

DESENVOLVIMENTO .NET

ENTITY FRAMEWORK

FLAVIO MORENI E PEDRO IVO



4

LISTA DE FIGURAS

Figura 4.1 – Instalando o Oracle.EntityFrameworkCore.....	6
Figura 4.2 – Classe DataBaseContext	7
Figura 4.3 – Estado das entidades do EF	13
Figura 4.4 – Diagrama de classe - Produto e Categoria	21
Figura 4.5 – <i>Extension Method</i> – Include	25
Figura 4.6 – Resultado do método Include.....	26
Figura 4.7 – Resultado da lista do relacionamento um para muitos.....	28

LISTA DE CÓDIGOS-FONTE

Código-fonte 4.1 – Script criação tabela Tipo Produto	7
Código-fonte 4.2 – Implementação DataBaseContext.....	8
Código-fonte 4.3 – Criando o DbSet para Tipo Produto	10
Código-fonte 4.4 – Modelo TipoProduto com anotações do EF	11
Código-fonte 4.5 – <i>Entity Framework</i> – Added.....	14
Código fonte 4.6 – <i>Entity Framework</i> – Modified	14
Código-fonte 4.7 – <i>Entity Framework</i> – Deleted.....	15
Código-fonte 4.8 – <i>Entity Framework</i> – Find	15
Código-fonte 4.9 – <i>Entity Framework</i> – List.....	15
Código-fonte 4.10 – Implementação classe TipoProdutoRepository com EF	17
Código-fonte 4.11 – OrderBy com LINQ.....	18
Código-fonte 4.12 – OrderBy descendente com LINQ	18
Código-fonte 4.13 – Cláusula <i>Where</i> com LINQ	19
Código-fonte 4.14 – Cláusula <i>Where</i> parametrizada variável comercializada com LINQ.....	19
Código-fonte 4.15 – Cláusula <i>Where</i> parametrizada variável idTipo com LINQ.....	19
Código-fonte 4.16 – Cláusula <i>Where</i> parametrizada variável descrição com LINQ ..	20
Código-fonte 4.17 – Cláusula <i>Where</i> e método <i>Contains</i> com LINQ	20
Código-fonte 4.18 – Script Oracle criação da tabela de produto	22
Código-fonte 4.19 – Anotações EF para a classe modelo Produto	23
Código-fonte 4.20 – DbSet Produto	23
Código-fonte 4.21 – Relacionamento um para um usando Include.....	24
Código-fonte 4.22 – Navigation Property para a lista de produtos	27
Código-fonte 4.23 – Relacionamento um para muitos usando Include	27

SUMÁRIO

4 ENTITY FRAMEWORK	5
4.1 Entity Framework Core	5
4.2 Implementação EF Core	7
4.2.1 Classe de Contexto	7
4.2.2 DbSet	9
4.2.3 Models e anotações	10
4.3 Outras anotações	11
4.3.1 Operações	11
4.3.2 Add	13
4.3.3 Modified – Update	14
4.3.4 Delete	14
4.3.5 Find	15
4.3.6 List	15
4.3.7 Todas as Operações	16
4.4 Operações Avançadas	17
4.5 LINQ	17
4.6 OrderBy	18
4.7 Where	18
4.8 Contains	20
4.9 Relacionamentos	20
4.10 Relacionamento um para um	24
4.11 Relacionamento um para muitos	26
4.12 Considerações finais	28
REFERÊNCIAS	30

4 ENTITY FRAMEWORK

4.1 Entity Framework Core

O *Entity Framework* ou EF é um conjunto de tecnologias ADO.NET que permite a você mapear seus objetos de modelos com entidade de banco de dados, podemos comparar o EF ao JPA da tecnologia Java.

É um framework testado e estabilizado por muitos anos para facilitar o acesso à base de dados, sem a necessidade de criações de camadas de conexões robustas ou até mesmo sem instruções SQL. Sua primeira versão foi lançada em meados de 2008, fazendo parte do .NET 3.5 SP1 e do Visual Studio 2008, e a partir do EF4.1 já começa a ser enviado ao NuGet.org, sendo uns dos pacotes mais baixados da atualidade.

Sabemos que o *Entity Framework* possui muitos assuntos detalhados, os quais são desbravados em livros e em diversos conteúdos. A cada necessidade e hábito, devemos analisar a melhor forma de implementação do EF, seguem algumas:

- Criar um banco de dados escrevendo apenas códigos, use o conceito de **Code First** para definir seu modelo e então gerar o banco de dados.
- Criar um banco de dados usando caixas de diálogos do Visual Studio para adicionar os modelos e então gerar o banco de dados, use conceito de **Model First**.
- Usar um banco de dados existente para criar seus modelos, use o conceito de **Database First**.

Lembrando que existem outras maneiras de implementação, porém decidimos usar a implementação com **Database First**, pelo fato de utilizarmos banco de dados Oracle.

A proposta deste capítulo é adicionar a biblioteca do *Entity Framework* em nosso projeto **FiapSmartCity**. Será preciso adicionar bibliotecas no projeto .NET, criar tabelas e relacionamentos no Oracle, remover os trechos de código que usam o padrão convencional ADO.NET e escrever os comandos para o EF.

Vamos iniciar pela instalação da biblioteca do EF para Oracle, acesse o *Nuget*, faça uma busca por Oracle e selecione a instalação da biblioteca **Oracle.EntityFrameworkCore**, veja na figura abaixo:

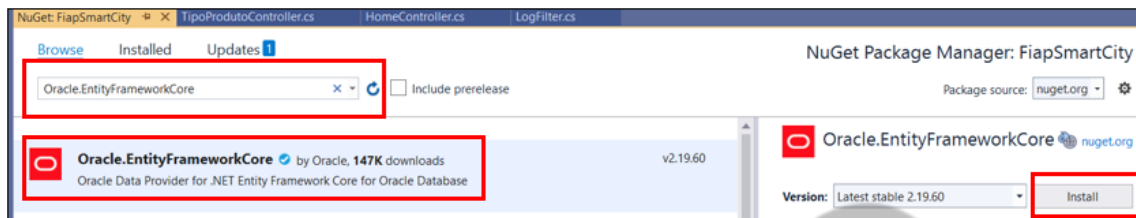


Figura 4.1 – Instalando o Oracle.EntityFrameworkCore
Fonte: Elaborado pelo autor (2018)

Não é preciso efetuar nenhuma alteração nesse arquivo, pois no capítulo anterior já adicionamos a string de conexão para o banco de dados **Oracle**.

Precisamos ter uma tabela para conseguir executar o trabalho com o EF, nos exemplos anteriores foi utilizada a tabela **TipoProduto**, vamos seguir usando a mesma.

Com a utilização dessa tabela anteriormente criada, vamos ter um problema no uso do EF, porém isso servirá como parte do aprendizado. O Oracle e EF são *case-sensitive*, ou seja, diferenciam maiúscula e minúscula, note que a tabela Tipo Produto possui seu nome e atributos declarados em caixa-alta. Com isso, serão necessários um trabalho maior e um pouco de digitação de código. Caso a tabela usada para trabalhar com o EF tenha suas colunas declaradas com o mesmo nome dos atributos de um modelo, pouca digitação ou quase nada será necessário para o funcionamento do EF.

Abaixo, segue o script para criação da tabela Tipo Produto no banco de dados Oracle:

```
CREATE TABLE TIPOPRODUTO (  
    IDTIPO      NUMBER          PRIMARY KEY,  
    DESCRICAOTIPO VARCHAR2(250) NOT NULL,  
    COMERCIALIZADO INT  
);  
  
CREATE SEQUENCE TIPOPRODUTO_IDTIPO_SEQ;  
  
CREATE OR REPLACE TRIGGER TR_SEQ_TIPOPRODUTO BEFORE INSERT ON  
TIPOPRODUTO FOR EACH ROW
```

```
BEGIN
SELECT TIPOPRODUTO_IDTIPO_SEQ.NEXTVAL
INTO :new.IDTIPO
FROM dual;
END;

--DROP TRIGGER TR_SEQ_TIPOPRODUTO;
--DROP SEQUENCE TIPOPRODUTO_IDTIPO_SEQ;
--DROP TABLE TIPOPRODUTO;
```

Código-fonte 4.1 – Script criação tabela Tipo Produto
Fonte: Elaborado pelo autor (2018)

Tudo pronto para a implementação!

4.2 Implementação EF Core

4.2.1 Classe de Contexto

Para que nossa aplicação possa utilizar as facilidades do framework EF Core, precisamos criar a classe de contexto, ou classe de acesso à base. A classe de contexto será uma subclasse de **System.Data.Entity.DbContext**, que tem a responsabilidade de interação com os objetos e com o banco de dados.

Dentro do *namespace* **Repository**, adicione uma pasta com o nome **Context**, em seguida adicione uma classe com o nome de **DataBaseContext**. Veja na figura:

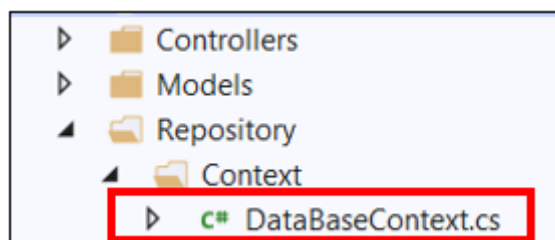


Figura 4.2 – Classe DataBaseContext
Fonte: Elaborado pelo autor (2018)

Agora é necessário declarar a classe **DataBaseContext** como uma subclasse de **System.Data.Entity.DbContext**. Em seguida, vamos sobrescrever o método **OnConfiguring** classe para declarar qual a string de conexão a ser.

No método **OnModelCreating**, vamos fazer duas configurações no EF e na conexão de dados.

O código abaixo apresenta a implementação da classe **DataBaseContext**, segue:

```
using System;
using System.IO;
using FiapSmartCity.Models;
using System.Data.Entity.DbContext;
using Microsoft.Extensions.Configuration;
using Oracle.EntityFrameworkCore;

namespace FiapSmartCity.Repository.Context
{
    public class DataBaseContext : DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            if (!optionsBuilder.IsConfigured)
            {
                var config = new ConfigurationBuilder().SetBasePath(Directory.GetCurrentDirectory()).AddJsonFile("appsettings.json").Build();

                optionsBuilder.UseOracle(config.GetConnectionString("FiapSmartCityConnection"));
            }
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
        }
    }
}
```

Código-fonte 4.2 – Implementação DataBaseContext
Fonte: Elaborado pelo autor (20S18)

4.2.2 DbSet

Vimos anteriormente a classe de contexto e algumas configurações particulares para nosso exemplo. Mas ainda precisamos incluir alguns itens na classe de contexto (**DbContext**) para seguirmos com os mapeamentos entre tabela e classes.

O item a ser incluído na classe de contexto é o objeto **DbSet**, que tem a responsabilidade de representar uma entidade e permitir a manipulação com as operações de criação, leitura, gravação e exclusão.

Na classe **DbContext**, é necessário declarar uma propriedade do tipo de **DbSet** para representar a entidade de TipoProduto. Veja no código-fonte Criando o DbSet para Tipo Produto:

```
using System;
using System.IO;
using FiapSmartCity.Models;
using System.Data.Entity.DbContext;
using Microsoft.Extensions.Configuration;
using Oracle.EntityFrameworkCore;

namespace FiapSmartCity.Repository.Contexts
{
    public class DataBaseContext : DbContext
    {
        public DbSet<TipoProduto> TipoProduto { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            if (!optionsBuilder.IsConfigured)
            {
                var config = new ConfigurationBuilder().SetBasePath(Directory.GetCurrentDirectory()).AddJsonFile("appsettings.json").Build();

                optionsBuilder.UseOracle(config.GetConnectionString("FiapSmartCityConnection"));
            }
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
```

```
{  
  
}  
  
}  
  
}
```

Código-fonte 4.3 – Criando o DbSet para Tipo Produto
Fonte: Elaborado pelo autor (2018)

4.2.3 Models e anotações

Agora é preciso deixar nosso modelo vinculado à nossa tabela. Para isso, vamos usar algumas anotações disponíveis nos *namespaces* **System.ComponentModel.DataAnnotations** e **System.ComponentModel.DataAnnotations.Schema**. São três (3) as anotações principais, a primeira é [Table], usada para classe; a segunda é [Key], que identifica a chave primária, e a terceira é [Column], para associar a propriedade da classe com a coluna da tabela.

IMPORTANTE: As anotações [Table], [Key] e [Column] não são comuns de ser utilizadas quando usamos um banco de dados diferente do Oracle, pois o EF, por convenção, associa tabela e campos pelo nome da classe e atributos. O cliente Oracle para EF é considerado muito pobre pela comunidade, pois itens básicos, como *case-sensitive*, não são tratados, causando diversos problemas de integração.

O código apresenta a classe **TipoProduto** com as anotações para o funcionamento do EF. Segue:

```
using System;  
using System.ComponentModel.DataAnnotations;  
using System.ComponentModel.DataAnnotations.Schema;  
  
namespace FiapSmartCity.Models  
{  
    [Table("TIPOPRODUTO")]  
    public class TipoProduto  
    {  
        [Key]  
        [Column("IDTIPO")]  
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]  
        public int IdTipo { get; set; }  
  
        [Required(ErrorMessage = "Descrição obrigatória!")]
```

```
[StringLength(50, ErrorMessage = "A descrição deve ter  
no máximo 50 caracteres")]  
[Display(Name="Descrição:")]  
[Column("DESCRICAOTIPO")]  
public String DescricaoTipo { get; set; }  
  
[Column("COMERCIALIZADO")]  
public bool Comercializado { get; set; }  
  
}  
}
```

Código-fonte 4.4 – Modelo TipoProduto com anotações do EF
Fonte: Elaborado pelo autor (2018)

4.3 Outras anotações

O *Entity Framework* disponibiliza outras anotações além das relatadas na seção anterior. É possível determinar tamanho de campos, definir uma chave estrangeira, determinar que o campo é requerido, dizer que um atributo não será mapeado com nenhuma coluna do banco de dados etc.

Segue a lista dos mais utilizados:

- MaxLength
- MinLength
- StringLength
- NotMapped
- ForeignKey
- InverseProperty

4.3.1 Operações

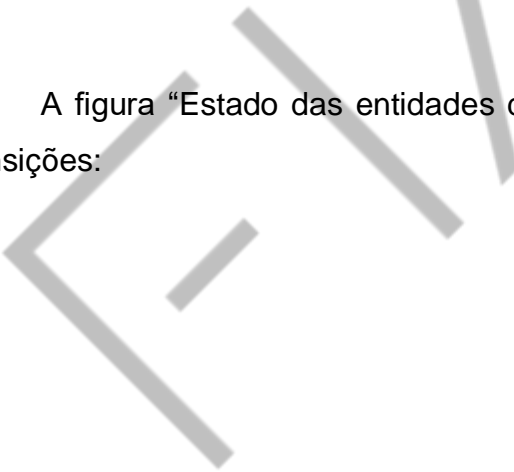
Temos implementado nossa classe de contexto, junto com a propriedade **DbSet**, nosso modelo está com as anotações necessárias, assim, temos o conjunto de Contexto, **DbSet** e **Model** preparados para executar operações no banco de dados com os recursos do *Entity Framework*.

Precisamos implementar as operações, porém precisamos entender um pouco do ciclo de vida de uma entidade no EF, desse modo ficará mais fácil entender as operações.

Temos cinco (5) estados para uma entidade no EF, são eles:

- **Detached** – a entidade (model) não está vinculada ao contexto, ou seja, não será persistida, alterada ou removida do banco de dados.
- **Unchanged** – a entidade está vinculada ao contexto, porém não sofreu nenhuma alteração. Toda consulta retornada do banco de dados tem por estado padrão o Unchanged.
- **Added** – indica que a entidade foi marcada para ser adicionada no banco de dados pelo contexto.
- **Modified** – a entidade teve alguma informação alterada, nesse caso o contexto precisa atuar para persistir a entidade no banco de dados.
- **Deleted** – indica que a entidade foi marcada para ser removida pelo contexto.

A figura “Estado das entidades do EF” apresenta os estados e as possíveis transições:



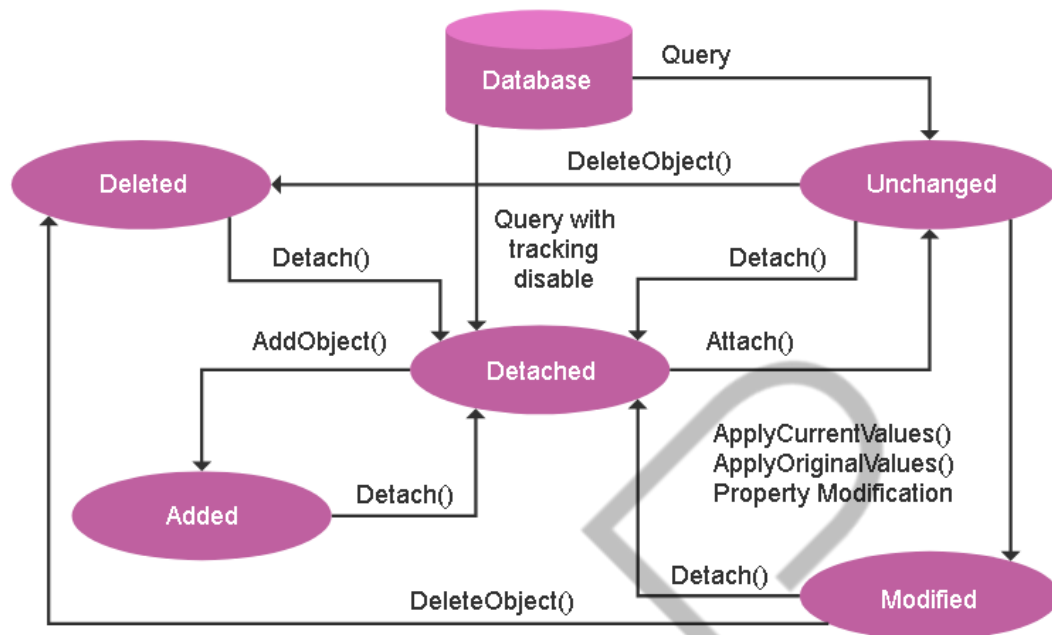


Figura 4.3 – Estado das entidades do EF
Fonte: Google Imagens (2018)

Entendido o estado de cada entidade, vamos às operações. No projeto **FiapSmartCity**, temos o componente da camada de acesso a dados (**TipoProdutoRepository**) com cinco (5) métodos que correspondem a comando SQL no banco de dados.

Um a um, vamos converter o código desses métodos para o uso dos recursos do EF. A versão atual utiliza recursos da biblioteca ADO.NET e possui comandos SQL descritos dentro de cada um deles. Com o EF, os comandos SQL serão removidos, a quantidade de linhas será reduzida e os comandos, simplificados.

4.3.2 Add

O bloco abaixo apresenta o método para a inserção de dados utilizando as propriedades do EF. Note a forma de uso, que inclui a criação da classe de contexto, adicionado os valores pelo objeto modelo com uso do método *Add*, e por fim, o método que efetiva a gravação dos dados. Veja o exemplo:

```
public void Inserir(TipoProduto tipoProduto)
{
```

```
// Cria a classe de contexto
DataBaseContext ctx = new DataBaseContext();

// Adiciona o objeto preenchido pelo usuário
ctx.TipoProduto.Add(tipoProduto);

// Salva as alterações
ctx.SaveChanges();
}
```

Código-fonte 4.5 – *Entity Framework* – Added
Fonte: Elaborado pelo autor (2018)

4.3.3 Modified – Update

Segue o exemplo para remoção de um objeto da base de dados. Essa operação necessita alterar o estado do registro para **modified** e depois efetivar a transação. Segue o código-fonte que apresenta o método alterar na estratégia do EF:

```
public void Alterar(TipoProduto tipoProduto)
{
    // Informa o contexto que um objeto foi alterado
    context.TipoProduto.Update(tipoProduto);

    // Salva as alterações
    context.SaveChanges();
}
```

Código fonte 4.6 – *Entity Framework* – Modified
Fonte: Elaborado pelo autor (2018)

4.3.4 Delete

Seguindo a linha do Update, a forma de exclusão altera o estado do objeto e efetiva a alteração, porém, antes de tentar efetuar a exclusão, será necessário criar uma instância da classe model e associar o Id de exclusão. Veja abaixo o exemplo de exclusão no método excluir da classe **TipoProdutoRepository**.

```
public void Excluir(int id)
{
    // Criar um tipo produto apenas com o Id
    var tipoProduto = new TipoProduto()
    {
        IdTipo = id
    };

    context.TipoProduto.Remove(tipoProduto);
    context.SaveChanges();
}
```

Código-fonte 4.7 – *Entity Framework* – Deleted
Fonte: Elaborado pelo autor (2018)

4.3.5 Find

O método **Find** será o responsável por recuperar os dados de um registro, consultando por meio de um Id. Veja o código abaixo:

```
public TipoProduto Consultar(int id)
{
    // Cria a classe de contexto
    DataBaseContext ctx = new DataBaseContext();

    // Recuperando o objeto TipoProduto de um determinado Id
    TipoProduto tipoProduto = ctx.TipoProduto.Find(id);

    return tipoProduto;
}
```

Código-fonte 4.8 – *Entity Framework* – Find
Fonte: Elaborado pelo autor (2018)

4.3.6 List

Nossa última operação básica é a operação que recupera todos os registros de tipos de produto, o famoso **SELECT *** será substituído por um simples método do EF.

O método de listagem requisita a importação do *namespace* **System.Linq**. Veja abaixo o exemplo de listagem de dados:

```
public IList<TipoProduto> Listar()
{
    IList<TipoProduto> lista = new List<TipoProduto>();

    // Cria a classe de contexto
    DataBaseContext ctx = new DataBaseContext();

    // Efetuando a listagem (Substituindo o Select *)
    lista = ctx.TipoProduto.ToList<TipoProduto>();

    // Retorna a lista
    return lista;
}
```

Código-fonte 4.9 – *Entity Framework* – List
Fonte: Elaborado pelo autor (2018)

4.3.7 Todas as Operações

Segue o código-fonte final da classe de acesso a dados:

```
using FiapSmartCity.Models;
using FiapSmartCity.Repository.Context;
using System.Collections.Generic;
using System.Linq;

namespace FiapSmartCity.Repository
{
    public class TipoProdutoRepository
    {
        // Propriedade que terá a instância do DataBaseContext
        private readonly DataBaseContext context;

        public TipoProdutoRepository()
        {
            // Criando um instância da classe de contexto do
EntityFramework
            context = new DataBaseContext();
        }

        public IList<TipoProduto> Listar()
        {
            return context.TipoProduto.ToList();
        }

        public TipoProduto Consultar(int id)
        {
            return context.TipoProduto.Find(id);
        }

        public void Inserir(TipoProduto tipoProduto)
        {
            context.TipoProduto.Add(tipoProduto);
            context.SaveChanges();
        }

        public void Alterar(TipoProduto tipoProduto)
        {
            context.TipoProduto.Update(tipoProduto);
            context.SaveChanges();
        }

        public void Excluir(int id)
    }
}
```



```
{  
    // Criar um tipo produto apenas com o Id  
    var tipoProduto = new TipoProduto()  
    {  
        IdTipo = id  
    };  
  
    context.TipoProduto.Remove(tipoProduto);  
    context.SaveChanges();  
}  
  
}
```

Código-fonte 4.10 – Implementação classe TipoProdutoRepository com EF
Fonte: Elaborado pelo autor (2018)

4.4 Operações Avançadas

Até essa seção, foram apresentados os recursos básicos do *Entity Framework*, recursos esses que possibilitam executar as operações de *CRUD* em uma aplicação com banco de dados.

As operações de *Insert*, *Update* e *Delete* não possuem forma avançada, pois sempre são executadas em um registro. Já as operações de consulta podem possuir formas complexas para o retorno de dados, por exemplo, o tratamento de relacionamento, ligações entre dois ou mais dados do sistema e filtro de informações.

Este capítulo apresentará exemplos de como executar consultas que incluam recursos de filtro, ordenação e ligação com mais entidades.

4.5 LINQ

LINQ é o acrônimo de *Language Integrated Query*, foi adicionado ao *.NET framework* versão 3.5 (ano de 2008) nas linguagens C# e VB.Net com o objetivo de efetuar consulta em diversas fontes de dados com uma sintaxe unificada, teve sua inspiração na linguagem SQL. É formada por um conjunto de métodos chamados operadores de consulta, tipos anônimos e expressões lambda.

Abaixo, segue uma lista de possíveis exemplos de uso:

- Filtrar informações em vetores.

- Filtrar informações em lista do tipo `IEnumerable<T>`.
- Consultar dados em arquivos XML.
- Gerenciar dados relacionais com objetos.
- Consulta de objetos do tipo `DataSet`.

4.6 OrderBy

O primeiro exemplo apresentado será para ordenar pela descrição os tipos de produto do site FiapSmartCity. Será usado um *Extension Method* **OrderBy** e uma expressão lambda para definir qual atributo será usado para ordenar.

Abaixo, segue o código de ordenação ascendente e decendente:

```
public IList<TipoProduto> ListarOrdenadoPorNome ()
{
    var lista =
        context.TipoProduto.OrderBy (t                =>
t.DescricaoTipo) .ToList<TipoProduto> () ;

    return lista;
}
```

Código-fonte 4.11 – OrderBy com LINQ
Fonte: Elaborado pelo autor (2018)

```
public IList<TipoProduto> ListarOrdenadoPorNomeDecendente ()
{
    var lista =
        context.TipoProduto.OrderByDescending (t          =>
t.DescricaoTipo) .ToList<TipoProduto> () ;

    return lista;
}
```

Código-fonte 4.12 – OrderBy decendente com LINQ
Fonte: Elaborado pelo autor (2018)

4.7 Where

Vamos usar novamente *Extension Method* **OrderBy** e uma expressão lambda, mas o objetivo agora é filtrar informações da lista. A sugestão para esse exemplo é exibir na lista apenas os tipos que são comercializados. Segue o exemplo:

```
public IList<TipoProduto> ListarTiposComercializados()
{
    // Filtro com Where
    var lista =
        context.TipoProduto.Where(t => t.Comercializado ==
'0')
            .OrderByDescending(t
t.DescricaoTipo).ToList<TipoProduto>();

    return lista;
}
```

Código-fonte 4.13 – Cláusula *Where* com LINQ
Fonte: Elaborado pelo autor (2018)

No próximo exemplo, o método recebe como parâmetro o valor do filtro para o campo comercializado, veja:

```
public IList<TipoProduto>
ListarTiposComercializadosCritério(char selecao)
{
    // Filtro com Where
    var lista =
        context.TipoProduto.Where(t => t.Comercializado ==
selecao)
            .OrderByDescending(t
t.DescricaoTipo).ToList<TipoProduto>();

    return lista;
}
```

Código-fonte 4.14 – Cláusula *Where* parametrizada variável comercializada com LINQ
Fonte: Elaborado pelo autor (2018)

Outro exemplo de **Where**, uma consulta com dois campos como filtro. Segue:

```
public IList<TipoProduto> ListarTiposComercializados(char
selecao)
{
    // Filtro com Where
    var lista =
        context.TipoProduto.Where(t => t.Comercializado ==
selecao && t.IdTipo > 2)
            .OrderByDescending(t
t.DescricaoTipo).ToList<TipoProduto>();

    return lista;
}
```

Código-fonte 4.15 – Cláusula *Where* parametrizada variável idTipo com LINQ
Fonte: Elaborado pelo autor (2018)

E para finalizar, um exemplo de consulta que retorna apenas uma linha, semelhante ao método **Find** do EF. Veja:

```
public TipoProduto ConsultaPorDescricao(string descricao)
{
    // Retorno único
    TipoProduto tipo =
        context.TipoProduto.Where(t => t.DescricaoTipo ==
descricao)
        .FirstOrDefault<TipoProduto>();

    return tipo;
}
```

Código-fonte 4.16 – Cláusula *Where* parametrizada variável descrição com LINQ
Fonte: Elaborado pelo autor (2018)

4.8 Contains

Veja o exemplo de filtro por parte da descrição do tipo de produto:

```
public IList<TipoProduto> ListarTiposParteDescricao(string
parteDescricao)
{
    // Filtro com Where e Contains
    var lista =
        context.TipoProduto.Where(t =>
        t.DescricaoTipo.Contains(parteDescricao))
        .ToList<TipoProduto>();

    return lista;
}
```

Código-fonte 4.17 – Cláusula *Where* e método *Contains* com LINQ
Fonte: Elaborado pelo autor (2018)

4.9 Relacionamentos

Chegamos ao ponto de avançarmos nossas pesquisas fazendo ligações entre duas informações relacionadas. Até aqui trabalhamos apenas com uma entidade que foi a **TipoProduto**, assim, para seguirmos, será necessária a criação de uma nova entidade.

A nova entidade receberá o nome de **Produto** e será associada ao **TipoProduto**, pois cada produto deve ser qualificado com um tipo. O diagrama abaixo apresenta essa ligação:

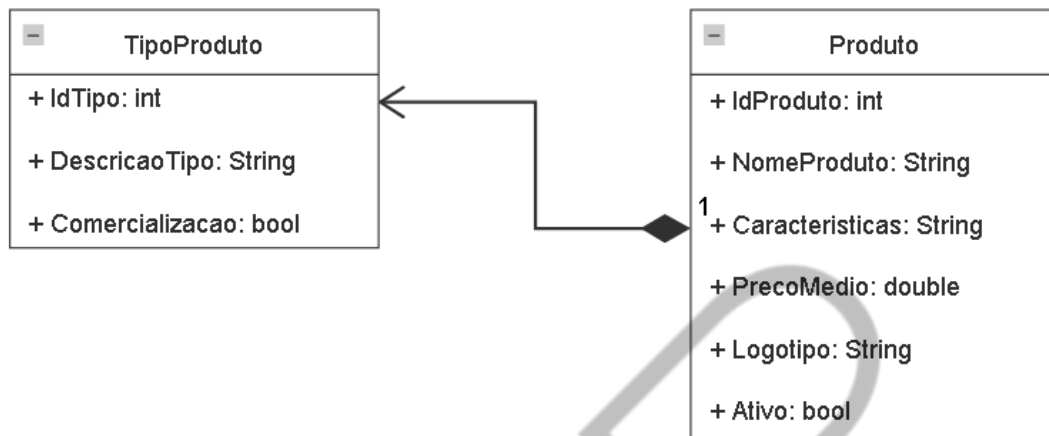


Figura 4.4 – Diagrama de classe - Produto e Categoria
Fonte: Elaborado pelo autor (2018)

Como estamos usando a estratégia de **Database First**, é necessária a criação da tabela no banco de dados. O código abaixo apresenta o script Oracle para criação da tabela e a chave estrangeira para a tabela de tipo de produto. Segue:

```
CREATE TABLE PRODUTO (
    IDPRODUTO          NUMBER          PRIMARY KEY,
    NOMEPRODUTO        VARCHAR(70)     NOT NULL,
    CARACTERISTICAS    VARCHAR(100)    NOT NULL,
    PRECOMEDIO         NUMBER,
    LOGOTIPO           VARCHAR(200)    NOT NULL,
    ATIVO              INT,
    IDTIPO             NUMBER,
    CONSTRAINT FK_IDTIPO FOREIGN KEY (IDTIPO) REFERENCES
    TIPOPRODUTO(IDTIPO)
);

CREATE SEQUENCE PRODUTO_IDPRODUTO_SEQ;

CREATE OR REPLACE TRIGGER TR_SEQ_CONTATO BEFORE INSERT ON
PRODUTO FOR EACH ROW
BEGIN
    SELECT PRODUTO_IDPRODUTO_SEQ.NEXTVAL
    INTO :new.IDPRODUTO
    FROM dual;
END;
```

```
/*
--DROP TRIGGER TR_SEQ_CONTATO;
--DROP SEQUENCE PRODUTO_IDPRODUTO_SEQ;
--DROP TABLE PRODUTO
SELECT * FROM PRODUTO;
SELECT * FROM TIPOPRODUTO;

INSERT INTO PRODUTO (NOMEPRODUTO, CARACTERISTICAS,
PRECOMEDIO, LOGOTIPO, ATIVO, IDTIPO )
VALUES ('Produto Tinta 1','Caracteristicas',
2000, 'http://www.logo.com.br/tinta1', 1, 1);
INSERT INTO PRODUTO (NOMEPRODUTO, CARACTERISTICAS,
PRECOMEDIO, LOGOTIPO, ATIVO, IDTIPO )
VALUES ('Produto Tinta 2','Caracteristicas',
1000, 'http://www.logo.com.br/tinta2', 0, 1);
INSERT INTO PRODUTO (NOMEPRODUTO, CARACTERISTICAS,
PRECOMEDIO, LOGOTIPO, ATIVO, IDTIPO )
VALUES ('Produto Agua 1','Caracteristicas', 10,
'http://www.logo.com.br/agua', 1, 7);
INSERT INTO PRODUTO (NOMEPRODUTO, CARACTERISTICAS,
PRECOMEDIO, LOGOTIPO, ATIVO, IDTIPO )
VALUES ('Produto Agua 2','Caracteristicas', 20,
'http://www.logo.com.br/agua2', 0, 7);
COMMIT;
*/
```

Código-fonte 4.18 – Script Oracle criação da tabela de produto
Fonte: Elaborado pelo autor (2018)

IMPORTANTE: O código no final de script é para auxiliar, caso a recriação da tabela seja necessária. Também existem algumas linhas para inserir produtos, verifique a chave usada para o Tipo de Produto.

Tabela criada, precisamos anotar nosso modelo e criar o **DbSet** para a manipulação. As anotações usadas são as mesmas do exemplo anterior: [Table], [Key] e [Column], mas a classe de modelo terá duas particularidades, são elas:

- **(Foreing Key)** – Atributo que representa a chave estrangeira, será mapeado como uma coluna.
- **(Navigation Property)** – Atributo que representa o modelo da tabela relacionada, e não receberá nenhuma anotação.

Veja no código a seguir a versão final da classe de modelo **Produto**:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
```

```
namespace FiapSmartCity.Models
{
    [Table("PRODUTO")]
    public class Produto
    {
        [Key]
        [Column("IDPRODUTO")]
        public int IdProduto { get; set; }

        [Column("NOMEPRODUTO")]
        public String NomeProduto { get; set; }

        [Column("CARACTERISTICAS")]
        public String Caracteristicas { get; set; }

        [Column("PRECOMEDIO")]
        public double PrecoMedio { get; set; }

        [Column("LOGOTIPO")]
        public String Logotipo { get; set; }

        [Column("ATIVO")]
        public bool Ativo { get; set; }

        //Foreing Key
        [Column("IDTIPO")]
        public int IdTipo { get; set; }

        //Navigation Property
        public TipoProduto Tipo { get; set; }
    }
}
```

Código-fonte 4.19 – Anotações EF para a classe modelo Produto
Fonte: Elaborado pelo autor (2018)

Em nossa classe de contexto (DataBaseContext) é preciso adicionar a propriedade **DbSet** para o modelo produto. Veja:

```
public DbSet<Produto> Produto { get; set; }
```

Código-fonte 4.20 – DbSet Produto
Fonte: Elaborado pelo autor (2018)

4.10 Relacionamento um para um

O relacionamento um para um será o primeiro demonstrado como exemplo. O objetivo será buscar um produto em nossa base e, em seu retorno, trazer os dados do tipo a que está associado (Id e descrição). Como estamos falando do domínio “Produto”, para organizar nossa aplicação vamos adicionar uma nova classe no *namespace* Repository, o nome será **ProdutoRepository** e a partir dela vamos adicionar os métodos para buscas avançadas.

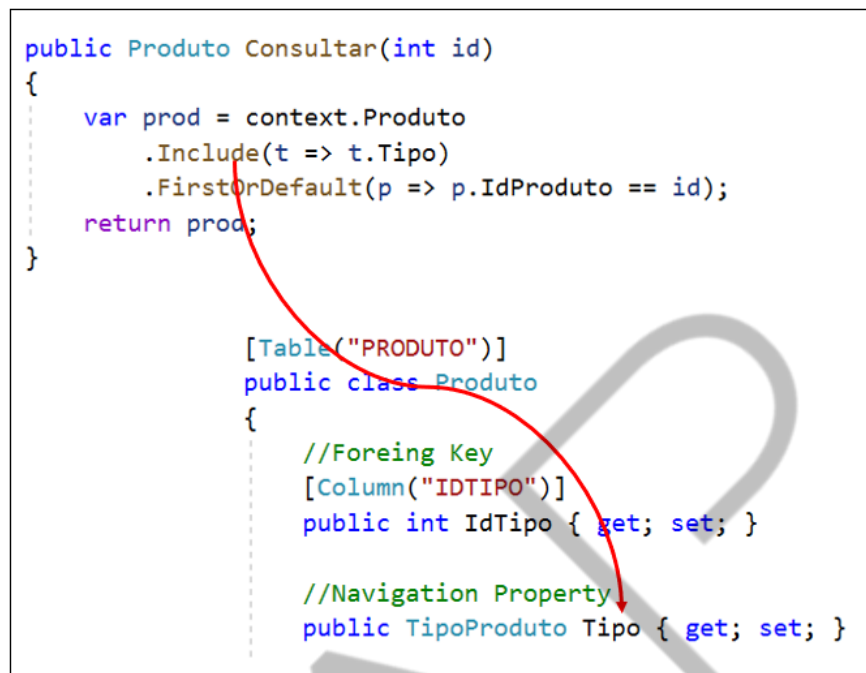
A busca do produto será feita pelo método a partir de dois *Extension Methods*. O primeiro é o método **Include**, que recebe como parâmetro o nome da *Navigation Property* declarado no modelo. O segundo é o método **FirstOrDefault**, responsável por filtrar o produto com o identificador desejado.

Segue o código com o método para busca do produto:

```
public Produto Consultar(int id)
{
    var prod = context.Produto.Include(t => t.Tipo)
                              .FirstOrDefault(p => p.IdProduto == id);
    return prod;
}
```

Código-fonte 4.21 – Relacionamento um para um usando Include
Fonte: Elaborado pelo autor (2018)

Veja na figura a ligação entre as entidades com o método Include, o nome passado como parâmetro para o método é o nome do atributo definido para ser a *Navigation Property*.



```
public Produto Consultar(int id)
{
    var prod = context.Produto
        .Include(t => t.Tipo)
        .FirstOrDefault(p => p.IdProduto == id);
    return prod;
}

[Table("PRODUTO")]
public class Produto
{
    //Foreign Key
    [Column("IDTIPO")]
    public int IdTipo { get; set; }

    //Navigation Property
    public TipoProduto Tipo { get; set; }
}
```

Figura 4.5 – *Extension Method – Include*
Fonte: Elaborado pelo autor (2018)

Para validar a consulta, pode ser feita uma chamada do método **BuscarPorId** e com a opção **QuickWatch** do *Debug* é possível verificar o conteúdo do objeto Produto retornado na consulta. Veja a Resultado do método Include:

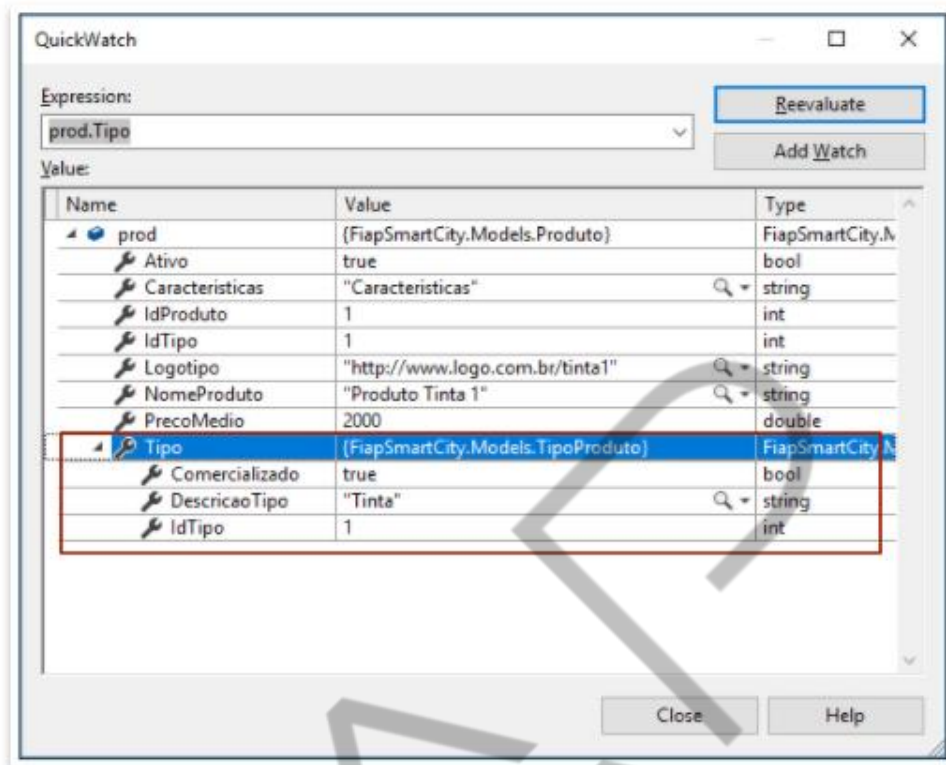


Figura 4.6 – Resultado do método Include
Fonte: Elaborado pelo autor (2018)

4.11 Relacionamento um para muitos

Para representar esse relacionamento, vamos fazer o processo inverso, assim, dado um tipo de produto, vamos recuperar todos os produtos associados. O código para executar essa operação é semelhante ao anterior, ou seja, devemos usar o método Include para recuperar a lista produto.

Porém, antes de implementar o código para recuperar as informações, faz-se necessário criar uma *Navigation Property* na classe **TipoProduto** que será uma lista de elementos *Produto*. Veja o código-fonte da classe **Models\TipoProduto**:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace FiapSmartCity.Models
```

```
{
    [Table("TIPOPRODUTO")]
    public class TipoProduto
    {
        [Key]
        [Column("IDTIPO")]
        public int IdTipo { get; set; }

        [Required(ErrorMessage= "Descrição obrigatória!")]
        [StringLength(50, ErrorMessage = "A descrição deve ter no máximo 50 caracteres")]
        [Display(Name="Descrição:")]
        [Column("DESCRICAOTIPO")]
        public String DescricaoTipo { get; set; }

        [Column("COMERCIALIZADO")]
        public bool Comercializado { get; set; }

        //Navigation Property
        public IList<Produto> Produtos { get; set; }
    }
}
```

Código-fonte 4.22 – Navigation Property para a lista de produtos
Fonte: Elaborado pelo autor (2018)

Agora podemos implementar nossa consulta. Vamos declarar o método **ListarProdutosPorTipo** na classe **ProdutoRepository**, que fará uma consulta a uma entidade tipo de produto, com o método **Include** devemos adicionar a entidade de produto para conseguir operar o relacionamento entre os elementos.

Segue a implementação do método:

```
public IList<Produto> ConsultarProdutosPorTipo(int idTipo)
{
    // Consulta a lista de produtos de um determinado tipo.
    var tipoProduto =
        context.TipoProduto
            .Include(t => t.Produtos)
            .FirstOrDefault(t => t.IdTipo == idTipo);

    return tipoProduto.Produtos;
}
```

Código-fonte 4.23 – Relacionamento um para muitos usando Include
Fonte: Elaborado pelo autor (2018)

A figura “Resultado da lista do relacionamento um para muitos” apresenta a janela **QuickWatch** da execução do método **ListarProdutosPorTipo** com o conteúdo da lista produtos preenchidos, veja:

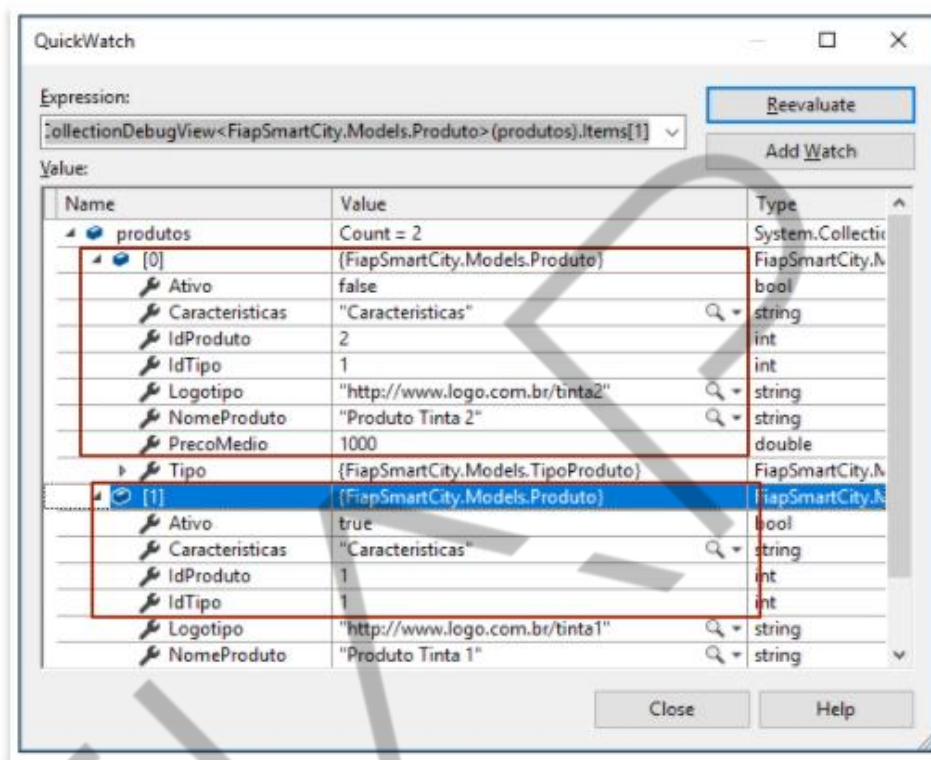


Figura 4.7 – Resultado da lista do relacionamento um para muitos

Fonte: Elaborado pelo autor (2018)

DICA: O LINQ e o *Entity Framework* permitem a inserção de vários *Extensions Methods* para filtrar, ordenar, incluir ligações entre outros em conjuntos. É possível adicionar métodos duplicados para atingir o objetivo de sua consulta. Por exemplo: o uso de dois (2) métodos includes na mesma linha de código.

4.12 Considerações finais

Neste capítulo, foi apresentado o framework ORM para a tecnologia .NET conhecido por *Entity Framework* ou EF. Por meio da aplicação **FiapSmartCity**, foi possível refatorar o código ADO.NET, trocando comandos SQL e chamadas das bibliotecas do ADO.NET por recursos do EF.

O *Entity Framework* declara como uma de suas principais vantagens a redução das linhas de código, a simplificação de comandos e a remoção de códigos SQL em programas.

AVANADE

REFERÊNCIAS

ARAÚJO, E. C. **Orientação a Objetos em C# - Conceitos e implementações em .NET**. São Paulo: Casa do Código, 2017.

ARAÚJO, E. C. **ASP.NET MVC5 Crie aplicações web na plataforma Microsoft**. São Paulo: Casa do Código, 2017.

DYKSTRA, T.; ANDERSON, R. **Getting Started with Entity Framework 6 Code First using MVC5**, Microsoft, 2014.

Entity Framework 6. Disponível em: <<https://docs.microsoft.com/pt-br/ef/ef6/>>. Acesso em: 10 fev. 2018.

Entity Framework Code First to a New Database. Disponível em: <[https://msdn.microsoft.com/en-us/library/jj193542\(v=vs.113\).aspx](https://msdn.microsoft.com/en-us/library/jj193542(v=vs.113).aspx)>. Acesso em: 10 fev. 2018.

Entity Framework Model First. Disponível em: <[https://msdn.microsoft.com/en-us/library/jj205424\(v=vs.113\).aspx](https://msdn.microsoft.com/en-us/library/jj205424(v=vs.113).aspx)>. Acesso em: 15 fev. 2018.

Implementando a funcionalidade básica CRUD com o Entity Framework no aplicativo ASP.NET MVC. Disponível em: <<https://docs.microsoft.com/pt-br/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-application>>. Acesso em: 15 fev. 2018.

SANCHEZ, F.; ALTHMANN, M. F. **Desenvolvimento web com ASP.NET MVC**. São Paulo: Casa do Código, 2013.