



Aula 03 - Entity Framework – ORM para conexão com banco de dados

Frameworks ORM (*Object-Relational Mapping*) representam uma forma de auxiliar o desenvolvedor a conectar uma api/aplicação ao banco de dados sem ter que fazer tudo manualmente e existem diversas ferramentas que empregam esta facilidade.

Esquenta sobre Frameworks ORM indicados

Canal Código Fonte TV – ORM (A ponte entre a O.O. e o Banco de dados) // Dicionário do programador:
<https://youtu.be/snOXxJa31GI>

Canal Alura – O que é um ORM: <https://youtu.be/x39vqeBTUmE>

Aprenderemos através do *Entity Framework Core* como realizar a conexão com banco de dados, utilizando o método Code First Migration, que cria o banco de dados a partir de classes de classes de Modelo existentes. Isso nos permitirá realizar as principais operações de Banco de dados que costumamos chamar de CRUD (Create, Read, Update e Delete)

1. Para configurar o uso do *Entity Framework Core*, abra o terminal e instale o pacote digitando `dotnet add package Microsoft.EntityFrameworkCore.SqlServer`. No decorrer do processo observe que será adicionada uma referência ao arquivo `RpgApi.csproj`
2. Agora instalaremos a ferramenta que permite trabalhar com os comandos do *Migration*, digitando no terminal o comando `dotnet tool install --global dotnet-ef`
3. Para finalizar instale o pacote com o comando `dotnet add package Microsoft.EntityFrameworkCore.Design` que permitirá fazer a customização das tabelas do banco de dados via programação C#

Para usar o *Entity Framework Core* precisaremos ter uma classe que realize a interação com o banco de dados representando uma sessão entre ela e o Banco, essa classe será a *DataContext* que herdará os métodos de *DbContext* (contida no framework)

4. Crie uma pasta **Data** e dentro desta pasta crie a classe **DataContext.cs**

```
using Microsoft.EntityFrameworkCore;

namespace RpgApi.Data
{
    0 references
    public class DataContext : DbContext
    {
    }
}
```

- Observe que estamos utilizando a herança à classe *DbContext* que necessita do `using` sinalizado. Use `CRTL + .` (ponto) para incluir mais rapidamente.



5. Crie um construtor dentro do corpo da classe. Atalho: digite ctor + clique TAB no teclado.

```
public class DataContext : DbContext
{
    0 references
    public DataContext(DbContextOptions<DataContext> options) : base(options)
    {
    }
}
```

- **Construtor** é como um método padrão que roda quando a esta classe for instanciada em memória, criando um objeto. O construtor **SEMPRE** tem o mesmo nome da classe e o modificador de acesso público.
 - Neste caso como a classe realiza uma herança da classe DbContext. O parâmetro Options que é recebido pelo construtor é passado também para a classe pai ou classe base.
6. Para configurar o mapeamento da classe Personagem no banco de dados utilize a codificação a seguir abaixo do construtor:

```
public DataContext(DbContextOptions<DataContext> options) : base(options)
{
}

0 references
1      2
public DbSet<Personagem> TB_PERSONAGENS { get; set; }
```

- (1) É a classe da pasta *Models*, será necessário fazer um *using* para adicionar o *namespace*.
 - (2) Representa o nome que será criado para a tabela no banco de dados.
7. Crie o método OnModelCreating que será responsável por alimentar a tabela de Personagens automaticamente quando o banco de dados estiver sendo criado

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Personagem>().HasData
    (
        //Inserir as linhas " new Personagem() { Id = 2,..." da lista de Personagens
    );
}
```

- Ao inserir os objetos da lista será necessário referenciar o using para RpgApi.Models.Enums



8. O método ficará com a seguir e ele utiliza a palavra-chave **override** que significa sobrescrever. Sobrescrita é um fundamento da orientação a objeto em que métodos pré-definidos ou existentes podem ser modificados tomando uma nova forma. Como o nome do método é `OnModelCreating` (durante a criação do modelo), entendemos que a criação acontecerá quando o banco de dados for criado ou atualizado.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Personagem>().HasData
    (
        new Personagem() { Id = 1 }, //Frodo Cavaleiro
        new Personagem() { Id = 2, Nome = "Sam", PontosVida=100, Forca=15, Defesa=25, Inteligencia=10,
        new Personagem() { Id = 3, Nome = "Galadriel", PontosVida=100, Forca=18, Defesa=21, Inteligencia=10,
        new Personagem() { Id = 4, Nome = "Gandalf", PontosVida=100, Forca=18, Defesa=18, Inteligencia=10,
        new Personagem() { Id = 5, Nome = "Hobbit", PontosVida=100, Forca=20, Defesa=17, Inteligencia=10,
        new Personagem() { Id = 6, Nome = "Celeborn", PontosVida=100, Forca=21, Defesa=13, Inteligencia=10,
        new Personagem() { Id = 7, Nome = "Radagast", PontosVida=100, Forca=25, Defesa=11, Inteligencia=10,
    );

    //Área para futuros Inserts no banco
}
```

9. Informe o nome da tabela que será criada no banco de dados para correlação com a classe `Personagem`

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Personagem>().ToTable("TB_PERSONAGENS");

    modelBuilder.Entity<Personagem>().HasData
    (
        new Personagem() { Id = 1, Nome = "Frodo", PontosVida = 100,
```

10. É possível criar um método de convenções gerais. O método a seguir vai fazer com que campos do tipo texto sejam criados no banco de dados como `varchar` com 200 caracteres caso não seja informado para a propriedade de maneira específica.

```
protected override void ConfigureConventions(ModelConfigurationBuilder configurationBuilder)
{
    configurationBuilder.Properties<string>().HaveColumnType("varchar").HaveMaxLength(200);
}
```



11. Agora será necessário configurar o caminho do Banco de Dados: Abra o arquivo `appsettings.json` e crie a estrutura a seguir, colando a string de conexão entre as aspas indicada na mensagem abaixo.

- Essa é uma estrutura de chave e valor para termos uma conexão de banco local, substitua pela indicação abaixo com os dados do seu banco SQL Server.

*Data Source=localhost; Initial Catalog=DB-DS-**NOME**; User Id=sa; Password=**SENHA**; TrustServerCertificate=True*

```
{..} appsettings.json > ...
1  {
2
3      "ConnectionStrings": {
4          "ConexaoLocal": "COLE A SUA STRING DE CONEXÃO LOCAL AQUI"
5      },
6
7      "Logging": {
8          "LogLevel": {
```

12. Vá até a classe `Program.cs` e programe a instrução que vai indicar que utilizaremos o contexto do Banco de Dados SQL Server junto com a string de conexão que indicamos (1), será necessário os usings no topo da classe conforme o trecho (2)

```
Program.cs
1  using Microsoft.EntityFrameworkCore; 2
2  using RpgApi.Data;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  builder.Services.AddDbContext<DataContext>(options => 1
7  {
8      options.UseSqlServer(builder.Configuration.GetConnectionString("ConexaoLocal"));
9  });
```

- Acesse File → Save All para salvar tudo que foi feito nas classes e arquivos.

Vamos conferir através do comando `dotnet ef -h` as opções que este comando oferece:

```
Commands:
  database      Commands to manage the database.
  dbcontext     Commands to manage DbContext types.
  migrations    Commands to manage migrations.
```

Como usaremos o *migrations*, use o comando `dotnet net ef migrations -h` para ver as opções de comandos.

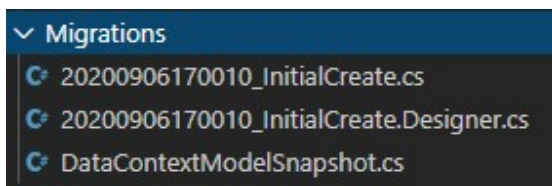
```
Commands:
  add      Adds a new migration.
  list     Lists available migrations.
  remove   Removes the last migration.
  script   Generates a SQL script from migrations.
```



13. O comando **migrations add** terá como missão transferir toda configuração das classes de modelo para o a migração que será feita. Adicionaremos junto ao comando a palavra **InitialCreate**, pois se trata da primeira vez que estamos fazendo os procedimentos com o **migrations**.

```
dotnet ef migrations add InitialCreate
```

Observe que foi criada uma pasta **Migrations** com alguns arquivos que conterão informações a respeito das tabelas e colunas que o **Entity Framework Core** utilizará para criar efetivamente as tabelas no Banco dados.



Observando a classe com sufixo **InitialCreate.cs**, podemos observar os métodos **Up** e **Down**, sendo que o primeiro exibe como será a configuração das tabelas, colunas e constraints detalhadamente, caso a migração seja confirmada, e o segundo desfaz o que for realizado no **Up**, caso seja necessário voltar a operação, o que costumamos chamar de **RollBack** e deletaria a tabela.


14. Execute o comando para confirmar e finalizar a criação da tabela na base de dados. Caso aconteça algum problema veja no arquivo **.csProj** como está a tag **InvariantGlobalization** e defina como **false**.

```
dotnet ef database update
```

- Abra o SQL Server, conecte ao servidor e confirme que a tabela e colunas estão criadas.

15. O Comando abaixo pode ser usado para criar um script SQL para ser executado caso o banco se perca ou tenha que ser criado em outro computador

```
dotnet ef migrations script -o ./script01_TabelaPersonagens.sql
```

- Será criado o arquivo abaixo  **script01_TabelaPersonagens.sql** com os comandos SQL que poderão ser executados no Somee ou no SQL

Deletar o banco de dados para novo comando de criação

- SQL Management Studio → Clicar com o botão direito no database → Delete → escolher as opções para fechar as conexões existentes (Close existente connections)

Nas próximas etapas configuraremos a controller para salvar os dados diretamente na base de dados que criamos nesta aula.