Entity Framework Core: Criando bases de dados com Migrations

[Facebook](https://www.facebook.com/sharer/sharer.php?u=http://www.devmedia.com.br/entity-framework-core-criando-bases-de-dados-com-migrations/36776" \o "Compartilhe no Facebook" \t "_blank) [Twitter](https://twitter.com/home?status=Entity%20Framework%20Core:%20Criando%20bases%20de%20dados%20com%20Migrations%0D%0Ahttp://www.devmedia.com.br/entity-framework-core-criando-bases-de-dados-com-migrations/36776" \o "Compartilhe no Twitter" \t "_blank)

 (1)  (0)

Neste artigo veremos como criar bases de dados através de entidades mapeadas com o Entity Framework Core em projetos ASP.NET Core.

**Motivação**

Em projetos onde a arquitetura é centralizada nas classes do domínio, a base de dados geralmente é criada no final da modelagem, por meio de um processo realizado automaticamente por ferramentas específicas ou frameworks.

Seguindo essa linha de projeto, neste artigo veremos como o Entity Framework Core, através do padrão Code First e da Fluent API, permite realizar o mapeamento das classes e a geração automática da base de dados a partir delas. Tudo isso será feito por meio de ferramentas de linha de comando, sem a necessidade de utilizar o Visual Studio.

http://www.devmedia.com.br/layout/icon/information-button.png Saiba mais sobre [como utilizar o ASP.NET Core](http://www.devmedia.com.br/criando-projetos-asp-net-core-com-yeoman/35721).

**Passo 1: Modelar as entidades**

Para modelar as entidades, que posteriormente serão refletidas em tabelas do banco de dados, criaremos aqui um projeto do tipo Class Library chamado *Projeto.Dominio* com duas classes: **Cliente** e **Pedido**, cujo código pode ser visto na**Listagem 1**.

|  |
| --- |
| 01 namespace Projeto.Dominio  02 {  03    public class Cliente  04    {  05        public Nullable<long> Id { get; set; }  06        public string Nome { get; set; }  07        public String Email { get; set; }  08        public IList<Pedido> Pedidos { get; set; }  09        public Cliente()  10        { }  11    }  12    public class Pedido  13    {  14        public Nullable<long> Id { get; set; }  15        public Double Total { get; set; }  16    }  17 } |

**Listagem 1**. Entidades do sistema

Note, na linha 8, que o cliente possui uma lista de pedidos, o que representa um relacionamento unidirecional.

**Passo 2: Adicionar as dependências**

Com o intuito de centralizar as configurações de persistência, criaremos uma segunda Class Library, com o nome *Projeto.Repositorio*. Feito isso, para que nela o Entity Framework funcione como esperado, precisamos inserir as dependências abaixo no arquivo *project.json* na seção *dependencies*:

01 "Microsoft.NETCore.App": "1.0.0",

02 "Microsoft.EntityFrameworkCore": "1.0.0",

03 "Microsoft.EntityFrameworkCore.Design": "1.0.0-preview2-final",

04 "Microsoft.EntityFrameworkCore.SqlServer": "1.0.0",

05 "Projeto.Dominio": "1.0.0"

O pacote *Microsoft.EntityFrameworkCore.Design*, na linha 03, é o responsável por permitir a execução do Migrations, que é um recurso do Entity Framework capaz de gerar e alterar bases de dados a partir das classes de modelo do projeto. Já na linha 05, estamos referenciando outra Class Library da nossa aplicação, a *Projeto.Dominio*. Essa é a forma com que o .NET Core lida com referências entre Class Libraries dentro de um mesmo projeto.

**Passo 3: Criar o DbContext**

Neste passo criaremos a classe que servirá de intermediária entre o modelo de classes e o banco de dados. Ela será responsável por processar o mapeamento objeto-relacional e fazer com que os dados trafeguem para ambos os lados: domínio e banco.

Sabendo disso, vamos criar um diretório chamado *FonteDeDados* e dentro dele a classe *PersisteContext.cs*, cujo código é apresentado na **Listagem 2**.

|  |
| --- |
| 01 using Projeto.Dominio;  02 namespace Projeto.Repositorio.FonteDeDados  03 {  04    public class PersisteContext : DbContext  05    {  06        public PersisteContext()  07        { }  08  09        public PersisteContext(DbContextOptions<PersisteContext> opcoes )  10            :base(opcoes)  11        { }  12  13        public DbSet<Cliente> Clientes { get; set; }  14        public DbSet<Pedido> Pedidos { get; set; }  15    }  16 } |

**Listagem 2**. PClasse PersisteContext.cs

Linhas 09 e 10: criamos um novo construtor para permitir a injeção de dependências do ASP.NET Core na camada de apresentação;

Linhas 13 e 14: mapeamos as classes para que sejam reconhecidas no processo de mapeamento objeto-relacional.

**Passo 4: Mapear as entidades com Fluent API**

A nova versão do EF aposentou a classe **EntityTypeConfiguration**. Diante disso, não podemos mais criar mapeamentos com a Fluent API separadamente do resto do código do DbContext. A partir de agora utilizaremos a classe **ModelBuilder**, que é passada como parâmetro ao método sobrescrito **OnModelCreating**, analisado mais adiante.

Antes de trabalhar diretamente nesse método, no entanto, criaremos na **PersisteContext** o mapeamento das entidades **Cliente**e **Pedido**. Para tal, utilizaremos métodos separados, um para cada entidade, passando **ModelBuilder**como parâmetro, de acordo com a **Listagem 3**.

|  |
| --- |
| 01 private void ConfiguraCliente(ModelBuilder construtorDeModelos)  02 {  03     construtorDeModelos.Entity<Cliente>( etd=>  04     {  05           etd.ToTable("tbCliente");  06           etd.HasKey(c => c.Id).HasName("id");  07           etd.Property(c => c.Id).HasColumnName("id").ValueGeneratedOnAdd();  08           etd.Property(c => c.Nome).HasColumnName("nome").HasMaxLength(100);  09           etd.Property(c => c.Email).HasColumnName("email").HasMaxLength(30);  10     });  11 }  12  13 private void ConfiguraPedido(ModelBuilder construtorDeModelos)  14 {  15     construtorDeModelos.Entity<Pedido>(etd =>  16     {  17           etd.ToTable("tbPedido");  18           etd.HasKey(p => p.Id).HasName("id");  19           etd.Property(p => p.Id).HasColumnName("id").ValueGeneratedOnAdd();  20           etd.Property(p => p.Total).HasColumnName("total");  21           etd.HasOne(c => c.Cliente).WithMany(p => p.Pedidos);  22     });  23 } |

**Listagem 3**. Novas formas de mapeamento do EF7

Em seguida, sobrescrevemos o método **OnModelCreating** e fazemos referência aos mapeamentos criados, como vemos na **Listagem 4**, concluindo assim a nossa classe de contexto.

|  |
| --- |
| 01 protected override void OnModelCreating(ModelBuilder construtorDeModelos)  02 {  03  construtorDeModelos.ForSqlServerUseIdentityColumns();  04  construtorDeModelos.HasDefaultSchema("ControlePedidos");  05  06  ConfiguraCliente(construtorDeModelos);  07  ConfiguraPedido(construtorDeModelos);  08 } |

**Listagem 4**. Referência dos métodos de mapeamento

Linha 3: instruímos o EF para que o banco de dados escolhido converta as chaves primárias utilizando o tipo **Identity**, que incrementa um valor inteiro a cada novo registro, trabalhando em conjunto com o método ValueGeneratedOnAdd(), visto na **Listagem 3;**

Linha 04: definimos o nome do banco de dados que será gerado pelo EF;

Linhas 6 e 7: invocamos os métodos criados anteriormente para realizar o mapeamento das duas entidades, evitando que o código do **OnModelCreating**se torne muito extenso.

http://www.devmedia.com.br/layout/icon/information-button.png Saiba mais sobre [mapeamento com Entity Framework Code First e Fluent API](http://www.devmedia.com.br/realizando-mapeamento-objeto-relacional-com-fluent-api-e-entity-framework-code-first/33364).

**Passo 5: Permitir à Class Library executar o Migrations**

Até aqui, criamos a nossa camada de repositório desacoplada das demais, mas o Entity Framework Core não permite a criação de bancos de dados via Migrations em Class Libraries.

Caso executemos os comados do Migrations nesse momento, receberemos duas mensagens de erro específicas. A primeira delas é:

|  |
| --- |
| "Could not invoke this command on the startup project 'Projeto.Repositorio'.  This preview of Entity Framework tools does not support commands on class library projects in ASP.NET Core and .NET Core applications." |

Para resolver essa situação é necessário criar um método Main na mesma Class Library, e para tal precisaremos adicionar uma classe na raiz do *Projeto.Repositorio*, chamada *Principal.cs*, cujo código pode ser visto no código abaixo:

|  |
| --- |
| 01 public class Principal  02 {  03  static void Main(string[] args){}  04 } |

Também precisamos abrir o project.json e dentro da seção buildOptions mudar o valor de "emitEntryPoint" para true, indicando que essa Class Library vai precisar gerar um executável. Uma regra básica que podemos utilizar para decidir qual valor atribuir a essa propriedade é: caso haja um método Main, então esse valor deverá ser true, caso não exista esse método, o valor será false.

Ao executarmos novamente qualquer comando do Migrations, receberemos uma outra mensagem de erro:

|  |
| --- |
| "No database provider has been configured for this DbContext.  A provider can be configured by overriding the DbContext. OnConfiguring method or by using AddDbContext on the application service provider.  If AddDbContext is used, then also ensure that your DbContext type accepts a DbContextOptions<TContext> object in its constructor and passes it to the base constructor for DbContext." |

Para permitir esse procedimento, de acordo com a mensagem precisamos criar uma classe estendendo uma interface do EF: **IDbContextFactory**. Na **Listagem 5** criamos essa classe dentro da camada de repositório, para que o Migrations a localize e permita a criação do banco de dados.

|  |
| --- |
| 01 namespace Projeto.Repositorio.FonteDeDados  02 {  03    public class FabricaPersisteContext : IDbContextFactory<PersisteContext>  04    {  05        private const string CONNECTIONSTRING = @"Data Source=GABRIEL-LAPTOP\SQLEXPRESSR2;Initial Catalog=ControlePedidos;Integrated Security=True";  06        public PersisteContext Create(DbContextFactoryOptions options)  07        {  08           var construtor = new DbContextOptionsBuilder<PersisteContext>();  09            construtor.UseSqlServer(CONNECTIONSTRING);  10            return new PersisteContext(construtor.Options);  11        }  12    }  13 } |

**Listagem 5**. Criação de uma fábrica de DbContext

Assim, quando o **DbContext** for chamado via injeção de dependências do ASP.NET Core, nas camadas mais altas, automaticamente essa factory será chamada e então devolverá uma nova instância do DbContext, dependendo do tipo de injeção.

**Passo 6: Criando a base de dados com Migrations**

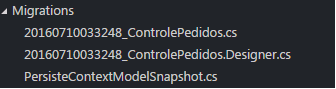
Para criar a primeira migration, devemos inserir no prompt de comandos o seguinte código:

|  |
| --- |
| dotnet ef migrations add Pedidos |

Executado esse passo, receberemos a seguinte mensagem:

|  |
| --- |
| Done. To undo this action, use 'dotnet ef migrations remove' |

Com esse comando, o Migrationscriará uma estrutura semelhante à que vemos na **Figura 1**, a partir da qual será gerado o nosso banco de dados.

  
**Figura 1.** Estrutura criada pelo Migrations

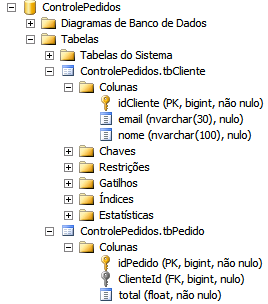
Agora, para sincronizar o modelo de classes com o banco de dados, digitamos o comando abaixo:

|  |
| --- |
| dotnet ef database update |

Caso tudo ocorra sem quaisquer problemas, receberemos a seguinte resposta do Migrations:

|  |
| --- |
| Applying migration '20160710033248\_ControlePedidos'.Done. |

Nesse momento já podemos verificar se realmente o banco e suas tabelas foram criadas, como mostra a **Figura 2**.

  
**Figura 2.** Banco de dados e tabelas criados

Com essa proposta, quando alterações forem feitas nas entidades, basta adicionar novas migrations e executá-las para que o banco seja atualizado.