05

Mudando o nome da tabela

Transcrição

Quando criamos uma propriedade em `LojaContext`, o Entity usa o nome da propriedade como nome da tabela. O problema é que não temos uma propriedade criada para `Endereco`. Como faremos para alterar o nome da tabela?

Usaremos também o `modelBuilder` no método `OnModelCreating()`. Com o objeto `modelBuilder`, mapearemos a entidade `Endereco` para a tabela `Enderecos`. O método ficará da seguinte maneira:

```
internal class LojaContext : DbContext
{
    // ...

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder
            .Entity<PromocaoProduto>()
            .HasKey(pp => new { pp.PromocaoId, pp.ProdutoId });

        modelBuilder
            .Entity<Endereco>()
            .ToTable("Enderecos");

        modelBuilder
            .Entity<Endereco>()
            .Property<int>("ClienteId");

        modelBuilder
            .Entity<Endereco>()
            .HasKey("ClienteId");

    }

    // ...
}
```

Resolvemos o problema do nome da tabela. Porém ainda precisamos fazer com que a tabela `Enderecos` seja **dependente** da tabela `Clientes`. Criaremos na classe `Endereco` uma propriedade que faz referencia para a classe `Cliente`.

```
namespace Alura.Loja.Testes.ConsoleApp
{
    public class Endereco
    {
        public int Numero { get; internal set; }
        public string Logradouro { get; internal set; }
        public string Complemento { get; internal set; }
        public string Bairro { get; internal set; }
        public string Cidade { get; internal set; }
        public Cliente Cliente { get; set; }
```

```
    }
}
```

A propriedade `Cliente` faz sentido em nosso modelo, afinal um endereço precisa de um cliente. Executaremos o comando `Add-Migration Cliente` e veremos que a migração será criada. Um aviso será dado no Console informando que a entidade endereço contém uma propriedade com o estado ***Shadow Property***.

Analisando a classe de migração, veremos que primeiro ocorrerá a criação da tabela `Clientes` com as propriedades `Nome` e `EnderecoDeEntrega`. Em seguida a tabela `Enderecos` será criada com suas propriedades, inclusive com a chave primária sendo a estrangeira de `Clientes`, trabalhando em forma de cascata.

A migração está fazendo sentido. Para sincronizarmos com o banco de dados, usaremos o comando SQL `Update-Database`.

Com as tabelas criadas, executaremos a aplicação para persistirmos as informações que configuramos no método `Main()` da classe `Program`:

```
static void Main(string[] args)
{
    var fulano = new Cliente();
    fulano.Nome = "Fulaninho de Tal";
    fulano.EnderecoDeEntrega = new Endereco()
    {
        Numero = 12,
        Logradouro = "Rua dos Inválidos",
        Complemento = "sobrado",
        Bairro = "Centro",
        Cidade = "Cidade"
    };

    using (var contexto = new LojaContext())
    {
        var serviceProvider = contexto.GetInfrastructure<IServiceProvider>();
        var loggerFactory = serviceProvider.GetService<ILoggerFactory>();
        loggerFactory.AddProvider(SqlLoggerProvider.Create());

        contexto.Clientes.Add(fulano);
        contexto.SaveChanges();
    }
}
```

O Entity inseriu as informações em `Clientes` e em seguida `Enderecos`, sendo que a chave primária do endereço é a mesma de cliente. Com isso entendemos como funciona os diversos tipo de relacionamentos.

Na próxima aula veremos como recuperar objetos relacionados.

Quando executado, o `Add-Migration` o `ModelBuilder` pode se perder, caso isso aconteça, ele pode gerar primeiro a tabela `Enderecos` antes da `Clientes` e não gera a dependência em cascata. Para corrigir esse problema basta adicionar na classe `Endereco` a propriedade `ClienteId`, não criar a ***ShadowProperty*** e referenciar a chave primaria o `ClienteId`.