**Antes/Depois - Entity Framework**

**Antes:**

* Era necessário manter os scripts de criação e atualização da estrutura de dados.
* Data Access Object: Classes necessárias para acessar as entidades do banco de dados (DAO).
* Comunicação com o banco de dados feita através de outra linguagem (SQL), o SQL é montado no próprio código, através de uma string.
* Trabalho excessivo no caso de mudanças, mesmo que mínimas: Por exemplo, no caso da remoção da propriedade de uma classe, teriamos que refatorar grande parte da classe DAO para deixá-la funcional.

**ORM - CRUD com ADO .NET e EF Core**

* Ferramenta que facilita o acesso à dados. O Entity Framework é um ORM e permite que nós trabalhemos bem menos com o SQL em si, fazendo com que possamos concentrar nosso esforço em trabalhar com o C# em si.
* É necessária a instação, pois o Entity não vem disponível automaticamente na criação de um projeto.
* Install-Package Microsoft.EntityFrameworkCore.SqlServer -Version 1.1, comando utilizado para instalar o Entity Framework Core como dependência do provider SQL Server.
* Existem vários providers disponíveis atualmente e podemos vê-los na página da Microsoft. Um desses providers é o InMemory, que acessa um banco de dados que fica na própria memória da aplicação. Seu principal uso é para testes em classes que utilizam o EF Core.
* Context: Classe utilizada para pelo EF Core para persistir todas as classes que devem ser persistidas pela aplicação.
* Na classe Context, devemos utilizar a API do EF Core, informar as classes a serem persistidas e informar o banco a ser utilizado, sobrescrevendo o metódo OnConfiguring da classe DbContext.
* Feito isso, toda persistência utilizará uma instância da classe Context. Assim, para adicionar um objeto no nosso banco de dados, por exemplo, podemos utilizar tal código: context.Produtos.Add(p);
* Por padrão, o EF Core cria uma transação para englobar os metódos SQL necessários sempre que o metódo SaveChanges() é chamado. Se for necessário um maior controle sobre transações, podem ser utilizados os metódos BeginTransaction() e UseTransaction().
* Com o ADO .NET, para obter todos os produtos do banco, é necessário fazer um loop e, dentro dele, converter cada objeto para uma instância da classe produto. Após isso, adicionamos em uma lista. Com o EF Core, simplesmente fazemos um repository.Produtos.ToList().
* Com o ADO .NET, para remover um produto do banco de dados, é montado um SQL de delete, etc. Com o EF Core, simplesmente percorremos nossa lista de produtos e utilizamos repository.Produtos.Remove(item). Depois, chamamos o metódo saveChanges().
* Finalizando o CRUD, o ADO .NET também monta uma string SQL para fazer o update, e então passa os parâmetros necessários. Com o EF Core, simplesmente obtemos o produto com o repository.Produtos.First() (por exemplo), mudamos a propriedade que desejamos (por exemplo, produtos.Nome) e então chamamos o metódo repository.Produtos.Update(produto). Então, chamamos o metódo SaveChanges().
* Criaremos um metódo ProdutoDAO com as operações do Entity. Como os metódos serão os mesmos da ProdutoDAO que utiliza o ADO .NET, criamos uma interface a qual as duas classes herdarão.

**Como o EF Core monta o SQL**

* Como o EF sabe que, por exemplo, ao fazemos p1.Nome = “Harry Potter” e, então, chamarmos o metódo SaveChanges(), o objeto deve ser atualizado? Primeiro, o nosso contexto herda da classe DbContext, a classe raíz do EF. Essa classe possui o ChangeTracker, que é responsável por rastrear todas as mudanças do contexto.
* Com o metódo Entries() (contexto.ChangeTracker.Entries()) podemos obter a lista de todas as instâncias que estão sendo manipuladas pelo contexto. Essas entidades possuem estados no EF. Se aquela entidade muda o estado, o metódo saveChanges() vai ter que agir.
* Podemos obter o estado da entidade com a propriedade State para cada objeto obtido no metódo Entries(). O que acontece é que, ao obter objetos do banco de dados, é criada uma instância de EntityEntry para cada registro obtido do banco de dados. Esses registros obtidos possuem como seu primeiro estado “Unchanged”.
* Ao alterarmos alguma propriedade de um objeto, com p1.Nome = “Harry Potter”, por exemplo, a propriedade State da instância de EntityEntry referente à esse objeto se torna “Modified”. É dessa forma que o EF rastreia as mudanças em cada instância das entidades.
* O ChangeTracker sabe que deve fazer um UPDATE no banco quando uma propriedade for alterada através de um snapshot guardado dos objetos por padrão. Ou seja, quando aquele objeto é recuperado, seja com o ToList(), First(), dentre outros, ele executa o metódo DetectChanges(), que verifica as propriedades atuais com as do snapshot criado inicialmente.
* É possível desligar a propriedade booleana AutoDetectChangesEnabled, desativando o monitoramento automático das mudanças. Isso pode ser necessário quando ocorrer uma gravação massiva através do metódo SaveChanges(), já que o ChangeTracker irá percorrer toda a lista de objetos monitorados.
* Se o estado de um objeto for Modified, o EF executará um Update ao chamarmos o SaveChanges(). Cada Entry do ChangeTracker, como já dito, tem uma referência para o objeto em si, e ele se encontra na propriedade Entity. Ao adicionar um objeto no contexto, o objeto fica com o estado Added nas Entries() do ChangeTracker e, ao chamar o saveChanges(), um objeto com esse estado é salvo no banco de dados através de um INSERT. Após isso, o objeto muda para o estado Unchanged.
* Ao removermos um objeto do nosso contexto, sua Entry fica com o state “Deleted”. Qualquer objeto que possua sua Entry neste estado será apagado ao chamarmos o SaveChanges(). Um objeto, ao ser apagado no banco, não será mais monitorado pelo ChangeTracker.
* Com o metódo de contexto Entry(), contexto.Entry(objeto), podemos obter a Entry de um determinado objeto. Se adicionarmos um produto no contexto e o apargamos sem chamar o SaveChanges(), ele ficará primeiro como Added e, ao remover, ele sumirá da lista. Ainda podemos obter o State desse objeto, com o metódo Entry(). Podemos ver que o state deste objeto é “Detached”.

**Como o EF nos ajuda a evoluir a aplicação**

* Ao mudarmos o nome de uma propriedade, o ProdutoDAO do ADO .NET não compilou e retornou alguns erros. Isso acontece porque adicionamos vários parametros para montar nosso SQL com o nome da propriedade. Sendo assim, temos que trocar o nome da propriedade em vários lugares.
* O ProdutoDAO do EF não retornou nenhum erro.
* Para adicionar um novo campo no banco, teriamos que criar um script de ALTER TABLE e executá-lo. Porém, com o EF, podemos fazer isso de outra forma. Devemos instalar o pacote de migrations (Install-Package Microsoft.EntityFrameworkCore.Tools), que fará a sincronização de nossas classes com o banco de dados.
* Get-Help EntityFramework: Exibe uma lista de comandos que podem ser utilizados, como o Add-Migration e o Update-Database, que são os comandos que iremos utilizar para fazer nossas migrations.
* Primeiro, utilizamos o comando Add-Migration “nome” e, então, aplicamos a migration com o comando Update-Database

**Relacionamento um para muitos no EF Core**

* Criamos uma nova classe Compra, que possui as propriedades Quantidade, Produto, Preço e Id. Então, adicionamos a classe em nosso contexto e fizemos a migration, para incluí-la no banco de dados.
* A migration incluiu uma coluna ProdutoId, pois não podemos ter uma referência para o objeto no banco de dados, então o EF automaticamente criou essa coluna para nós, referenciando a chave primária da tabela Produto. Porém, a coluna foi criada como nullable, e não queremos isso, já que toda compra obrigatoriamente tem que ter um produto. Para resolvermos isso, adicionamos a propriedade ProdutoId na classe Compra. Como essa propriedade é um int, ao recriarmos a migration, o produtoId agora não é mais nullable.