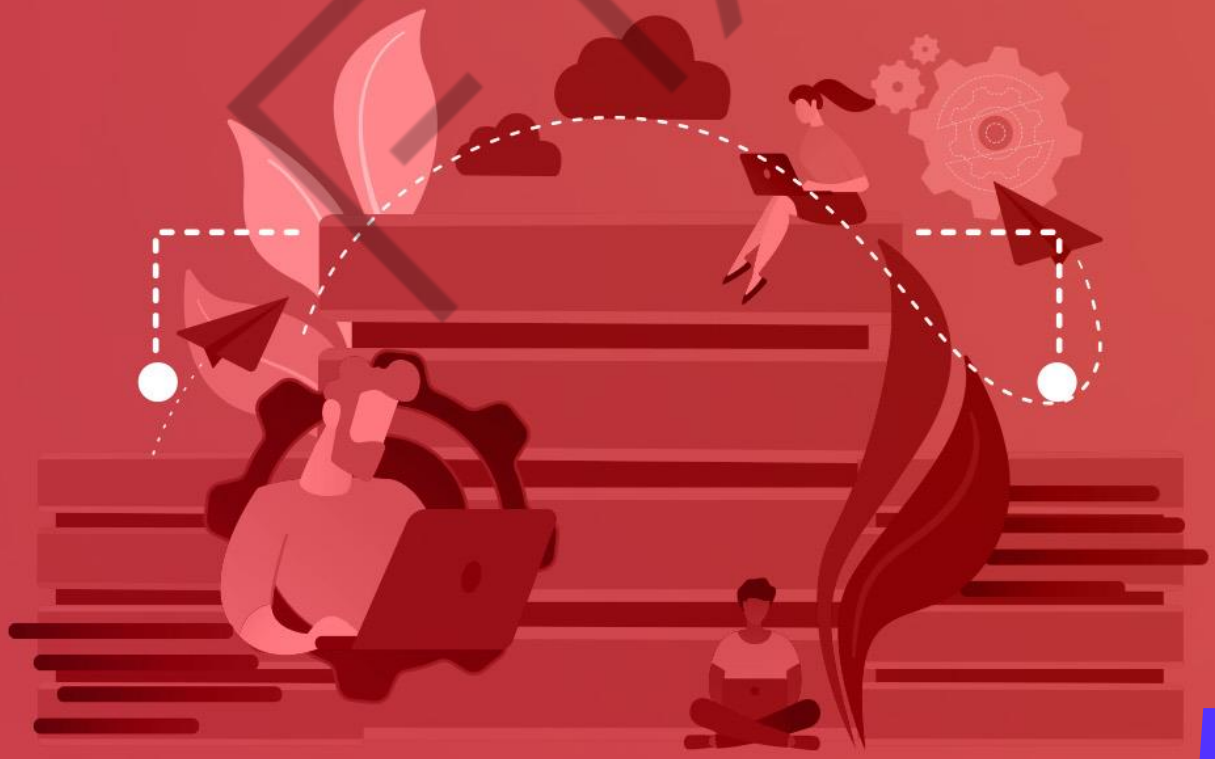
FIAP ON

FRAMEWORKS JAVA, .NET & WEBSERVICES

E O C# AGORA **ACESSA O BANCO SEM SOFRIMENTO**



5B

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

1 E O C# AGORA ACESSA O BANCO SEM SOFRIMENTO

1.1 Entity Framework Core

O *Entity Framework* (EF) é um conjunto de tecnologias ADO.NET que permite ao usuário mapear seus objetos de modelos com entidade de banco de dados. O EF pode ser comparado 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 criação de camadas de conexões robustas e até mesmo sem a necessidade de instruções SQL. Sua primeira versão foi lançada em meados de 2008, fazendo parte do .NET 3.5 SP1 e do Visual Studio 2008. A partir do EF4.1, o framework já começa a ser enviado ao NuGet.org, sendo um dos pacotes mais baixados da atualidade.

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

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

Lembrando que há 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 **Fiap.Web.AspNet**. 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.

Iniciaremos pela instalação da biblioteca do EF para Oracle, acesse o *Nuget*, faça uma busca por Oracle e selecione a instalação da biblioteca **Oracle.EntityFrameworkCore**. Selecione a versão 7.2.x. Veja a Figura “Instalando o Oracle.EntityFrameworkCore” abaixo:

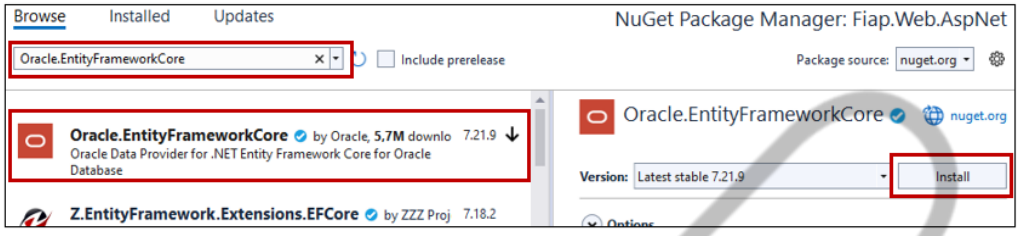


Figura 1 – Instalando o Oracle.EntityFrameworkCore
Fonte: Elaborado pelo autor (2022)

Não é necessário efetuar nenhuma alteração no arquivo **appsettings.json**, 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 **Representante**, seguiremos, portanto, usando essa tabela.

Como reutilizaremos essa tabela criada anteriormente, teremos um problema no uso do EF, porém isso servirá como parte do aprendizado. O Oracle e o EF são *case-sensitive*, ou seja, diferenciam maiúscula e minúscula, note que a tabela **Representante** possui seu nome e seus 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.

Caso a tabela e sua estrutura tenham sido criadas de maneira diferente dos scripts ou tenham sofrido alguma modificação, utilize o script para a criação da tabela **Representante** no banco de dados Oracle:

E o C# agora acessa o banco sem sofrimento

```
CREATE TABLE REPRESENTANTE (
        REPRESENTANTEID NUMBER PRIMARY KEY,
        NOMEREPRESENTANTE VARCHAR2(80) NOT NULL,
        CPF VARCHAR2(12) NOT NULL
);

CREATE SEQUENCE REPRESENTANTE_ID_SEQ;

CREATE OR REPLACE TRIGGER TR_SEQ_REPRESENTANTE BEFORE INSERT ON REPRESENTANTE FOR EACH
ROW
BEGIN
        SELECT REPRESENTANTE_ID_SEQ.NEXTVAL INTO :new.REPRESENTANTEID FROM dual;
END;

--DROP TRIGGER TR_SEQ_REPRESENTANTE;
--DROP SEQUENCE REPRESENTANTE_ID_SEQ;
--DROP TABLE REPRESENTANTE;
```

Código-fonte 1 – Script para a criação da tabela Representante
Fonte: Elaborado pelo autor (2022)

Tudo pronto para a implementação!

1.2 Implementação EF Core

1.2.1 Classe de Contexto

Para que a 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 como responsabilidade interagir com os objetos e com o banco de dados.

Dentro do *namespace* **Repository**, adicione uma pasta com o nome **Context e**, em seguida, adicione uma classe com o nome de **DataBaseContext**. Veja a Figura "Classe DataBaseContext":

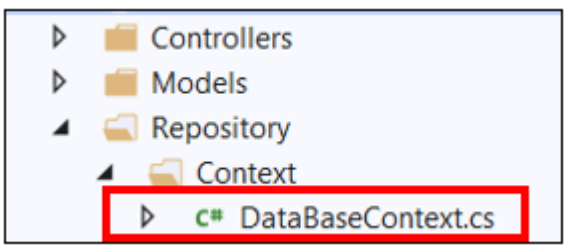


Figura 2 – Classe DataBaseContext
Fonte: Elaborado pelo autor (2022)

Agora, é necessário declarar a classe **DataBaseContext** como uma subclasse de **System.Data.Entity.DbContext**. Em seguida, vamos sobrescrever os dois construtores herdados da superclasse **DbContext**.

O Código-Fonte "Implementação DataBaseContext" apresenta a implementação da classe **DataBaseContext**:

```
using Microsoft.EntityFrameworkCore;

namespace Fiap.Web.AspNet.Repository.Context
{
   public class DataBaseContext : DbContext
   {

      public DataBaseContext(DbContextOptions options) : base(options)
      {
      }

      protected DataBaseContext()
      {
      }

   }
}
```

Código-fonte 2 – Implementação DataBaseContext
Fonte: Elaborado pelo autor (2022)

Com nossa classe de contexto criada e declarada, precisamos configurar o projeto para gerenciar as instâncias e conexões com o banco de dados. Esse passo é feito na classe **Program.cs** e tem como objetivo obter a string de conexão com o banco, especificar qual banco de dados e drive vamos utilizar e deixar disponível a conexão disponível para o uso em qualquer classe do projeto. O Código-Fonte "Configurando e instanciando o DataBaseContext" demonstra a implementação citada anteriormente:

```
using Fiap.Web.AspNet.Repository.Context;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

var connectionString = builder.Configuration.GetConnectionString("DatabaseConnection");
builder.Services.AddDbContext<DataBaseContext>(options =>
```

E o C# agora acessa o banco sem sofrimento

```
    options.UseOracle(connectionString).EnableSensitiveDataLogging(true)
);


// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
}
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Código-fonte 3 – Configurando e instanciando o DataBaseContext
Fonte: Elaborado pelo autor (2022)

1.2.2 DbSet

Vimos, anteriormente, a classe de contexto e algumas configurações particulares para nosso exemplo. Mas ainda precisamos incluir alguns itens na classe de contexto (**DataBaseContext**) 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 **DataBaseContext**, é necessário declarar uma propriedade do tipo de **DbSet** para representar a entidade de **Representante**. Veja o Código-Fonte “Criando o DbSet para Representante" a seguir:

E o C# agora acessa o banco sem sofrimento

```
using Fiap.Web.AspNet.Models;
using Microsoft.EntityFrameworkCore;

namespace Fiap.Web.AspNet.Repository.Context
{
   public class DataBaseContext : DbContext
   {


      // Propriedade que será responsável pelo acesso a tabela de Representantes
      public DbSet<RepresentanteModel> Representante { get; set; }



      public DataBaseContext(DbContextOptions options) : base(options)
      {
      }

      protected DataBaseContext()
      {
      }

   }
}
```

Código-fonte 4 – Criando o DbSet para Representante
Fonte: Elaborado pelo autor (2022)

1.2.3 Models e anotações

Agora, é preciso deixar nosso modelo vinculado à nossa tabela. Para isso, usaremos algumas anotações disponíveis nos *namespaces:* **System.ComponentModel.DataAnnotations** e **System.ComponentModel.DataAnnotations.Schema**. São três 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 muito utilizadas quando trabalhamos com 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.

E o C# agora acessa o banco sem sofrimento

O código apresenta a classe **RepresentanteModel** com as anotações para o funcionamento do EF. Veja o Código-Fonte “Modelo Representante com anotações do EF”:

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

namespace Fiap.Web.AspNet.Models
{

   [Table("REPRESENTANTE")]
   public class RepresentanteModel
   {


      [Key]
      [Column("REPRESENTANTEID")]
      public int RepresentanteId { get; set; }

      [Column("NOMEREPRESENTANTE")]
      [Required(ErrorMessage = "Nome do representante é obrigatório!")]
      [StringLength(80,
         MinimumLength = 2,
         ErrorMessage = "O nome deve ter, no mínimo, 2 e, no máximo, 80 caracteres")]
      [Display(Name = "Nome do Representante")]
      public string? NomeRepresentante { get; set; }

      [Column("CPF")]
      [Required(ErrorMessage = "CPF é obrigatório!")]
      [Display(Name = "CPF")]
      public string? Cpf { get; set; }

      // Essa anotação é apenas um exemplo de um propriedade não mapeada em uma coluna do banco
de dados
      [NotMapped]
      public string? Token { get; set; }

      public RepresentanteModel()
      {

      }

      public RepresentanteModel(int representanteId, string nomeRepresentante)
      {
         RepresentanteId = representanteId;
         NomeRepresentante = nomeRepresentante;
      }

      public RepresentanteModel(int representanteId, string cpf, string nomeRepresentante)
      {
         RepresentanteId = representanteId;
         Cpf = cpf;
         NomeRepresentante = nomeRepresentante;
```

E o C# agora acessa o banco sem sofrimento

```
    }
  }
}
```

Código-fonte 5 – Modelo Representante com anotações do EF
Fonte: Elaborado pelo autor (2022)

1.3 Outras anotações

O *Entity Framework* disponibiliza outras anotações além das abordadas 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
- ForeingKey
- InverseProperty

1.3.1 Operações

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

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

Temos cinco 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:

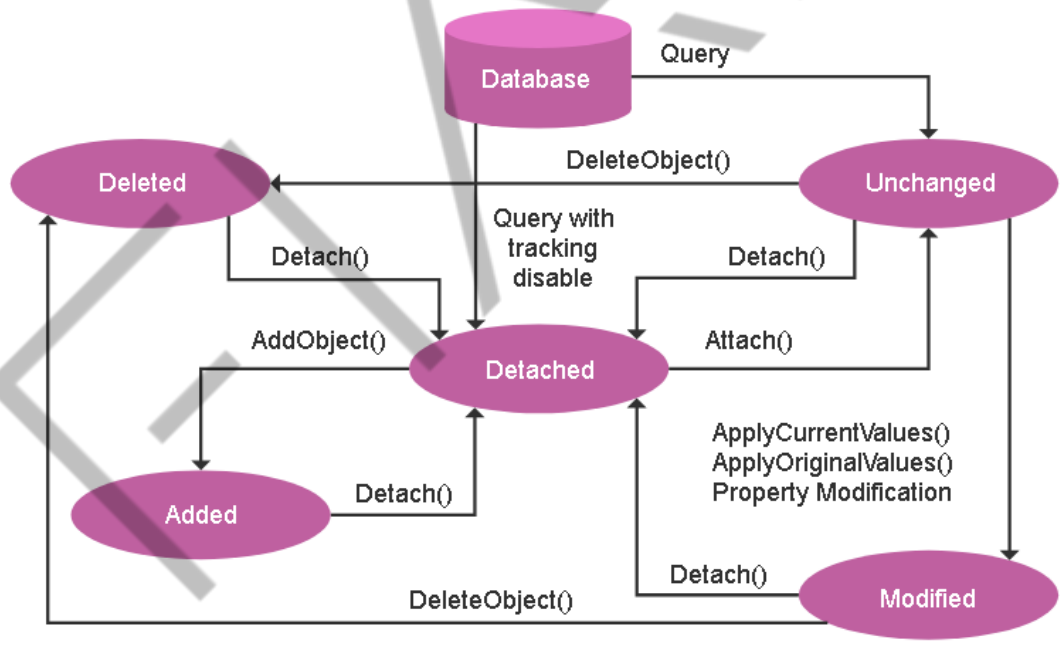


Figura 3 – Estado das entidades do EF
Fonte: Google Imagens (2022)

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

E o C# agora acessa o banco sem sofrimento

Um a um será convertido 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.

1.3.2 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, adicionados os valores pelo objeto modelo com uso do método *Add* e, por fim, o método que efetiva a gravação dos dados. Veja o exemplo no Código-Fonte "*Entity Framework* – Added":

```
public void Inserir(RepresentanteModel representante)
{
   // a propriedade dataBaseContext é declarada no escopo da classe
   // e sua instância é recebida pelo construtor da classe Repository
   // veja na solução final da classe RepresentanteRepository

   // Adiciona o objeto preenchido pelo usuário
   dataBaseContext.Representante.Add(representante);

   // Salva as alterações
   dataBaseContext.SaveChanges();

}
```

Código-fonte 6 – *Entity Framework* – Added
Fonte: Elaborado pelo autor (2022)

1.3.3 Modified – Update

Segue o exemplo para alteração 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 (*Entity Framework* – Modified) que apresenta o método alterar na estratégia do EF:

```
public void Alterar(RepresentanteModel representante)
{
   dataBaseContext.Representante.Update(representante);

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

Código-fonte 7 – *Entity Framework* – Modified
Fonte: Elaborado pelo autor (2022)

1.3.4 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, no Código-Fonte "*Entity Framework* – Deleted", o exemplo de exclusão no método excluir da classe **RepresentanteRepository**.

```
public void Excluir(int id)
{
   var representante = new RepresentanteModel(id,"");

   dataBaseContext.Representante.Remove(representante);

   // Salva as alterações
   dataBaseContext.SaveChanges();

}
```

Código-fonte 8 – *Entity Framework* – Deleted
Fonte: Elaborado pelo autor (2022)

1.3.5 Find

O método **Find** será o responsável por recuperar os dados de um registro, consultando-os por meio de um Id. Veja o Código-Fonte "*Entity Framework* – Find" abaixo:

```
public RepresentanteModel Consultar(int id)
{
   // Recuperando o objeto Representante de um determinado Id
   var representante = dataBaseContext.Representante.Find(id);

   return representante;
}
```

Código-fonte 9 – *Entity Framework* – Find
Fonte: Elaborado pelo autor (2022)

E o C# agora acessa o banco sem sofrimento

1.3.6 List

Nossa última operação básica é a 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, no Código-Fonte "*Entity Framework* – List", o exemplo de listagem de dados:

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

   // Efetuando a listagem (Substituindo o Select *)
   lista = dataBaseContext.Representante.ToList<RepresentanteModel>();

   return lista;
}
```

Código-fonte 10 – *Entity Framework* – List
Fonte: Elaborado pelo autor (2022)

1.3.7 Todas as operações

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

```
using Fiap.Web.AspNet.Models;
using Fiap.Web.AspNet.Repository.Context;

namespace Fiap.Web.AspNet.Repository
{
   public class RepresentanteRepository
   {

      private readonly DataBaseContext dataBaseContext;

      public RepresentanteRepository(DataBaseContext ctx)
      {
         dataBaseContext = ctx;
      }



      public IList<RepresentanteModel> Listar()
      {
         var lista = new List<RepresentanteModel>();

         // Efetuando a listagem (Substituindo o Select *)
         lista = dataBaseContext.Representante.ToList<RepresentanteModel>();

         return lista;
```

E o C# agora acessa o banco sem sofrimento

```
        }

        public RepresentanteModel Consultar(int id)
        {
            // Recuperando o objeto Representante de um determinado Id
            var representante = dataBaseContext.Representante.Find(id);

            return representante;
        }

        public void Inserir(RepresentanteModel representante)
        {
            // Adiciona o objeto preenchido pelo usuário
            dataBaseContext.Representante.Add(representante);

            // Salva as alterações
            dataBaseContext.SaveChanges();

        }

        public void Alterar(RepresentanteModel representante)
        {
            dataBaseContext.Representante.Update(representante);

            // Salva as alterações
            dataBaseContext.SaveChanges();
        }

        public void Excluir(int id)
        {
            var representante = new RepresentanteModel(id,"");

            dataBaseContext.Representante.Remove(representante);

            // Salva as alterações
            dataBaseContext.SaveChanges();

        }


    }
}
```

Código-fonte 11 – Implementação da classe RepresentanteRepository com EF
Fonte: Elaborado pelo autor (2022)

IMPORTANTE: Como a classe de contexto do EF será usada em todos os métodos da classe **Repository**, ela foi declarada um atributo private com o nome dataBaseContext, e, no construtor da classe **RepresentanteRepository**, é instanciado de fato o contexto do EF.

E o C# agora acessa o banco sem sofrimento

1.3.6 Testando

Acredito que, quando chegado a esse ponto, tenha sido notado um erro de compilação na classe **RepresenteteController** e **ClienteController** na linha que efetua a instância do nosso repositório. O erro ocorre,pois modificamos o construtor padrão da classe **RepresentanteRepository** para aceitar um parâmetro do tipo **DataBaseContext.** Assim, é necessário efetuar uma modificação em nosso controller para gerar uma instância do DataBaseContext, que será repassada no construtor do nosso repositório. O código-fonte "Gerando a instância do contexto no *Controller* de Representante" apresenta como devemos solucionar esse problema. Veja:

```
using Fiap.Web.AspNet.Controllers.Filters;
using Fiap.Web.AspNet.Models;
using Fiap.Web.AspNet.Repository;
using Fiap.Web.AspNet.Repository.Context;
using Microsoft.AspNetCore.Mvc;


namespace Fiap.Web.AspNet.Controllers
{
   public class RepresentanteController : Controller
   {

      private RepresentanteRepository representanteRepository;

      // O parametro enviado no construtor do Controller é gerenciado pelo próprio framework .NET
      // Esse recurso é chamada de Injeção de Dependência
      public RepresentanteController(DataBaseContext dataBaseContext)
      {

         representanteRepository = new RepresentanteRepository(dataBaseContext);
      }
```

Código-fonte 12 – Gerando a instância do contexto no *Controller* de Representante
Fonte: Elaborado pelo autor (2022)

Use o exemplo do *Controller* de representante para corrigir o mesmo problema na classe ***ClienteController***.

Você pode baixar a solução até esse ponto de desenvolvimento nos links abaixo:

Git: https://github.com/FIAPON/Fiap.Web.AspNet/tree/orm

E o C# agora acessa o banco sem sofrimento

Zip: https://github.com/FIAPON/Fiap.Web.AspNet/archive/refs/heads/orm.zip

.

E o C# agora acessa o banco sem sofrimento

2 OPERAÇÕES AVANÇADAS

Até esta seção, foram apresentados os recursos básicos do *Entity Framework*, 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 apresentar 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.

2.1 LINQ

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

A seguir, temos 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.

E o C# agora acessa o banco sem sofrimento

2.2 OrderBy

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

Abaixo, segue o código-fonte de ordenação ascendente e descendente:

```
public IList<RepresentanteModel> ListarOrdenadoPorNome()
{
   var lista = new List<RepresentanteModel>();

   lista = dataBaseContext
      .Representante
      .OrderBy( r => r.NomeRepresentante).ToList<RepresentanteModel>();

   return lista;
}
```

Código-fonte 13 – OrderBy com LINQ
Fonte: Elaborado pelo autor (2022)

```
public IList<RepresentanteModel> ListarOrdenadoPorNomeDescendente()
{
   var lista = new List<RepresentanteModel>();

   lista = dataBaseContext
      .Representante
      .OrderByDescending(r => r.NomeRepresentante).ToList<RepresentanteModel>();

   return lista;
}
```

Código-fonte 14 – OrderBy descendente com LINQ
Fonte: Elaborado pelo autor (2022)

2.3 Where

Usaremos novamente um *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 registros de uma entidade qualquer que tenha um atributo específico com o valor verdadeiro. Segue o exemplo:

```
public IList<ExemploModel> ListarFiltroExemplo()
{
   // Filtro com Where
   var lista =
```

E o C# agora acessa o banco sem sofrimento

```
            dataBaseContext
            .ExemploModel
            .Where( e => e.PropriedadeBoolean == true )
            .OrderByDescending(e => e.NomePropriede).ToList<ExemploModel>();

        return lista;
    }
```

Código-fonte 15 – Cláusula *Where* com LINQ
Fonte: Elaborado pelo autor (2022)

IMPORTANTE: o código-fonte "Cláusula Where com LINQ" é apenas um exemplo e não funcionará no projeto desse material.

No próximo exemplo, o método recebe como parâmetro o valor do filtro para o a propriedade a ser filtrada. Veja o Código-Fonte "Cláusula *Where* parametrizada variável comercializada com LINQ" abaixo:

```
    public IList<ExemploModel> ListarFiltroExemplo(bool selecao)
    {
        // Filtro com Where
        var lista =
            dataBaseContext
            .ExemploModel
            .Where(e => e.PropriedadeBoolean == selecao)
            .OrderByDescending(e => e.NomePropriede).ToList<ExemploModel>();

        return lista;
    }
```

Código-fonte 16 – Cláusula *Where* parametrizada variável com parâmetro com LINQ
Fonte: Elaborado pelo autor (2022)

Outro exemplo de ***Where é*** uma consulta com dois campos como filtro. Veja o Código-Fonte "Cláusula *Where* com duas propriedades LINQ" abaixo:

```
    public IList<ExemploModel> ListarFiltroExemplo(bool selecao)
    {
        // Filtro com Where
        var lista =
            dataBaseContext
            .ExemploModel
            .Where(e =>
                    e.PropriedadeBoolean == selecao &&
                    e.PropriedadeInt > 10
                )
            .OrderByDescending(e => e.NomePropriede).ToList<ExemploModel>();

        return lista;
```

E o C# agora acessa o banco sem sofrimento

```
}
```

Código-fonte 17 – Cláusula *Where* com duas propriedades LINQ
Fonte: Elaborado pelo autor (2022)

E, para finalizar, um exemplo de consulta que retorna **apenas uma linha**, semelhante ao método ***Find*** do EF. Veja o Código-Fonte "Cláusula *Where* parametrizada variável CPF com LINQ" abaixo:

```
public RepresentanteModel ConsultarPorCpf(String cpf)
{
   var representante = dataBaseContext.Representante.
         Where(r => r.Cpf == cpf).
            FirstOrDefault<RepresentanteModel>();

   return representante;
}
```

Código-fonte 18 – Cláusula *Where* parametrizada variável CPF com LINQ
Fonte: Elaborado pelo autor (2022)

2.4 Contains

Veja o exemplo de filtro para uma busca por parte do nome de um representante Código-Fonte "Cláusula *Where* e método *Contains* com LINQ", esse exemplo tem a mesma funcionalidade do comando LIKE '%%' da linguagem SQL:

```
public RepresentanteModel ConsultarPorParteNome(String nomeParcial)
{
   var representante = dataBaseContext.Representante.
         Where(r => r.NomeRepresentante.Contains(nomeParcial)).
            FirstOrDefault<RepresentanteModel>();

   return representante;
}
```

Código-fonte 19 – Cláusula *Where* e método *Contains* com LINQ
Fonte: Elaborado pelo autor (2022)

2.5 Relacionamentos

Chegamos ao ponto de avançarmos nossas pesquisas fazendo ligações entre duas informações relacionadas. Até aqui, trabalhamos apenas com uma entidade, que foi a **Representante**, porém, temos uma outra entidade em nosso projeto denominada Cliente, que poderíamos aplicar todos os conceitos do EF em sua classe Model e seu

E o C# agora acessa o banco sem sofrimento

Repositório se não fosse o fato dessa entidade ter um relacionamento de 1 para muitos com a entidade Representante.

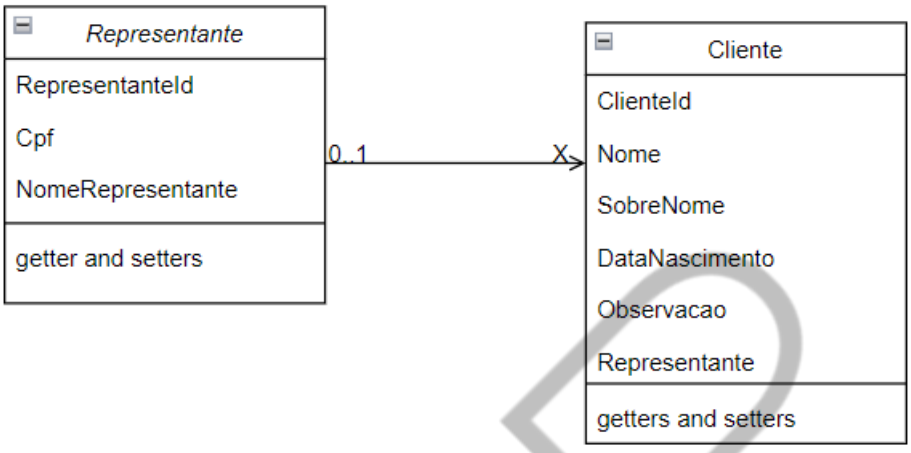


Figura 4 – Diagrama de classe – Cliente e Representante
Fonte: Elaborado pelo autor (2022)

Caso você não tenha criado a tabela de Clientes, segue o Código-Fonte “Script Oracle para a criação da tabela de Cliente” Veja:

```
CREATE TABLE CLIENTE (
        CLIENTEID NUMBER PRIMARY KEY,
        NOME VARCHAR2(80) NOT NULL,
        DATANASCIMENTO DATE,
   OBSERVACAO VARCHAR2(80) NOT NULL,
   REPRESENTANTEID NUMBER NOT NULL,
   CONSTRAINT FK_REPRESENTANTE
      FOREIGN KEY (REPRESENTANTEID)
      REFERENCES REPRESENTANTE(REPRESENTANTEID)
);


CREATE SEQUENCE CLIENTE_ID_SEQ;

CREATE OR REPLACE TRIGGER TR_SEQ_CLIENTE BEFORE INSERT ON CLIENTE FOR EACH ROW
BEGIN
        SELECT CLIENTE_ID_SEQ.NEXTVAL INTO :new.CLIENTEID FROM dual;
END;
```

Código-fonte 20 – Script Oracle para a criação da tabela de Cliente
Fonte: Elaborado pelo autor (2012)

IMPORTANTE: O código no final do script é para auxiliar caso a recriação da tabela seja necessária.

Com a 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-Fonte “Anotações EF para a classe modelo Cliente”, a versão final da classe de modelo **ClienteModel**:

```
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Fiap.Web.AspNet.Models
{
   [Table("CLIENTE")]
   public class ClienteModel
   {
      [HiddenInput]
      [Key]
      [Column("CLIENTEID")]
      public int ClienteId { get; set; }

      [Display(Name = "Nome do Cliente")]
      [Required(ErrorMessage = "O nome do cliente é obrigatório")]
      [MaxLength(70, ErrorMessage = "O tamanho máximo para o campo nome é de 70 caracteres.")]
      [MinLength(2, ErrorMessage = "Digite um nome com 2 ou mais caracteres")]
      [Column("NOME")]
      public string? Nome { get; set; }


      [Display(Name = "Data de Nascimento")]
      [Required(ErrorMessage = "Data de nascimento é obrigatório")]
      [DataType(DataType.Date, ErrorMessage = "Data de nascimento incorreta")]
      [Column("DATANASCIMENTO")]
      public DateTime DataNascimento { get; set; }

      [Display(Name = "Observação")]
      [Column("OBSERVACAO")]
      public string? Observacao { get; set; }


      //Campo de chave Estrangeira - Foreign Key (FK)
      [Display(Name = "Representante")]
      [Column("REPRESENTANTEID")]
      public int RepresentanteId { get; set; }
```

E o C# agora acessa o banco sem sofrimento

```
        //Objeto - Navigation Object
        public RepresentanteModel? Representante { get; set; }

    }
}
```

Código-fonte 21 – Anotações EF para a classe modelo Cliente
Fonte: Elaborado pelo autor (2022)

Em nossa classe de contexto (**DataBaseContext**) é preciso adicionar a propriedade **DbSet** para o modelo produto. Veja o Código-Fonte “DbSet Produto”:

```
        // Propriedade que será responsável pelo acesso a tabela de Representantes
        public DbSet<RepresentanteModel> Representante { get; set; }

        // Propriedade que será responsável pelo acesso a tabela de Cliente
        public DbSet<ClienteModel> Cliente { get; set; }
```

Código-fonte 22 – DbSet Cliente
Fonte: Elaborado pelo autor (2022)

2.6 Relacionamento um para um

O relacionamento um para um será o primeiro demonstrado como exemplo. O objetivo aqui é buscar um cliente em nossa base e, em seu retorno, trazer os dados do representante a que está associado (Id e Nome). A implementação da classe ClienteRepository já apresenta essa funcionalidade usando os recursos do ADO.NET e do comando SQL Inner Join. A figura “Comparação entre comando entre relacionamento ADO.NET versus EF” apresenta o mesmo método escrito no formato mais tradicional versus a forma com o EntityFramework, veja:

E o C# agora acessa o banco sem sofrimento

Figura 5 – Comparação entre comando entre relacionamento ADO.NET versus EF
Fonte: Elaborado pelo autor (2022)

A busca do cliente 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 (Representante). O segundo é o método **FirstOrDefault**, responsável por filtrar o cliente com o identificador desejado.

Veja, na Figura "*Extension Method* – Include", a ligação entre as entidades com o método Include. O nome passado como parâmetro para o método é o do atributo definido para ser a *Navigation Property*.

E o C# agora acessa o banco sem sofrimento

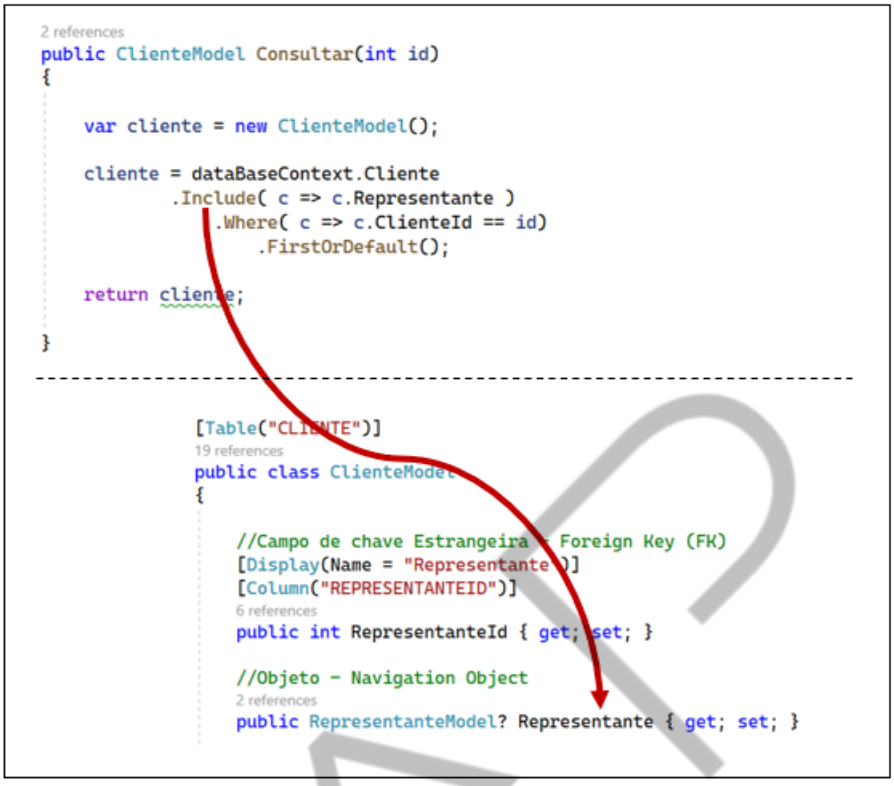


Figura 6 – *Extension Method* – Include
Fonte: Elaborado pelo autor (2022)

Para colocar tudo isso em prática, temos alguns ajustes para fazer na classe **ClienteRepository** e na classe **ClienteController**. Esses ajustes são semelhantes aos que fizemos no **RepresentanteRepository**, que modificamos seu construtor para aceitar o objeto **DataBaseContext** e ajustamos todos os métodos que usavam o ADO.Net para executar as operações de banco de dados. O código-fonte "ClienteRepository usando EF e relacionamento" apresenta as alterações, com destaque aos métodos listar e consultar que apresentam o uso do comando Include, com o objetivo de executar uma consulta com dados relacionados entre duas tabelas.

```
using Fiap.Web.AspNet.Models;
using Fiap.Web.AspNet.Repository.Context;
using Microsoft.EntityFrameworkCore;
using System.Data;

namespace Fiap.Web.AspNet.Repository
{
   public class ClienteRepository
   {

      private readonly DataBaseContext dataBaseContext;
```

E o C# agora acessa o banco sem sofrimento

```
        public ClienteRepository(DataBaseContext ctx)
        {
            dataBaseContext = ctx;
        }


        public IList<ClienteModel> Listar()
        {
            var lista = dataBaseContext.Cliente
                    .Include(c => c.Representante)
                        .ToList<ClienteModel>();

            return lista;

        }


        public ClienteModel Consultar(int id)
        {

            var cliente = new ClienteModel();

            cliente = dataBaseContext.Cliente
                    .Include( c => c.Representante )
                        .Where( c => c.ClienteId == id)
                            .FirstOrDefault();

            return cliente;

        }

        public void Inserir(ClienteModel cliente)
        {
            dataBaseContext.Cliente.Add(cliente);
            dataBaseContext.SaveChanges();
        }


        public void Alterar(ClienteModel cliente)
        {
            dataBaseContext.Cliente.Update(cliente);
            dataBaseContext.SaveChanges();
        }

        public void Excluir(int id)
        {
            var cliente = new ClienteModel { ClienteId = id };

            dataBaseContext.Cliente.Remove(cliente);
            dataBaseContext.SaveChanges();

        }

    }
}
```

E o C# agora acessa o banco sem sofrimento

Código-fonte 23 – ClienteRepository usando EF e relacionamento
Fonte: Elaborado pelo autor (2022)

O código-fonte “ClienteController ajuste” apresenta a versão de uso com os ajustes a classe de repositório de clientes.

```
using Fiap.Web.AspNet.Controllers.Filters;
using Fiap.Web.AspNet.Models;
using Fiap.Web.AspNet.Repository;
using Fiap.Web.AspNet.Repository.Context;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace Fiap.Web.AspNet.Controllers
{
    public class ClienteController : Controller
    {

        private ClienteRepository clienteRepository;
        private RepresentanteRepository representanteRepository;

        public ClienteController(DataBaseContext dataBaseContext)
        {

            clienteRepository = new ClienteRepository(dataBaseContext);
            representanteRepository = new RepresentanteRepository(dataBaseContext);
        }

        [LogFilter]
        public IActionResult Index()
        {
            // Retornando para View a lista de Clientes
            var lista = clienteRepository.Listar();

            return View(lista);
        }

        // Anotação de uso do Verb HTTP Get
        [HttpGet]
        public IActionResult Cadastrar()
        {
            // Método que carrega os dados do representante e
            // monta uma combo para exibição na tela
            ComboRepresentantes();

            return View(new ClienteModel());
```

```
    }

    // Anotação de uso do Verb HTTP Post
    [HttpPost]
    public IActionResult Cadastrar(ClienteModel cliente)
    {
        if ( ModelState.IsValid ) {

            clienteRepository.Inserir(cliente);

            TempData["mensagem"] = "Cliente cadastrado com sucesso";
            return RedirectToAction("Index", "Cliente");
        } else
        {
            // Método que carrega os dados do representante e
            // monta uma combo para exibição na tela
            ComboRepresentantes();

            return View(cliente);
        }
    }


    [HttpGet]
    public IActionResult Editar([FromRoute] int id)
    {
        // Método que carrega os dados do representante e
        // monta uma combo para exibição na tela
        ComboRepresentantes();

        var Cliente = clienteRepository.Consultar(id);
        return View(Cliente);
    }

    [HttpPost]
    public IActionResult Editar(ClienteModel cliente)
    {
        if (ModelState.IsValid)
        {
            clienteRepository.Alterar(cliente);

            TempData["mensagem"] = "Cliente alterado com sucesso";
            return RedirectToAction("Index", "Cliente");
        }
        else
        {
            // Método que carrega os dados do representante e
            // monta uma combo para exibição na tela
            ComboRepresentantes();

            return View(cliente);
        }

    }
```

E o C# agora acessa o banco sem sofrimento

```
        [HttpGet]
        public IActionResult Consultar(int id)
        {
            var Cliente = clienteRepository.Consultar(id);
            return View(Cliente);
        }

        [HttpGet]
        public IActionResult Excluir(int id)
        {
            clienteRepository.Excluir(id);

            TempData["mensagem"] = "Cliente excluído com sucesso";
            return RedirectToAction("Index", "Cliente");
        }


        private void ComboRepresentantes()
        {
            var listaRepresentantes = representanteRepository.Listar();
            var selectListRepresentantes = new SelectList(listaRepresentantes, "RepresentanteId",
"NomeRepresentante");
            ViewBag.representantes = selectListRepresentantes;
        }

    }
}
```

Código-fonte 24 – ClienteController Ajuste
Fonte: Elaborado pelo autor (2022)

Para validar a consulta de cliente com os dados do representante carregados, pode ser feita uma chamada do método **Consultar**, e, com a opção ***QuickWatch*** do *Debug,* é possível verificar o conteúdo do objeto Cliente retornado na consulta. Veja a Figura “Resultado do método Include”:

E o C# agora acessa o banco sem sofrimento

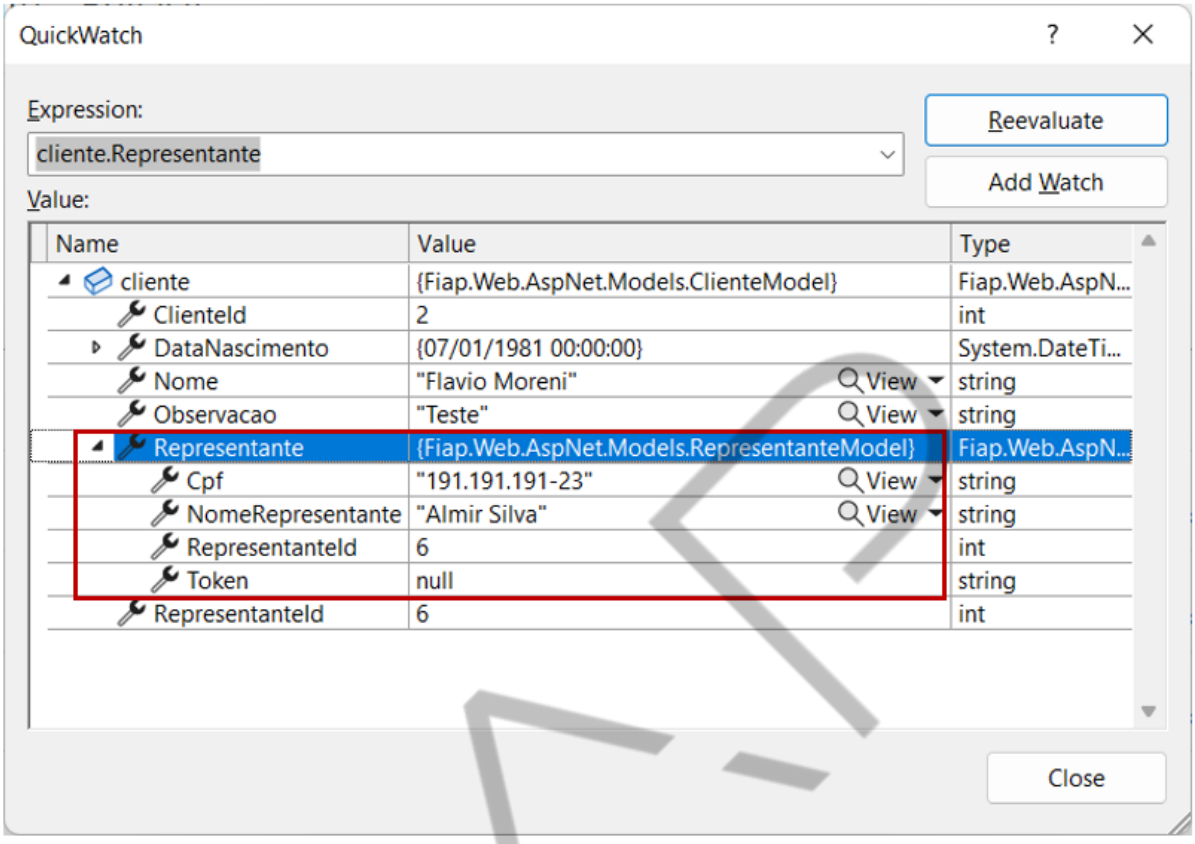


Figura 7 – Resultado do método Include
Fonte: Elaborado pelo autor (2022)

2.7 Relacionamento um para muitos

Para representar esse relacionamento, faremos o processo inverso. Dado um representante, recuperaremos todos os clientes associados. O código-fonte 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 a fim de recuperar as informações, faz-se necessário criar uma *Navigation Property* na classe **Representante** que será uma lista de elementos *Clientes*. Veja o código-fonte da classe **Models\RepresentanteModel**:

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

namespace Fiap.Web.AspNet.Models
```

```
{

    [Table("REPRESENTANTE")]
    public class RepresentanteModel
    {
        [Key]
        [Column("REPRESENTANTEID")]
        public int RepresentanteId { get; set; }

        [Column("NOMEREPRESENTANTE")]
        [Required(ErrorMessage = "Nome do representante é obrigatório!")]
        [StringLength(80,
            MinimumLength = 2,
            ErrorMessage = "O nome deve ter, no mínimo, 2 e, no máximo, 80 caracteres")]
        [Display(Name = "Nome do Representante")]
        public string? NomeRepresentante { get; set; }

        [Column("CPF")]
        [Required(ErrorMessage = "CPF é obrigatório!")]
        [Display(Name = "CPF")]
        public string? Cpf { get; set; }

        // Essa anotação é apenas um exemplo de um propriedade não mapeada em uma coluna do banco
de dados
        [NotMapped]
        public string? Token { get; set; }


         //Navegation Property
         public IList<ClienteModel> Clientes { get; set; }

        public RepresentanteModel()
        {

        }

        public RepresentanteModel(int representanteId, string nomeRepresentante)
        {
            RepresentanteId = representanteId;
            NomeRepresentante = nomeRepresentante;
        }

        public RepresentanteModel(int representanteId, string cpf, string nomeRepresentante)
        {
            RepresentanteId = representanteId;
            Cpf = cpf;
            NomeRepresentante = nomeRepresentante;
        }
    }
}
```

Código-fonte 25 – Navigation Property para a lista de clientes
Fonte: Elaborado pelo autor (2018)

E o C# agora acessa o banco sem sofrimento

Agora podemos implementar nossa consulta. Vamos declarar o método **ListarRepresentantesComClientes** na classe **RepresentanteRepository**, que fará uma consulta a uma entidade representantes. Com o método **Include**, devemos adicionar a entidade de cliente para conseguir operar o relacionamento entre os elementos.

Veja a implementação do método no Código-Fonte “Relacionamento um para muitos usando Include”:

```
public IList<RepresentanteModel> ListarRepresentantesComClientes()
{

   var lista = new List<RepresentanteModel>();

   lista = dataBaseContext.Representante
         .Include( r => r.Clientes)
            .ToList<RepresentanteModel>();

   return lista;
}
```

Código-fonte 26 – 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 **ListarRepresentantesComClientes** com o conteúdo da lista de clientes preenchidos, veja:

E o C# agora acessa o banco sem sofrimento

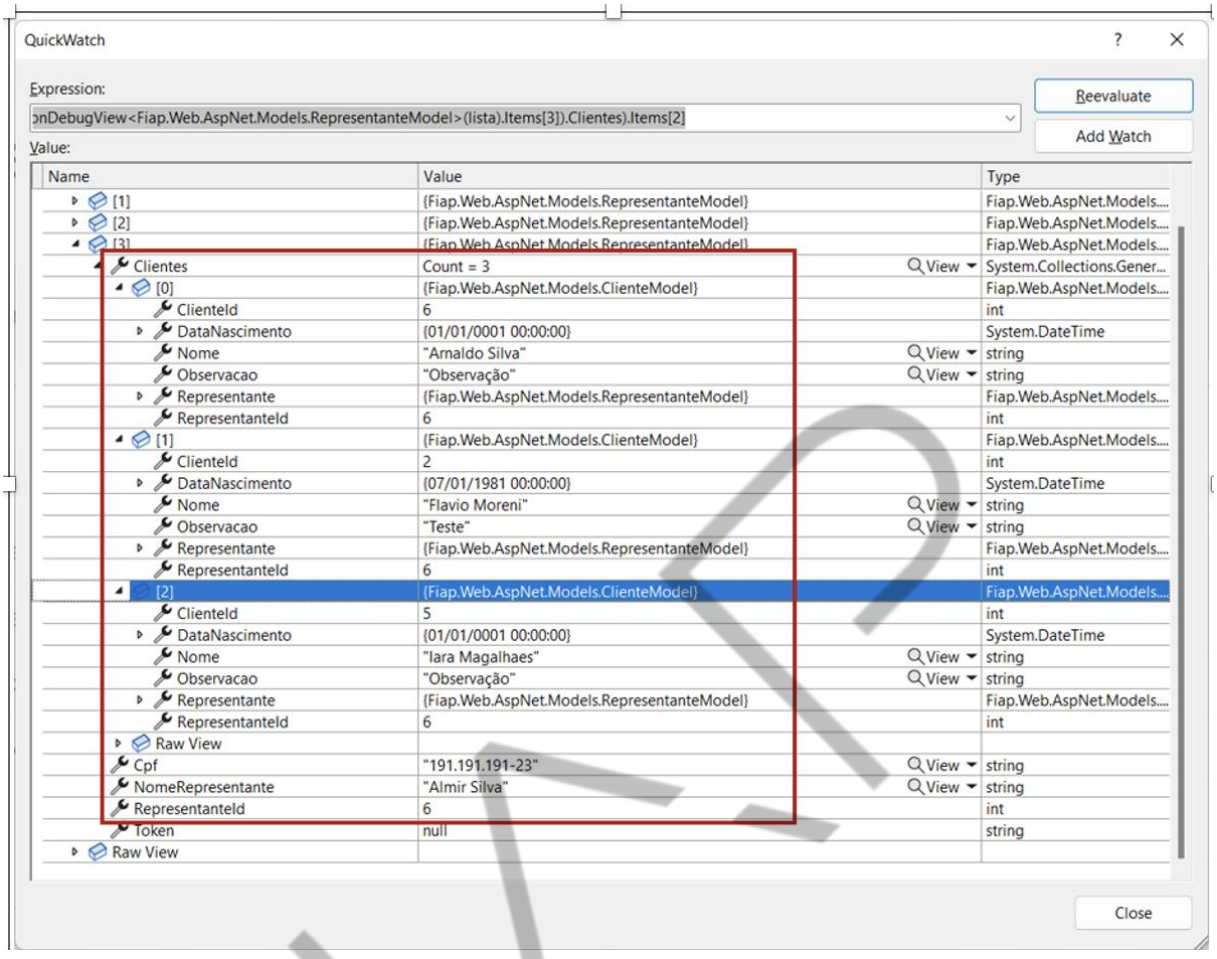


Figura 8 – Resultado da lista do relacionamento um para muitos
Fonte: Elaborado pelo autor (2022)

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 a fim de atingir o objetivo de sua consulta. Por exemplo: o uso de dois métodos “Include” na mesma linha de código.

E o C# agora acessa o banco sem sofrimento

CONCLUSÃO

Neste capítulo foi apresentado o framework ORM para a tecnologia .NET, conhecido como *Entity Framework* ou EF. Por meio da aplicação **Fiap.Web.AspNet**, 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 algumas das suas principais vantagens: a redução das linhas de código, a simplificação de comandos e a remoção de códigos SQL em programas.

Seguem os links para download da solução **Fiap.Web.AspNet**:

Git: https://github.com/FIAPON/Fiap.Web.AspNet/tree/orm-relacionamento

Zip: https://github.com/FIAPON/Fiap.Web.AspNet/archive/refs/heads/orm-relacionamento.zip

E o C# agora acessa o banco sem sofrimento

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 Core MVC** – Aplicações modernas em conjunto com o Entity Framework. São Paulo: Casa do Código, 2018.

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

MICROSOFT. **ENTITY Framework 6.** Disponível em: https://docs.microsoft.com/pt-br/ef/ef6/. Acesso em: 13 jan. 2021.

MICROSOFT. **ENTITY Framework Code First to a New Database**. Disponível em: https://msdn.microsoft.com/en-us/library/jj193542(v=vs.113).aspx. Acesso em: 13 jan. 2021.

MICROSOFT. **ENTITY Framework Model First**. Disponível em: https://msdn.microsoft.com/en-us/library/jj205424(v=vs.113).aspx. Acesso em: 13 jan. 2021.

MICROSOFT. **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: 13 jan. 2021.

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