ORM - EntityFrameWork

De começo há duas abordagens:

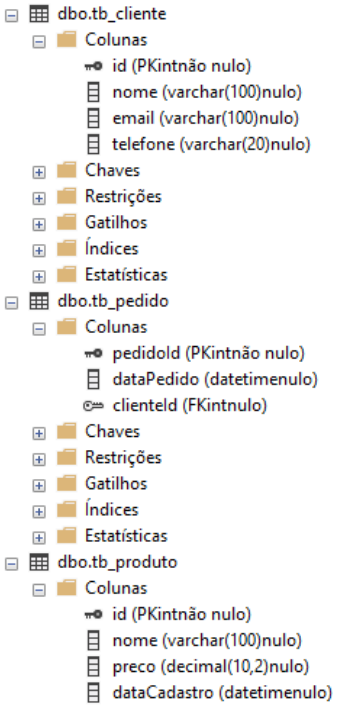
→ **Code-first**: cria-se primeiro a classe Model, para depois criar as tabelas no BD através de migration

→ **Database-first**: o banco de dados é criado e depois são criados as classes em cima do BD.

Database-first

Para os testes usaremos o seguinte script no SQLServer:

```sql
use databaseFirstCrudDemo;

CREATE TABLE tb_cliente (
    id INT PRIMARY KEY IDENTITY,
    nome VARCHAR(100),
    email VARCHAR(100),
    telefone VARCHAR(20)
);

CREATE TABLE tb_produto (
    id INT PRIMARY KEY IDENTITY,
    nome VARCHAR(100),
    preco DECIMAL(10, 2),
    dataCadastro DATETIME
);

CREATE TABLE tb_pedido (
    pedidoId INT PRIMARY KEY IDENTITY,
    dataPedido DATETIME,
    clienteId INT,
    FOREIGN KEY (clienteId) REFERENCES tb_cliente(id)
);
```

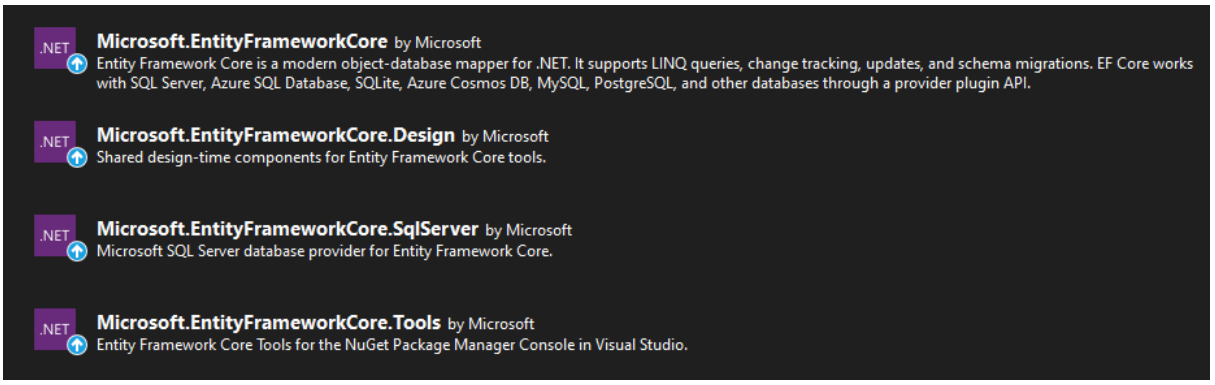


Foram criados as tabelas `tb_cliente`, `tb_produto` & `tb_pedido` no sqlserver.

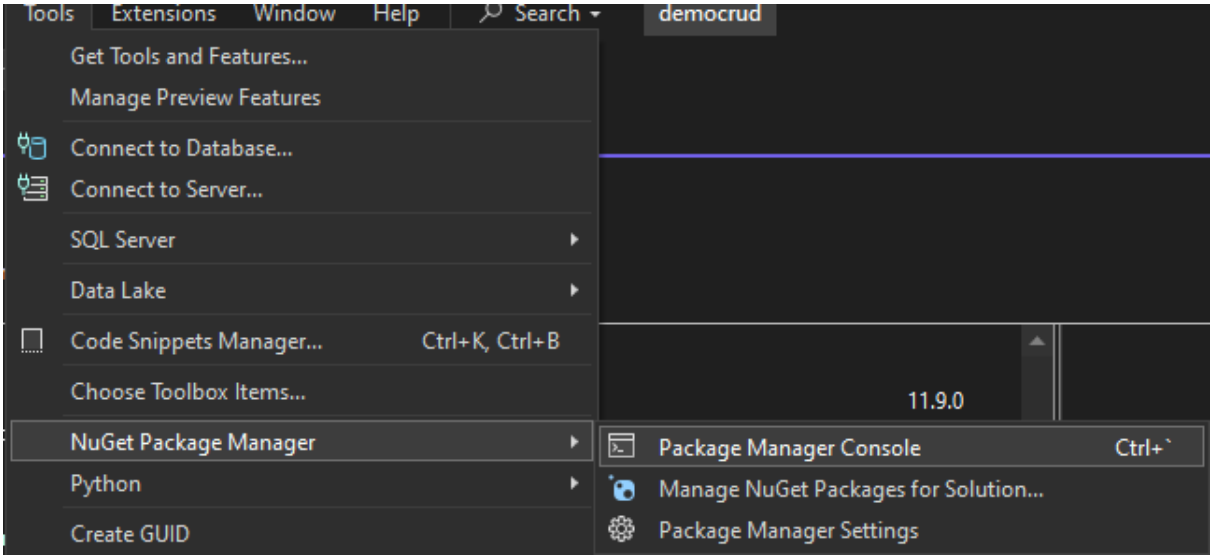
Agora indo para o código, será necessário os plugins:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Design
- Microsoft.EntityFrameworkCore.Tools

E possivelmente o Microsoft.EntityFrameworkCore.SqlServer (mas vai depender do BD utilizado)



Dentro do Visual Studio, vamos para `Tools > Nuget Package manager > Package Manager Console`



Dentro do terminal basta usar o comando `Scaffold-DbContext "SuaStringDeConexao" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models`

- “suastringdeconexao” - adicionar a string de conexão adequada
- “Microsoft.EntityFrameworkCore.SqlServer” - o provedor do BD, poderia ser mysql, postgres (`Pomelo.EntityFrameworkCore.MySql` ou

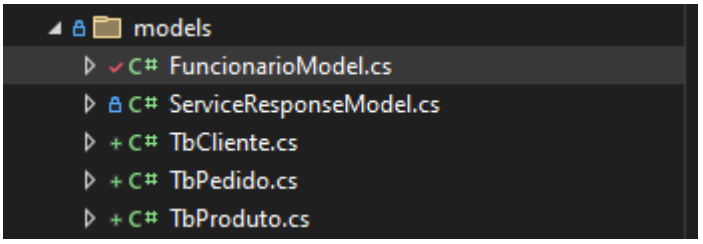
`Npgsql.EntityFrameworkCore.PostgreSQL`) ou outro.

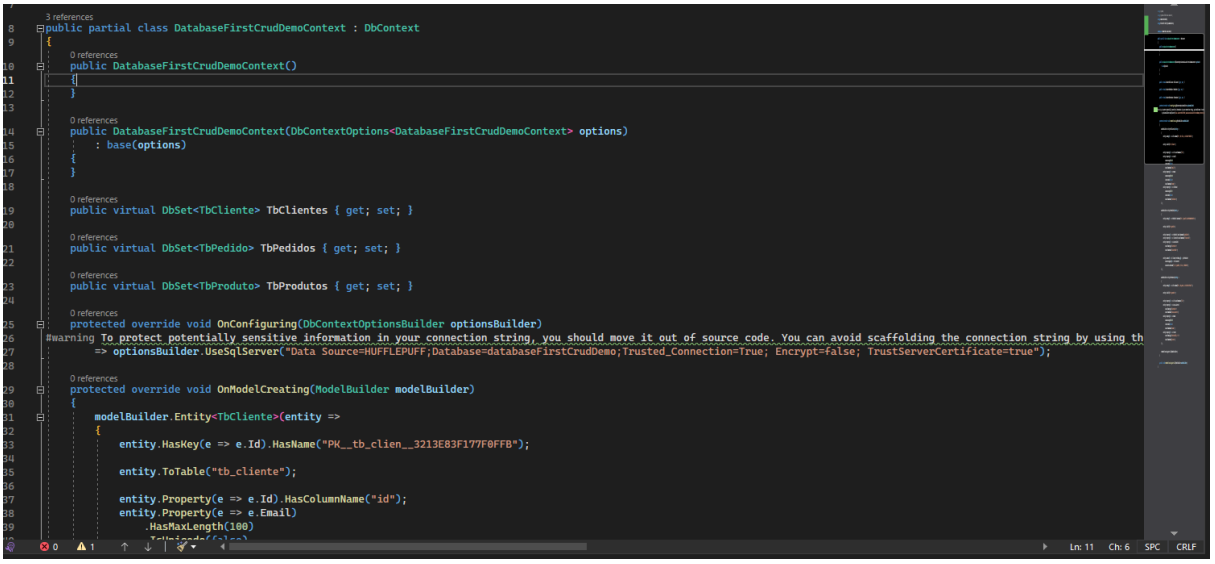
- “-OutputDir Models” - esse parâmetro é OPCIONAL, porém especifica o diretório de saída na qual as classes serão criadas, nesse caso o Models

O comando final ficou:

```
Scaffold-DbContext "Data Source=HUFFLEPUFF;Database=databaseFirstCrudDemo;Trusted_Connection=True; Encrypt=false; TrustServerCertificate=true" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

O resultado foi a criação das classes juntamente com a classe de Context configurando o banco de dados:





Agora é possível criar o CRUD em cima dessas classes.

Database-first SEM SCAFFOLD

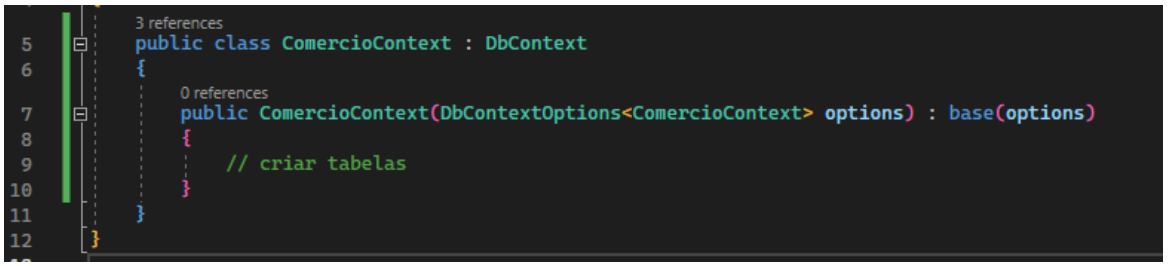
→ Criar classe de contexto

→ Criar conexão no Program.cs

→ Fazer mapeamento das tabelas dentro do contexto

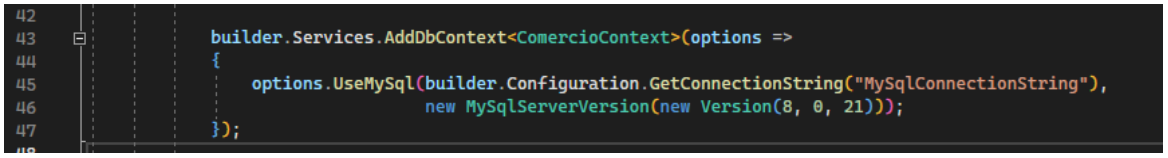
→ Criar crud

1) Criar classe de contexto



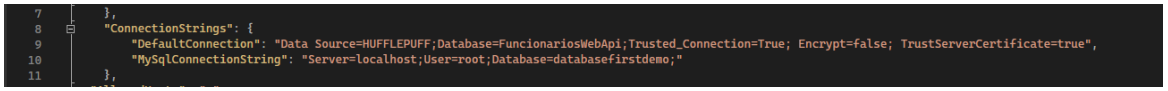
```
3 references
public class ComercioContext : DbContext
{
    0 references
    public ComercioContext(DbContextOptions<ComercioContext> options) : base(options)
    {
        // criar tabelas
    }
}
```

2) Criar conexão no Program.cs



```
builder.Services.AddDbContext<ComercioContext>(options =>
{
    options.UseMySql(builder.Configuration.GetConnectionString("MySqlConnectionString"),
                     new MySqlServerVersion(new Version(8, 0, 21)));
});
```

O trecho “MySqlConnectionString” está configurado dentro do appsettings



```
},
"ConnectionStrings": {
    "DefaultConnection": "Data Source=HUFFLEPUFF;Database=FuncionariosWebApi;Trusted_Connection=True; Encrypt=false; TrustServerCertificate=true",
    "MySqlConnectionString": "Server=localhost;User=root;Database=databasefirstdemo;"
},
```

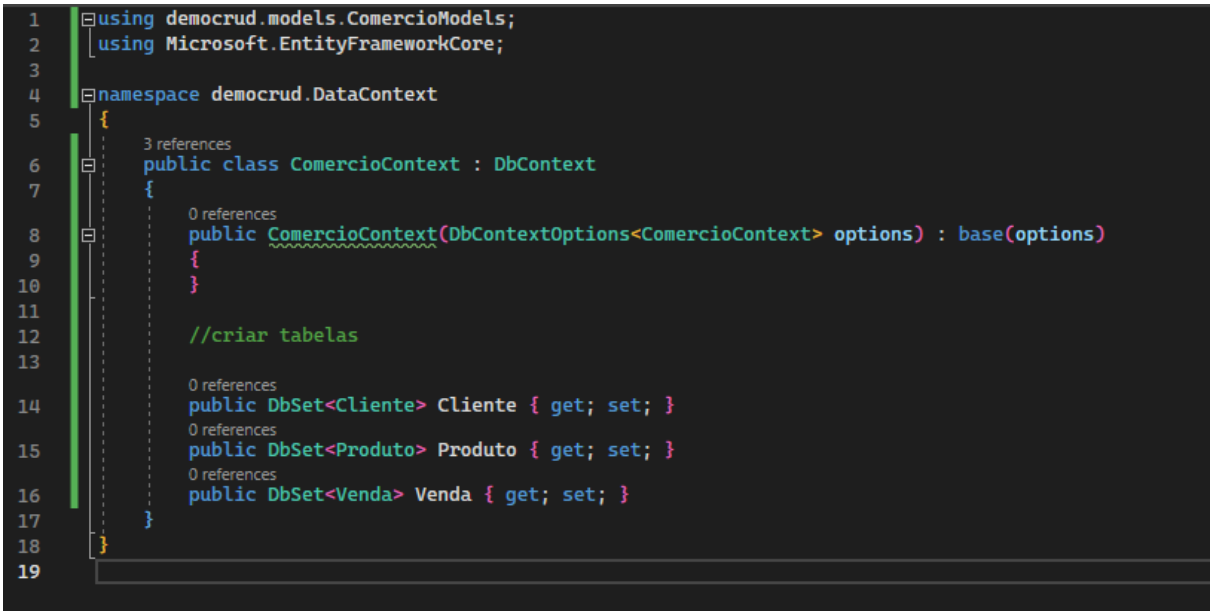
3) Mapear as tabelas

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

namespace democrud.models.ComercioModels
{
    [Table("tb_cliente")]
    0 references
    public class Cliente
    {
        0 references
        public int id {  get; set; }
        //[Column("name")]
        0 references
        public string name { get; set; }
        0 references
        public string email { get; set; }
        0 references
        public string telefone { get; set; }
    }
}
```

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

namespace democrud.models.ComercioModels
{
    [Table("tb_produto")]
    0 references
    public class Produto
    {
        0 references
        public int id {  get; set; }
        0 references
        public string nome { get; set; }
        0 references
        public decimal preco { get; set; }
    }
}
```

```cs
namespace democrud.models.ComercioModels
{
    0 references
    public class Venda
    {
        0 references
        public int id { get; set; }
        0 references
        public int cliente_id {  get; set; }
        0 references
        public int produto_id {  get; set; }
        0 references
        public DateTime dataVenda { get; set; }
    }
}
```

4) Agora referenciamos as classes dentro do Contexto



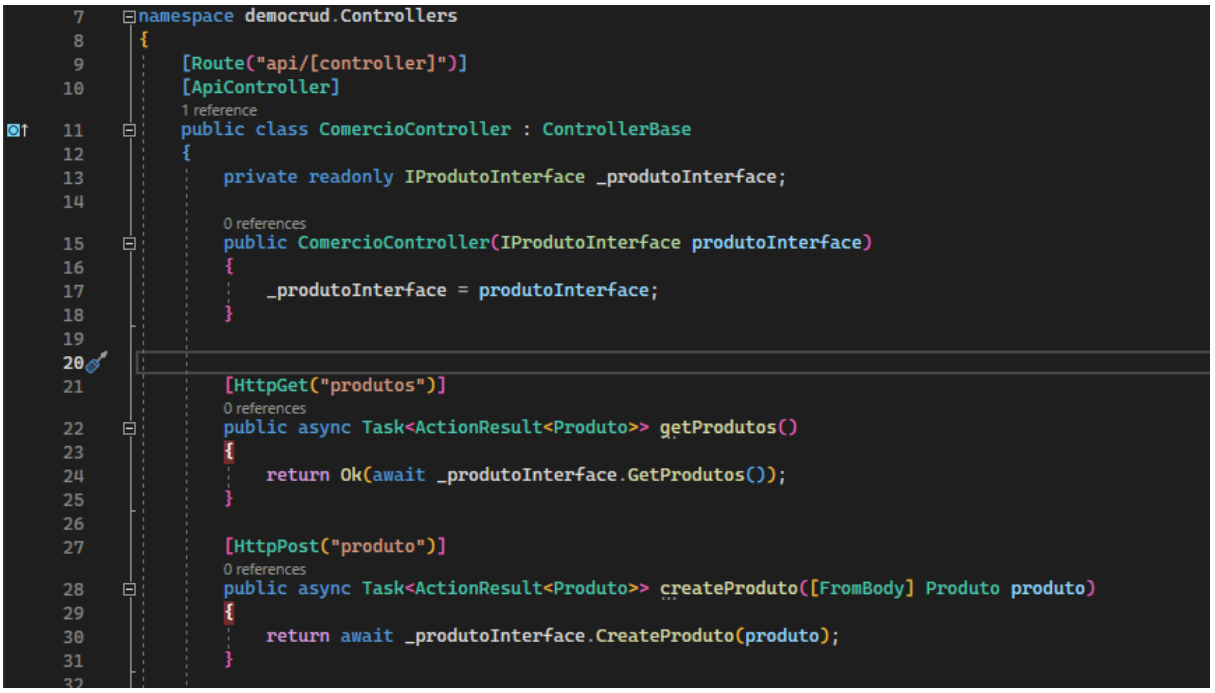
```
using democrud.models.ComercioModels;
using Microsoft.EntityFrameworkCore;

namespace democrud.DataContext
{
    3 references
    public class ComercioContext : DbContext
    {
        0 references
        public ComercioContext(DbContextOptions<ComercioContext> options) : base(options)
        {
        }

        //criar tabelas

        0 references
        public DbSet<Cliente> Cliente { get; set; }
        0 references
        public DbSet<Produto> Produto { get; set; }
        0 references
        public DbSet<Venda> Venda { get; set; }
    }
}
```

Nota do chatgepeto:

> A sobrescrita do método `OnModelCreating` é comumente usada na abordagem code-first para definir o mapeamento entre as entidades do aplicativo e as tabelas do banco de dados. No entanto, na abordagem database-first, o mapeamento já está definido no próprio banco de dados.

> O método `OnConfiguring` geralmente é usado para configurar o provedor de banco de dados e a string de conexão. No entanto, na configuração do Startup, você já está configurando o contexto do banco de dados e a string de conexão usando a injeção de dependência, então não há necessidade de sobrescrever esse método na classe de contexto.

5) Agora é possível criar crud através do context que foi criado.



```
namespace democrud.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class ComercioController : ControllerBase
    {
        private readonly IProdutoInterface _produtoInterface;

        0 references
        public ComercioController(IProdutoInterface produtoInterface)
        {
            _produtoInterface = produtoInterface;
        }


        [HttpGet("produtos")]
        0 references
        public async Task<ActionResult<Produto>> getProdutos()
        {
            return Ok(await _produtoInterface.GetProdutos());
        }

        [HttpPost("produto")]
        0 references
        public async Task<ActionResult<Produto>> createProduto([FromBody] Produto produto)
        {
            return await _produtoInterface.CreateProduto(produto);
        }
```

Necessário fazer a configuração da injeção de dependência dentro do program.cs:

```
builder.Services.AddScoped<IProdutoInterface, ProdutoService>();
```