01

BaseRepository

Transcrição

> Começando deste ponto? No link a seguir, você pode fazer o DOWNLOAD completo do projeto (https://github.com/alura-cursos/ASPNETCore20/archive/2b7ce47a897e36bed019548fa7169a249fa779c4.zip) para continuar seus estudos a partir deste capítulo.

Nossa aplicação está funcionando e, aparentemente, o catálogo está correto, mas se olharmos nossa tabela de produtos (`dbo.Produto`), perceberemos que os produtos são importados e duplicados no banco de dados. Isto é, o arquivo `livros.json` é importado repetidas vezes, cada vez que subimos o servidor.

Colocaremos um filtro no nosso repositório de produtos para fazermos uma única importação. Assim, caso o produto já exista, ele será ignorado. Vamos abrir `ProdutoRepository.cs` e pegar o método `SaveProdutos()` , no qual incluiremos o filtro para evitar a duplicação.

Caso o produto não exista, faremos a importação do mesmo, a partir do Dbset, naquele conjunto de entidades mantido pelo Entity Framework, equivalente a uma tabela e representado pelo `contexto.Set<Produto>()` , o qual colocaremos dentro da condição `if` .

Faremos uma consulta para verificar se um produto com o código que está sendo importado existe ou não, com a cláusula do LINQ `Where()` , que utilizará uma expressão lambda. `p.Codigo` terá que ser igual a `livro.Codigo` , e esta expressão retornará os elementos que têm o mesmo código do livro que está sendo importado.

E, então, chamaremos um método (`Any()`), que retornará verdadeiro ou falso, verdadeiro somente se o produto for encontrado. Caso este não seja encontrado, executaremos o código da linha que fará a importação deste produto, que moveremos para cima.

```
public void SaveProdutos(List<Livro> livros)
{
    foreach (var livro in livros)
    {
        if (!contexto.Set<Produto>().Where(p => p.Codigo == livro.Codigo).Any())
        {
            contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));
        }
    }
    contexto.SaveChanges();
}
```

Aproveitando que estamos no repositório, vamos eliminar as duplicações da expressão `contexto.Set<Produto>()` . Para isso, extrairemos uma **variável local** com "Ctrl + .", selecionando a opção "Introduzir local para "contexto.Set()"".

Com isso, será criado um `dbSet` , uma variável local que representa o DbSet de `Produto` . No entanto, queremos criar um campo para ser reutilizado durante o tempo de vida desta classe.

```
foreach (var livro in livros)
{
```

```
    {
        Microsoft.EntityFrameworkCore.DbSet<Produto> dbSet = contexto.Set<Produto>();
        if (!dbSet.Where(p => p.Codigo == livro.Codigo).Any())
        {
            contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));
        }
    }
    contexto.SaveChanges();
```

Assim, moveremos a linha para cima, e colocaremos o `DbSet` como campo privado:

```
    private readonly ApplicationContext contexto;
    private readonly DbSet<Produto> dbSet = contexto.Set<Produto>();
```

Com isto, temos a declaração do `dbSet`, que deixaremos só, e moveremos a atribuição do `dbSet`, isto é, o trecho `dbSet = contexto.Set<Produto>();`, para baixo, ao construtor, após o qual poderemos eliminar a atribuição da linha de cima:

```
    private readonly ApplicationContext contexto;
    private readonly DbSet<Produto> dbSet;

    public ProdutoRepository(ApplicationContext contexto)
    {
        this.contexto = contexto;
        dbSet = contexto.Set<Produto>();
    }
```

Feito isso, substituiremos todos os lugares em que há `contexto.Set<Produto>()` pelo campo local `dbSet`, em:

```
    public IList<Produto> GetProdutos()
    {
        return dbSet.ToList();
    }
```

Assim como em:

```
    if (!dbSet.Where(p => p.Codigo == livro.Codigo).Any())
    {
        dbSet.Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));
    }
```

Em seguida, criaremos uma classe base que ajudará na eliminação destas duplicações de código, para todos os repositórios a serem utilizados. Além de `ProdutoRepository`, usaremos `CadastroRepository`, `PedidoRepository` e `ItemPedidoRepository`. A classe base será criada na pasta "Repositories" e se chamará "BaseRepository".

Criaremos esta classe copiando um trecho de código do `ProdutoRepository` — os dois campos locais, `contexto` e `dbSet`. Também reaproveitaremos o construtor, copiando o trecho correspondente e colando em `BaseRepository`.

```
    public class BaseRepository
    {
        private readonly ApplicationContext contexto;
```

```
    private readonly DbSet<Produto> dbSet;

    public ProdutoRepository(ApplicationContext contexto)
    {
        this.contexto = contexto;
        dbSet = contexto.Set<Produto>();
    }
}
```

Agora, modificaremos o nome do construtor para `BaseRepository`, importaremos a referência de `DbSet` e de `Produto`. Nossa classe precisará ser genérica, e não específica para `Produto`, como está no momento. No lugar de `Produto`, utilizaremos um tipo genérico, um *base model*, `T`.

Ele terá que ser uma entidade que herdará do `BaseModel`, isto é, será preciso um filtro para o tipo `T`, com `where`. Também teremos que alterar a visibilidade dos campos para deixá-los como `protected`, pois queremos que eles sejam visíveis por todas as classes derivadas.

O código ficará da seguinte maneira:

```
public class BaseRepository<T> where T : BaseModel
{
    protected private readonly ApplicationContext contexto;
    protected private readonly DbSet<T> dbSet;

    public BaseRepository(ApplicationContext contexto)
    {
        this.contexto = contexto;
        dbSet = contexto.Set<T>();
    }
}
```

Em `ProdutoRepository.cs`, colocaremos que a classe herdará de `BaseRepository`, cujo tipo será `Produto`, eliminaremos os campos locais que tínhamos criado na classe, bem como o construtor, pois iremos recriá-lo selecionando `ProdutoRepository`, usando "Ctrl + ." e clicando em "Gerar construtor "ProdutoRepository(contexto)"".

```
public class ProdutoRepository : BaseRepository<Produto>, IProdutoRepository
{
    public ProdutoRepository(ApplicationContext contexto) : base(contexto)
    {
    }

    public IList<Produto> GetProdutos()
    {
        return dbSet.ToList();
    }
    // código omitido
}
```

Continuando, vamos criar outras classes de repositório, selecionando a pasta "Repositories > Adicionar > Classe..." com o botão direito do mouse. Criaremos `PedidoRepository`, classe que será herdada de `BaseRepository` do tipo `Pedido`.

Implementaremos o construtor com "Ctrl + ." e "Gerar construtor "PedidoRepository(contexto)"". Então, faremos uma cópia deste arquivo para criarmos `ItemPedidoRepository`, não esquecendo de renomear a classe e o construtor no arquivo também, além de trocarmos o tipo de `BaseRepository`, que é `ItemPedido`, e não `Pedido`.

Por fim, faremos o mesmo para a classe `CadastroRepository`, ou seja, trocando a classe e o construtor para deixá-los com o mesmo nome, e substituindo `Pedido` por `Cadastro`.

Todas estas classes que criamos poderão ser injetadas por injeções de dependências. Para isto, modificaremos a classe `Startup`, na qual incluiremos tudo que acabamos de criar e onde se localizam as instâncias temporárias:

```
services.AddTransient<IDataService, DataService>();
services.AddTransient<IProdutoRepository, ProdutoRepository>();
services.AddTransient<IPedidoRepository, PedidoRepository>();
services.AddTransient<ICadastroRepository, CadastroRepository>();
services.AddTransient<IItemPedidoRepository, ItemPedidoRepository>();
```

Criaremos as interfaces, começando por `PedidoRepository`, que acessaremos e deixaremos assim:

```
namespace CasaDoCodigo.Repositories
{
    public interface IPedidoRepository
    {

    }

    public class PedidoRepository : BaseRepository<Pedido>, IPedidoRepository
    {
        public PedidoRepository(ApplicationContext contexto) : base(contexto)
    }
}
```

Faremos o mesmo para as demais interfaces, de modo equivalente, fazendo os ajustes necessários. Estamos com os repositórios criados e prontos para serem usados!