhe no método ExibirDiscografia, na classe Artista, para poder mostrar o ano de lançamento também

public void ExibirDiscografia()

{

Console.WriteLine($"Discografia do artista {Nome}");

foreach (var musica in Musicas)

{

Console.WriteLine($"Música: {musica.Nome} " +

$"- Ano de Lançamento: {musica.AnoLancamento}");

}

}

Com essas mudanças, temos um projeto simples que relaciona duas tabelas Artistas e Musicas.

**CRIANDO UMA API WEB COM ASP.NET CORE**

**Necessário**: projeto da aula anterior ou o projeto exemplo: <https://github.com/alura-cursos/3506-csharpWeb-screensound-curso1/tree/aula05-relacionamento>

**NOVO PROJETO - ScreenSound.API**

Com o objetivo de centralizar as informações de artistas e músicas em uma API, para que vários usuários possam consumir essa API, que vai futuramente estar hospedada num servido Azure, é criado um novo projeto que vai representar essa API

*Botão direito em ScreenSound Solução -> Adicionar -> Novo Projeto*

*Tipo: ASP.NET Core Vazio*

*Nome: ScreenSound.API - .net 8.0*

**MODELOS COMPARTILHADOS - Artista e Musica NUMA BIBLIOTECA DE CLASSES COMPARTILHADA**

Pensando em ter os modelos da aplicação acessíveis por todos os projetos criados, seja o console, a API ou qualquer outro futuramente criado, cria-se um biblioteca de classes com a pasta modelos (recortada do projeto console)

*Botão direito em ScreenSound Solução -> Adicionar -> Novo Projeto*

*Tipo: Biblioteca de classes*

*Nome: ScreenSound.Shared.Modelos - .net 8.0*

E para que o projeto console tenha acesso a essa biblioteca de classes:

*Botão direito em ScreenSound/Dependências -> Adicionar referência ao projeto -> selecionar ScreenSound.Shared.Models*

**ACESSO AO BD COMPARTILHADO**

Pensando em ter as configurações de acesso ao banco de dados disponível para todos os projetos, criamos um novo projeto ScreenSound.Shared.Dados com as pastas Banco e Migrations (recortadas de ScreenSound)

*Botão direito em ScreenSound Solução -> Adicionar -> Novo Projeto*

*Tipo: Biblioteca de classes*

*Nome: ScreenSound.Shared.Dados - .net 8.0*

Para que o projeto Shared.Dados funcione, ele vai precisar de duas coisas:

- Referência ao projeto Shared.Modelos;

*Botão direito em ScreenSound.Shared.Dados/Dependências -> Adicionar referência ao projeto -> selecionar ScreenSound.Shared.Models*

- Todas as bibliotecas do EntityFramework

Clique botão esquerdo em ScreenSound -> copiar tag ItemGroup

Clique botão esquerdo em ScreenSound.Shared.Dados -> colar

Para que o projeto ScreenSound funcione, adicionar a referência ao projeto Shared.Dados

*Botão direito em ScreenSound/Dependências -> Adicionar referência ao projeto -> selecionar ScreenSound.Shared.Dados*

**UTILIZANDO API - CONSULTA AOS ARTISTAS**

Tornar os projetos Shared.Modelos e Shared.Dados visível na API

*Botão direito em ScreenSound.Api/Dependências -> Adicionar referência ao projeto -> selecionar ScreenSound.Shared.Dados e ScreenSound.Shared.Modelos*

Mudar o método program/app.MapGet para efetuar a consulta

app.MapGet("/", () =>

{

var dal = new DAL<Artista>(new ScreenSoundContext());

return dal.Listar();

});

Porém, como o projeto API retorna os dados serializados para json no navegador, no momento de fazer essa serialização ele encontra um loop infinito, que existe na relação entre Artista e Música (Artista possui uma lista de músicas, e cada música possui um artista (que possui uma lista de músicas)....)

Para que a serialização ignore esse loop infinito, é necessário configurar a serialização na classe program do projeto API

var builder = WebApplication.CreateBuilder(args);

//IGNORAR CICLOS NA SERIALIZAÇÃO

builder.Services.Configure<Microsoft.AspNetCore.Http.Json.JsonOptions>

(options => options.SerializerOptions.ReferenceHandler =

ReferenceHandler.IgnoreCycles);

var app = builder.Build();

**MUDANÇA NAS FUNÇÕES DE CONSULTA**

Agora as rotas de consulta usam a classe Results, e a rota de consultar todos os artistas muda para /Artistas

app.MapGet("/Artistas", () =>

{

var dal = new DAL<Artista>(new ScreenSoundContext());

return Results.Ok(dal.Listar());

});

E é adicionada uma nova rota para consulta de um artista com nome específico

//LISTA PRIMEIRO ARTISTA COM NOME ESPECÍFICO

app.MapGet("/Artistas/{nome}", (string nome) =>

{

var dal = new DAL<Artista>(new ScreenSoundContext());

var artista = dal.RecuperarPrimeiroPor(

a => a.Nome.ToUpper().Equals(nome.ToUpper()));

if(artista is null)

{

return Results.NotFound();

}

return Results.Ok(artista);

});

**FUNÇÃO DE INSERÇÃO DE ARTISTA NA BD**

//ADICIONA UM ARTISTA RECEBIDO NO CORPO DA REQUISIÇÃO

app.MapPost("/Artistas",([FromBody]Artista artista) =>

{

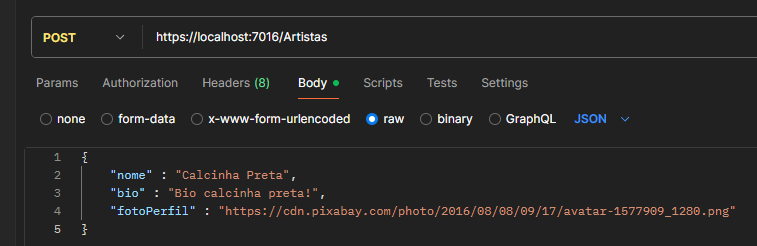
var dal = new DAL<Artista>(new ScreenSoundContext());

dal.Adicionar(artista);

return Results.Ok();

});

E para realizar a requisição com o objeto artista em formato json, utilizar o Postman



**INJEÇÃO DE DEPENDÊNCIA - ScreenSoundContext e DAL<Artista>**

Em todos os métodos atuais da API, há uma mesma linha, que é a instanciação de um objeto DAL<Artista>, responsável por estabelecer a conexão com a base de dados e pelos métodos de acesso aos dados. Para que não haja essa repetição, pode ser usada injeção de dependência para que, através do builder.Services, esse objeto seja disponibilizado para todos os métodos da API

//INJEÇÃO DE DEPENDÊNCIA - DAL<Artista> E ScreenSoundContext

builder.Services.AddDbContext<ScreenSoundContext>();

builder.Services.AddTransient<DAL<Artista>>();

E em cada método, listar esses objetos já instanciados pelo builder nos parâmetros

**CRUD - ARTISTA**

app.MapGet("/Artistas", ([FromServices] DAL<Artista> dal) =>

{

return Results.Ok(dal.Listar());

});

app.MapGet("/Artistas/{nome}", ([FromServices] DAL < Artista > dal,

string nome) =>

{

var artista = dal.RecuperarMuitosPor(

a => a.Nome.ToUpper().Equals(nome.ToUpper()));

if(artista is null)

{

return Results.NotFound();

}

return Results.Ok(artista);

});

app.MapPost("/Artistas",([FromServices] DAL < Artista > dal,

[FromBody]Artista artista) =>

{

dal.Adicionar(artista);

return Results.Ok();

});

app.MapDelete("/Artistas/{id}", ([FromServices] DAL<Artista> dal,

int id) =>

{

var artista = dal.RecuperarPrimeiroPor(a => a.Id.Equals(id));

if (artista is null)

{

return Results.NotFound();

}

dal.Deletar(artista);

return Results.NoContent();

});

app.MapPut("/Artistas", ([FromServices] DAL<Artista> dal,

[FromBody] Artista artista) =>

{

var artistaAtualizado = dal.RecuperarPrimeiroPor(

a => a.Id.Equals(artista.Id));

if(artistaAtualizado is null)

{

return Results.NotFound();

}

artistaAtualizado.Nome = artista.Nome;

artistaAtualizado.Bio = artista.Bio;

artistaAtualizado.FotoPerfil = artista.FotoPerfil;

dal.Atualizar(artistaAtualizado);

return Results.Ok();

/\*

artista [FromBody] -> artistaAtualizado <-> artista no BD

\*/

});

**MÉTODOS DE EXTENSÃO PARA ENDPOINTS DE ARTISTAS E MUSICAS**

Observando o código da classe program do projeto API, temos muitos endpoints, que podem ser agrupados em classes diferentes utilizando métodos de extensão.

Criar uma pasta Endpoints com duas classes: ArtistasExtensions e MusicasExtensions

**ArtistasExtension**

Criar um classe com um método de extensão se resume em:

- a classe precisa ser estática

- o método de extensão, que vai guardar os trechos de código pertinentes, é um método estático cujo parâmetro é *this WebApplication app*

public static class ArtistasExtensions

{

public static void AddEndPointsArtistas(this WebApplication app)

{

//TODOS ENDPOINTS DE ARTISTAS

}

}

MusicasExtensions

public static class MusicasExtensions

{

public static void AddEndPointsMusicas(this WebApplication app)

{

//TODOS ENDPOINTS DE MUSICAS

}

}

E no program

app.AddEndPointsArtistas();

app.AddEndPointsMusicas();

**USO DO SWAGGER**

Configurar projeto API com os pacotes necessários

<PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="8.0.0" />

<PackageReference Include="Swashbuckle.AspNetCore" Version="6.5.0" />

<PackageReference Include="Swashbuckle.AspNetCore.Swagger" Version="6.5.0" />

<PackageReference Include="Swashbuckle.AspNetCore.SwaggerUI" Version="6.5.0" />

Configurar o Swagger no program.cs

//SWAGGER

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(options =>

{

options.SwaggerDoc("v1", new Microsoft.OpenApi.Models.OpenApiInfo

{

Title = "ScreenSound API",

Version = "v1",

Description = "Uma API para gerenciamento de artistas e músicas",

});

/\*

options.SwaggerDoc("v1", new OpenApiInfo

{

Version = "v1",

Title = "ScreenSound API",

Description = "Uma API para gerenciamento de artistas e músicas",

});

\*/

});

.

.

//USO SWAGGER

app.UseSwagger();

app.UseSwaggerUI();

**DIMINUIR EXPOSIÇÃO DA ESTRUTURA DA APLICAÇÃO - USO DE DTO com Record**

Para diminuir, na documentação Swagger, qual é a estrutura necessária para criar um objeto Artista e cadastrá-lo na base, usaremos um objeto que vai intermediar essa operação e exibir apenas o essencial para o usuário da API.

*(Em estudos anteriores isso foi chamado de DTO - Data Tranfer Object)*

Para isso é criado um novo record ArtistaRequest, que vai guardar apenas as informações de Nome e Bio do artista

(Lembrando que record é uma estrutura similar a class, porém, mais leve e mais simples.)

ScreenSound.API/Requests/ArtistaRequest

public record ArtistaRequest(string nome, string bio);

Mudar o método mapPost no endpoint de Artistas para receber um objeto ArtistaRequest e não um Artista

app.MapPost("/Artistas", ([FromServices] DAL<Artista> dal,

[FromBody] ArtistaRequest artistaRequest) =>

{

var artista = new Artista(artistaRequest.nome, artistaRequest.bio);

dal.Adicionar(artista);

return Results.Ok();

});

Dessa forma, o cadastro de Artistas tem um DTO para diminuir a exposição de dados, o que pode ser observado no Swagger

