08

ProdutoRepository

Transcrição

A partir de agora, faremos pequenas alterações no nosso código para melhorarmos a qualidade e estrutura do nosso projeto. Temos, por exemplo, o método `InicializaDB()` que cria e salva produtos no banco de dados, além de ler o arquivo JSON. São muitas responsabilidades para um método só.

Sendo assim, extrairemos para um novo método as seguintes linhas, em que lemos o arquivo JSON. Selecionaremos o trecho e usaremos "Ctrl + ." e "Extract Method". O método será chamado de `GetLivros()`, e retornará uma lista de livros.

```
var json = File.ReadAllText("livros.json");
var livros = JsonConvert.DeserializeObject<List<Livro>>(json);
```

Em seguida, salvaremos os livros no banco de dados, em forma de produtos. E extrairemos o seguinte trecho para o método `SaveProdutos()`:

```
foreach (var livro in livros)
{
    contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));
}
contexto.SaveChanges();
```

Isto não é o suficiente para melhorarmos o nosso projeto. Na verdade, queremos ter uma classe especializada em ler, gravar, fazer alterações e manipular os dados da entidade de produtos. Para isso, utilizaremos um padrão de projeto de acesso a dados a partir da criação de um diretório. Clicaremos com o botão direito do mouse em "CasaDoCodigo > Add > New Folder" e renomearemos a pasta com "Repositories".

Porém, ao tentarmos nomear, seremos informados de que a pasta já existe. Portanto, clicaremos nela com o lado direito do mouse e em "Include In Project". Com isto, começaremos a criar nosso repositório de produtos, ao qual adicionaremos uma nova classe, `ProdutoRepository`.

Moveremos nosso método `SaveProdutos()` para esta nova classe, recortando-o de `DataService.cs`. Com isso, deixamos de ter acesso ao objeto `contexto`, porque ele ficou na classe `DataService`. Então, copiaremos o campo previamente criado para `contexto`, isto é, `private readonly ApplicationContext contexto`, e o colaremos na nova classe de repositório.

Em `ProdutoRepository.cs` teremos, portanto:

```
namespace CasaDoCodigo.Repositories
{
    public class ProdutoRepository
    {
        private readonly ApplicationContext contexto;

        private void SaveProdutos(List<Livro> livros)
        {
            foreach (var livro in livros)
            {
                contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));
```

```
                contexto.Set<Produto>().Add(new Produto(livro.Codigo, livro.Nome, livro.Preco));
            }
            contexto.SaveChanges();
        }
    }
}
```

No entanto, a classe `Repositories` não instanciará o `contexto`, pois ela o receberá em seu construtor através da injeção de dependência. Criaremos um novo construtor para a nova classe selecionando o campo `contexto`, e usando "Ctrl + ." e "Generate constructor". Com isso, o `contexto` será fornecido via injeção de dependência:

```
private readonly ApplicationContext contexto;

public ProdutoRepository(ApplicationContext contexto)
{
    this.contexto = contexto;
}
```

Vamos mudar a visibilidade de `SaveProdutos()`, de `private` para `public`:

```
public void SaveProdutos(List<Livro> livros)
{
    // foreach omitido
}
```

O programa indicará um problema em `SaveProdutos()`, pois `Livro` se encontra menos acessível do que `SaveProdutos()`, em `DataService.cs`. Recortaremos a classe e a colaremos junto à `ProdutoRepository`, ao fim de seu código e, além disto, usaremos `public`.

Em seguida, criaremos e extrairemos uma interface a partir da classe `ProdutoRepository`, usando "Ctrl + ." mais uma vez. Agora sim, a interface no novo arquivo está sendo gerada, `IProdutoRepository`, com o método `SaveProdutos()`.

Então, voltaremos a `DataService.cs` para podermos consumir esse método do repositório e, como `Livro` em `List<Livro> livros = GetLivros()` foi movido, usaremos "Ctrl + ." para resolvermos a referência. Assim, `SaveProdutos()` sumirá desta classe, sendo necessário acessarmos o `ProdutoRepository`, que está em `IProdutoRepository`.

Para isso, precisaremos registrar a interface `IProdutoRepository` na classe `Startup` para podermos utilizar a injeção de dependência. Usaremos `services.AddTransient` para criarmos uma instância temporária para este tipo, e incluiremos o nome de `IProdutoRepository`:

```
public void ConfigureServices(IServiceCollection services)
{
    // código omitido

    services.AddTransient<IDataService, DataService>();
    services.AddTransient<IProdutoRepository, ProdutoRepository>();
}
```

De volta a `DataService.cs` e, no local em que temos o construtor, estamos recebendo `ApplicationContext contexto`. No entanto, precisamos receber o `ProdutoRepository` no construtor do `DataService.cs`, então fica faltando criarmos um campo

para este repositório, a ser chamado de `produtoRepository`:

```
class DataService : IDataService
{
    private readonly ApplicationContext contexto;
    private readonly IProdutoRepository produtoRepository;

    public DataService(ApplicationContext contexto,
        IProdutoRepository produtoRepository)
    {
        this.contexto = contexto;
        this.produtoRepository = produtoRepository;
    }

    // código omitido
}
```

Utilizaremos este campo privado no `DataService` para chamar `SaveProdutos()`:

```
public void InicializaDB()
{
    contexto.Database.EnsureCreated();

    List<Livro> livros = GetLivros();

    produtoRepository.SaveProdutos(livros);
}
```

Rodaremos a app e constataremos que tudo funciona corretamente. Reabriremos o SQL Server para verificar o carregamento da lista de produtos no banco de dados, selecionando a tabela de produtos (`dbo.Produto`) em "Databases > CasaDoCodigo > Tables" com o botão direito do mouse e, em "View Data", para visualizarmos os dados.

Os produtos são carregados como esperado, a partir do nosso arquivo JSON, porém desta vez utilizando o repositório de produtos.