

Laboratório de Desenvolvimento de Software

Célio Carvalho
cdf@estg.ipp.pt

P. PORTO



CONTEXTUALIZAÇÃO (1/7)

- O .NET Core é uma *framework* de desenvolvimento de software de código aberto desenvolvido pela Microsoft. Permite o desenvolvimento e execução de aplicações em vários Sistemas Operativos (SO) Windows, Linux, macOS (*framework* multiplataforma).
- A primeira versão do .NET Core foi lançada em 2016, e trata-se de uma evolução do .Net Framework que foi cuja primeira versão foi lançada em 2002. A Microsoft lançou o .NET Core porque percebeu que deveria abrir o seu ecossistema para outros SO como o Linux e o macOS.
- O lançamento do .NET 5.0, unificou os projetos do .NET Core e do .NET Framework. A partir desse momento, o *roadmap* de ambos os projetos passou a ser apenas um. O .NET 5.0 foi lançado em novembro de 2020.
- O código do .NET Core é totalmente aberto, e é suportado pela .NET Foundation. As aplicações podem ser desenvolvidas em vários editores como o Visual Studio Code, Microsoft Visual Studio, etc. Possui também um *Command Line Interface (CLI)* que permite o desenvolvimento (e.g. *build, tests*) via linha de comandos.
- Podem desenvolver-se vários tipos de aplicações com esta tecnologia (e.g., *cloud, consola, web, mobile*).

continua no slide seguinte...



CONTEXTUALIZAÇÃO (2/7)

- Com o .NET Core é possível criar código fiável e de bom desempenho. Além disso permite desenvolver:
 - ✓ Código assíncrono;
 - ✓ Tipos genéricos;
 - ✓ LINQ (consulta integrada com a linguagem (e.g. C#));
 - ✓ Programação paralela;
 - ✓ *Delegates* e *lambdas*;
 - ✓ *Reflection*;
 - ✓ *Etc.*
- Existem dois componentes essenciais para a compilação de um projeto .NET Core: o código fonte, e o ficheiro de projeto que contém configurações, dependências, etc. As imagens abaixo, apresentam um exemplo de código C# e um exemplo de um csproj (ficheiro de projeto).

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
  </PropertyGroup>
</Project>
```

C#

```
Console.WriteLine("Hello, World!");
```

.NET CLI

```
dotnet run
Hello, World!
```

continua no slide seguinte...



CONTEXTUALIZAÇÃO (3/7)

- Existem duas distribuições do .NET Core:
 - ✓ **.NET Runtime** que contém os ficheiros essenciais para que uma aplicação desenvolvida em .NET possa executar.
 - ✓ **.NET SDK** que, além do *runtime* do .NET, contém também ferramentas e bibliotecas adicionais para criar e testar aplicações (e.g. o MSBuild).
- O *runtime* do .NET é definido como *managed code*, principalmente porque usa um *garbage collector* na gestão de memória.
- Existem várias linguagens de programação no .NET Core: C# (orientado a objetos), F# (orientado a dados) e Visual Basic (orientado a objetos) (linguagem mais próxima da linguagem humana).
- As aplicações .NET são compiladas em *Intermediate Language* (IL). Trata-se de uma linguagem suportada pelos vários SO e arquiteturas de processadores. Quando executada, este código é compilado para o código nativo do CPU onde a aplicação está a ser executada (e.g. Arm64, x64).

continua no slide seguinte...



CONTEXTUALIZAÇÃO (4/7)

- Alguns tipos definidos nas bibliotecas .NET Core:
 - ✓ Todos os tipos derivam da classe `System.Object`;
 - ✓ Contém tipos primitivos como o `System.Boolean` e `System.Int32`;
 - ✓ Existem vários tipos de coleções como listas genéricas e dicionários (`List<T>`, `Dict< TKey, TValue >`);
 - ✓ Existem tipos para lidar com dados como `System.Data.Dataset` e `System.Data.DataTable`;
 - ✓ Contém componentes relacionados com rede (e.g. `System.Net.Http.HttpClient`);
 - ✓ Contém mecanismos que facilitam a serialização (e.g. `System.Text.Json.JsonSerializer`);
 - ✓ Etc.
- A partilha de código compilado é simples, utilizando o **NuGet Package Manager**.
 - <https://www.nuget.org/>
- O *Command Line Interface* (CLI) facilita a integração com vários ambientes de CI/CD (e.g. GitLab, GitHub, Azure DevOps).

continua no slide seguinte...



CONTEXTUALIZAÇÃO (5/7)

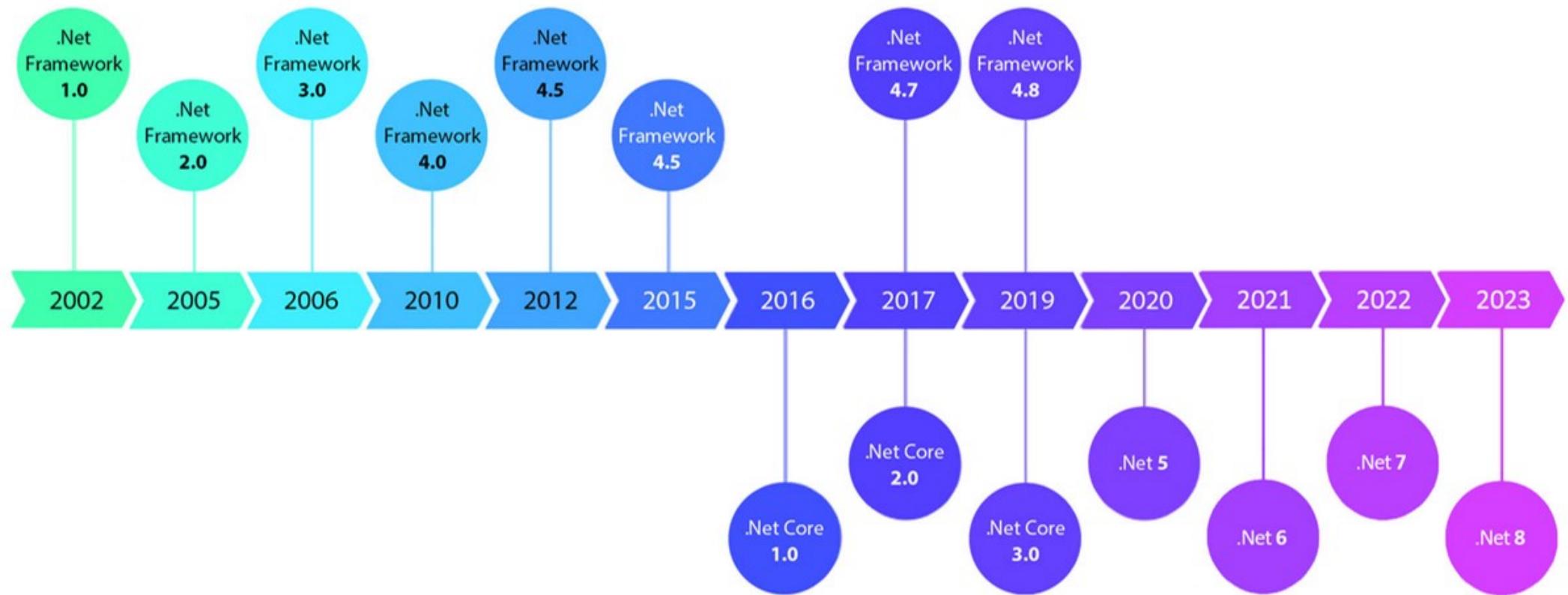
- As aplicações desenvolvidas em .NET podem ser publicadas de duas formas distintas:
 - ***Self-contained***, contém tudo o que é necessário para a aplicação executar no sistema *host* (permite executar uma aplicação .NET num ambiente onde o *runtime* .NET não está instalado).
 - ***Framework-dependent***, só contém o código necessário a executar, o que significa que o sistema *host* tem que ter instalado o .NET *runtime* instalado para que a aplicação funcione.
- As implementações .NET com maior relevância são:
 - **.NET Framework**, é a *framework* original e que ainda é bastante usada em aplicações para Windows desktop e *server*.
 - **Mono**, é a implementação do .NET Framework para multiplataforma (Windows, Android, iOS).
 - **.NET Core**, é a evolução do .NET Framework para multiplataforma (Windows, Linux, macOS), mais voltada para a *cloud*.

continua no slide seguinte...



CONTEXTUALIZAÇÃO (6/7)

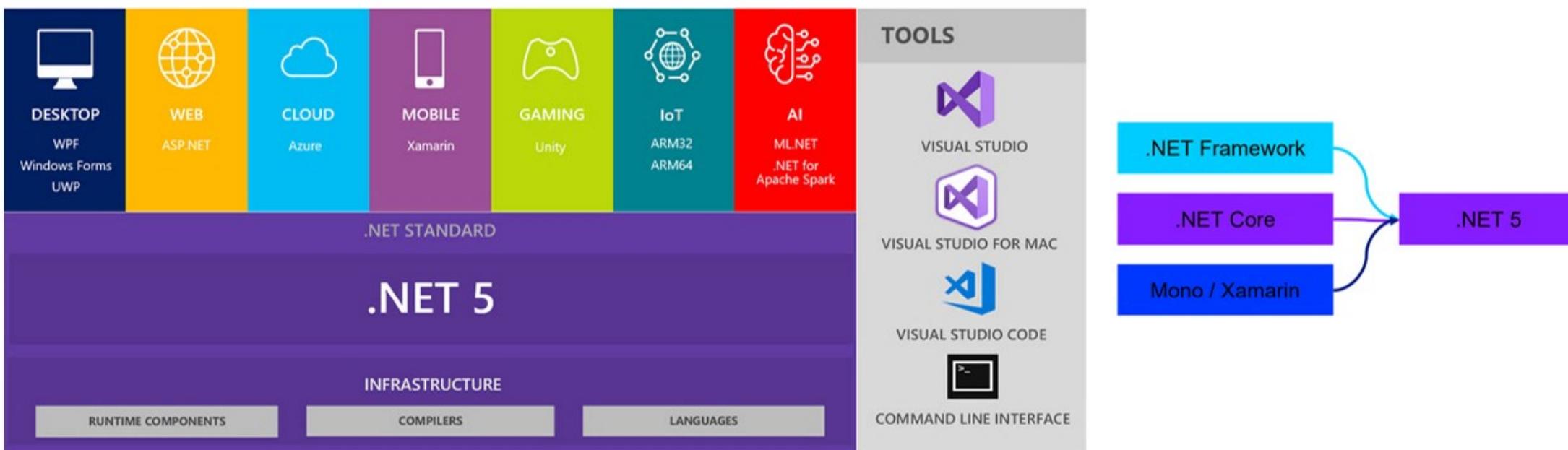
- A decisão de parar o desenvolvimento do .NET Framework e apostar no .NET Core (multiplataforma) foi anunciada pela Microsoft em 12nov de 2014.





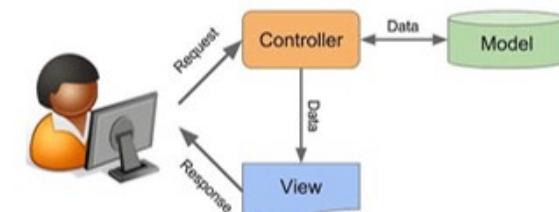
CONTEXTUALIZAÇÃO (7/7)

- Para o futuro, a Microsoft vê o .NET Core como uma plataforma unificada. A ideia é ter apenas uma framework que permita criar aplicações para vários SO: Windows, Linux, macOS, iOS, Android, tvOS, watchOS, WebAssembly, etc. A visão que existe é que seja possível criar com .NET aplicações para *desktop, web, cloud, mobile*, etc.



MODEL-VIEW-CONTROLLER (MVC)

- Padrão de desenvolvimento de software que divide a aplicação em 3 camadas ou componentes principais: **Model**, **View** e **Controller** (MVC).
- Os **Models** são componentes que representam os dados da aplicação. Uma aplicação pode ter vários **Models** e, normalmente, abstraem as entidades existentes no domínio do problema a resolver. Por exemplo, no domínio bancário, provavelmente existirão os **Models**: Conta, Titular, Balcão, Funcionário, etc.
- As **Views** são os componentes de interface com os utilizadores. São várias as tecnologias que permitem desenvolver o *front-end* de uma aplicação (e.g. ASP.Net Razor, React, Flutter).
- Os **Controllers** são os componentes que suportam a interação efetuada com o utilizador. Encapsulam o comportamento da aplicação (regras de negócio), processando as instruções recebidas das **Views** recorrendo aos **Models** para trabalhar os dados. Funcionam como um conjunto de classes do tipo serviço.
- EXEMPLO: A submissão da **View** Criar titular resultante do pressionar do botão Submeter, envia um **Model** com os dados de um Titular ao **Controller**, para que este valide o conteúdo das suas propriedades e persista o novo titular na base de dados.

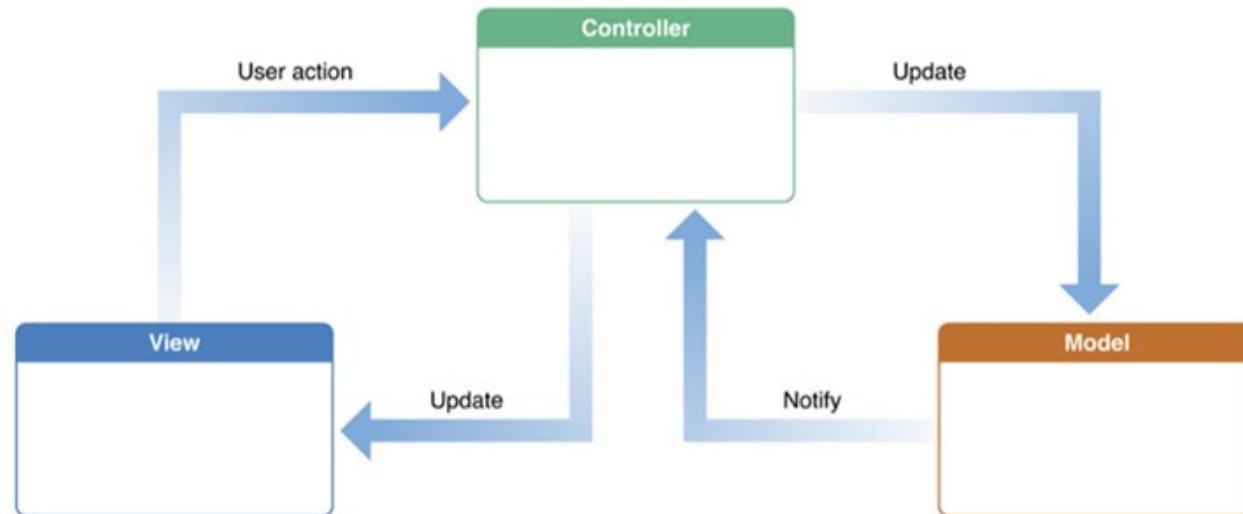
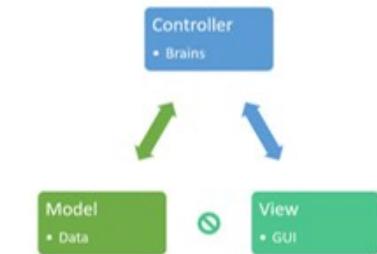
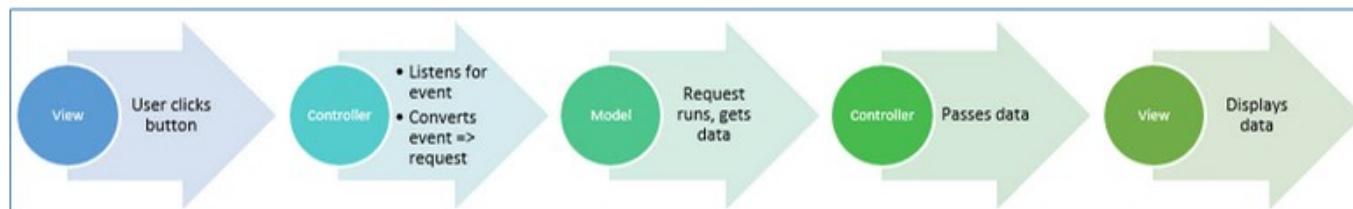


MODEL-VIEW-CONTROLLER (MVC) | Vantagens

- Padrão muito utilizado em aplicações web.
- *Loosely Coupled*, tornando os componentes mais independentes entre si.
- Promove a reutilização de código (e.g. classes Model).
- Mais fácil manutenção e modificação.
- Cada componente pode ser testado separadamente.
- Facilita o planeamento do desenvolvimento.

MODEL-VIEW-CONTROLLER (MVC) | Arquitetura (1/2)

- O MVC separa de forma clara a camada de apresentação da camada da lógica de negócio.



MODEL-VIEW-CONTROLLER (MVC) | Arquitetura (2/2)

The screenshot shows the Apple.com homepage from July 14, 1997. The top navigation bar includes links for "Find It", "Product Information", "Customer Support", "Technology & Research", "Developer World", "Groups & Interests", "Resources Online", and "About Apple". A sidebar on the left lists "Apple Sites Worldwide" with links for Switzerland and Taiwan. The main content area features the Apple logo and the text "Welcome to Apple". A central banner promotes "Introducing CyberDrive" with the tagline "Register today for a free CD-ROM." Below this, a section titled "What's Hot" highlights "Preorder Mac OS 8" and "Be the First to Know". To the right, there are sections for "iMATE 300" (Mobile, Affordable, & Smart) and "MOVIES FROM MARS" (QuickTime VR Takes You Out of this World).

JULY 14 1997

Welcome to Apple

Find It

Product Information

Customer Support

Technology & Research

Developer World

Groups & Interests

Resources Online

About Apple

Apple Sites Worldwide

Switzerland

Taiwan

Introducing CyberDrive

Register today for a free CD-ROM.

What's Hot

Preorder Mac OS 8

Now you can [preorder Mac OS 8](#), described by Macworld

Be the First to Know

Learn about new Macintosh software releases the moment

iMATE 300

Mobile, Affordable, & Smart

MOVIES FROM MARS

QuickTime VR Takes You Out of this World

ENTITY FRAMEWORK | CONTEXTUALIZAÇÃO (1/3)

Entity Framework Core

- Da integração de uma aplicação com uma base de dados, resulta muitas vezes na repetição de código. Normalmente existem várias entidades no modelo de classes e, para cada uma dessas entidades, implementam-se as operações Create, Read, Update & Delete (CRUD).
- A implementação repetitiva deste código demora tempo e é sujeita a erros de implementação. Além disso, a manutenção deste código também pode ser morosa, e sujeita a erros de implementação.
- O **Entity Framework (EF)** é uma *framework Object/Relational Mapping (O/RM)* que foi criada para facilitar o processo de implementação do CRUD sobre os dados de bases de dados. Basicamente, disponibiliza automaticamente implementações de ligação a dados para executar as operações do CRUD.
- O EF é uma evolução do ADO.Net. O ADO.Net é uma tecnologia que permite implementar lógica de ligação a dados com vários Sistemas de Gestão de Bases de Dados (SGBD) (e.g. SQL Server).
- As *frameworks O/RM* permitem editar e consultar dados, ligando de forma simples e com pouco desenvolvimento, os objetos da solução (*Models*) às bases de dados onde os dados são guardados.

continua no slide seguinte...

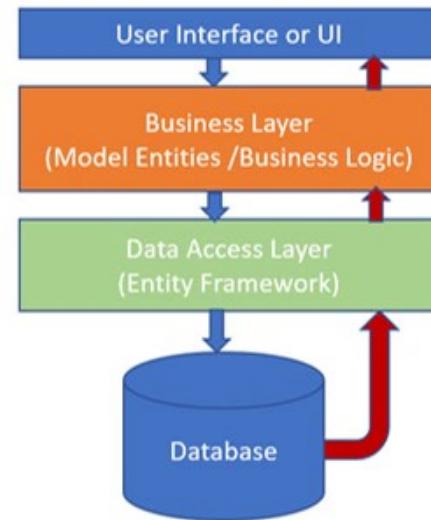
ENTITY FRAMEWORK | CONTEXTUALIZAÇÃO (2/3)

- Durante o desenvolvimento de uma aplicação, especificam-se classes de dados (*Models*) que pertencem ao domínio do problema a resolver.
- A criação dos *Models* é importante para o EF, porque é através dos *Models* que os dados da base de dados são representados dentro da aplicação (e.g. quando se carrega um registo da tabela Products da base de dados, estes dados ficam guardados em memória num objeto do tipo Product).
- Neste contexto, existem 3 estratégias que podem ser seguidas na geração dos *Models*:
 - **Code-first**, especificam-se as classes em código e, posteriormente, gera-se o *schema* correspondente na base de dados;
 - **Model-first**, as classes são modeladas visualmente e, posteriormente é gerado o *schema*.
 - **Database-first**, Primeiro cria-se o *schema* da base de dados e, tendo por base esse *schema*, pede-se ao EF que gere os *Models* no projeto.
- Nesta Unidade Curricular (UC) será utilizado a estratégia *code-first*.

continua no slide seguinte...

ENTITY FRAMEWORK | CONTEXTUALIZAÇÃO (3/3)

- O EF facilita a integração das aplicações .NET com bases de dados (e.g. SQL Server, PostgreSQL, sqlite, Azure CosmosDB).
- A utilização de uma framework O/RM, permite que o programador não tenha que lidar diretamente com a base de dados. Permite que não tenha que escrever instruções SQL, a não ser que prefira ou, em casos mais específicos, o tenha que fazer.
- Quando é executada uma instrução sobre os dados (e.g. inserção de um objeto), o EF traduz automaticamente essa operação em instruções SQL considerando o *schema* e o SBGD subjacente.
- O EF facilita o trabalho dos programadores, promove a velocidade de programação, e evita erros no desenvolvimento de código de integração com bases de dados, já que não tem que ser desenvolvido.
- Com o EF é possível controlar transações, e lidar com o relacionamento entre entidades (e.g. lazy loading).



ENTITY FRAMEWORK | NuGet *packages*

Entity Framework 

- O quadro abaixo resume os Nuget *packages* normalmente necessários para um projeto EF.

Nome	Descrição
Microsoft.EntityFrameworkCore.SqlServer	Extensão do package Microsoft.EntityFrameworkCore, específico para Microsoft SQL Server.
Microsoft.EntityFrameworkCore.Design	Auxilia o programador no desenvolvimento na criação e migração de bases de dados em EF.
Microsoft.EntityFrameworkCore.Tools	Módulo auxiliar ao anterior, para apoiar na migração de bases de dados. Também disponibiliza comandos a serem executados na consola.

ENTITY FRAMEWORK | Demonstração OnStore1 (1/14)

Entity Framework Core

- Nesta demonstração vai-se criar uma pequena aplicação para mostrar, de forma simples, o funcionamento do EF. Nesta aplicação vão ser implementadas as operações CRUD sobre a tabela Products de uma base de dados (BD) chamada OnStore. Nesta demonstração o EF será integrado com SQL Server.
- Crie uma aplicação .NET 7.0 utilizando o *template* ConsoleApp.

Configure your new project

Console App C# Linux macOS Windows Console

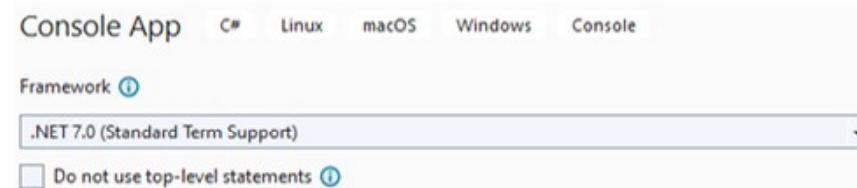
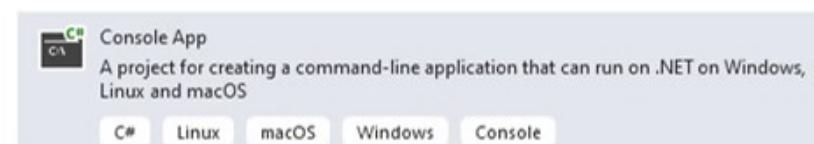
Project name
OnStore1

Location
D:\[REDACTED]\dotnet_ef_onstore1\

Solution name ⓘ
OnStore1

Place solution and project in the same directory

Project will be created in "D:\[REDACTED]\dotnet_ef_onstore1\OnStore1\OnStore1\"

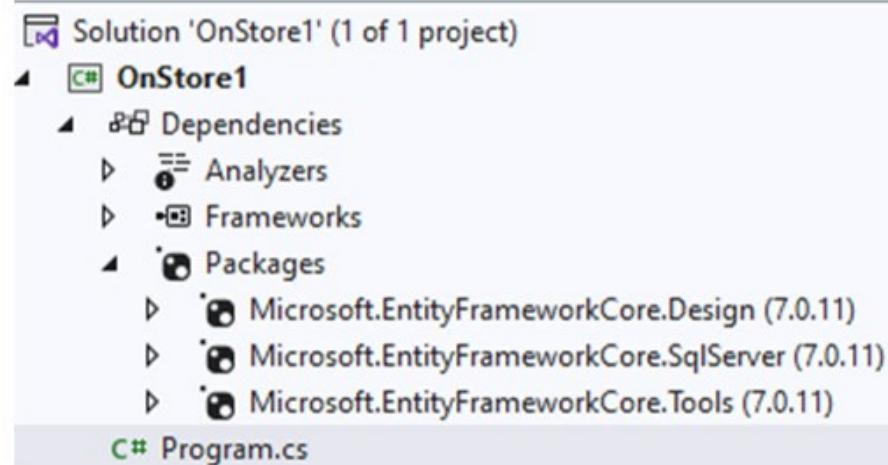


continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (2/14)

Entity Framework Core

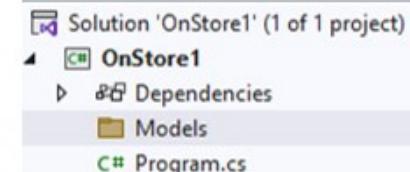
- Inicie-se o desenvolvimento desta demonstração, carregando os NuGet *packages* necessários para se trabalhar com EF. Utilize a opção seguinte para aceder aos NuGet *packages*.
 - Tools | NuGet Package Manager | Manage Nuget Packages for Solution
- Instale na solução atual os seguintes packages NuGet:
 - Microsoft.EntityFrameworkCore.SqlServer
 - Microsoft.EntityFrameworkCore.Design
 - Microsoft.EntityFrameworkCore.Tools
- Agora que já estão disponíveis na solução os *packages* necessários inicie-se o desenvolvimento da solução.
- Nesta e nas restantes demonstrações deste documento, não será utilizada comentação. Esta boa prática não será seguida, apenas para ser mais fácil apresentar o conteúdo (*código*) no formato de *slides*.
- Note que a comentação é importante para o produto final de *software* desenvolvido. Deve implementar a comentação seguindo as convenções, dada a sua importância para a leitura e organização do código.



continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (3/14)

- Um dos primeiros passos no desenvolvimento de uma aplicação é a criação dos *Models*. Crie uma pasta chamada *Models* dentro da solução (por convenção esta pasta chama-se sempre **Models**).
- Crie a classe *Product* dentro da pasta *Models* com o conteúdo apresentado na imagem.
- A propriedade *Id* é uma propriedade especial que representa o identificador único de objeto. Este atributo é mapeado automaticamente pelo EF para uma *primary key* (PK) na BD. Portanto, neste caso, é desnecessário indicar o atributo *Key*.
- A propriedade *Description* está ser inicializada como *null!*. Esta é uma forma de o programador indicar ao .NET Core que tem consciência que a propriedade é *null* por defeito.
- O último atributo está decorado com o atributo *Column* para indicar o tipo de dados que se pretende para este campo na BD.



```
internal class Product
{
    [Key] // unnecessary
    0 references
    public int Id { get; set; }

    0 references
    public string Description { get; set; } = null!;

    [Column(TypeName = "decimal(6, 2)")]
    0 references
    public decimal Price { get; set; }

    0 references
    public Category Category { get; set; }

    0 references
    public int CategoryId { get; set; }
}
```

continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (4/14)

Entity Framework Core

- Existem várias *Annotations* que podem ser definidas em EF:
 - ✓ **Required**, que indica que a Propriedade mapeia para um atributo não-nulo na BD;
 - ✓ **MinLength**, que indica o tamanho mínimo de uma *string*;
 - ✓ **MaxLength**, que indica o tamanho máximo de uma *string*;
 - ✓ **ForeignKey**, que indica tratar-se de uma chave estrangeira para outra tabela;
 - ✓ **Column**, indica o nome de uma coluna na BD;
 - ✓ **DatabaseGenerated**, indica um valor calculado na BD;
 - ✓ **Index**, indica ao EF que deve ser gerado um índice na BD do(s) atributo(s) mencionado(s);
 - ✓ Etc.
- Poderá consultar na internet mais *annotations* de EF:
 - <https://learn.microsoft.com/en-us/ef/ef6/modeling/code-first/data-annotations>
- No decurso das demonstrações deste documento serão introduzidas / indicadas algumas *Annotations* das possíveis. Para informação acerca do seu significado, utilize o link indicado acima.

continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (5/14)

Entity Framework Core

- Seguindo com a demonstração, adicione-se agora a classe Category que representará a categoria dos produtos. Crie a classe abaixo com o conteúdo apresentado.

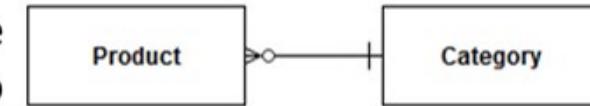
- O tipo `string?` indica ao EF que o atributo da BD deve ser *nullable*.

- A propriedade `Products` é uma propriedade de navegação. Através desta propriedade é criada uma relação de `1..*` na BD.

- Edite a classe `Product` e, agora que existe a classe `Category`, adicione as duas propriedades seguintes (conforme a imagem):

- Category**, é também uma propriedade de navegação, e representa a categoria de um produto; e
- CategoryId**, será criada pelo EF automaticamente se não especificada, é uma *shadow property*. Representa a chave estrangeira da categoria.

continua no slide seguinte...



```
internal class Category
{
    0 references public int Id { get; set; }

    0 references public string? Name { get; set; }

    0 references public ICollection<Product> Products { get; set; }
}
```

```
internal class Product
{
    0 references public int Id { get; set; }

    0 references public string Description { get; set; } = null!;

    [Column(TypeName = "decimal(6, 2)")]
    0 references public decimal Price { get; set; }

    0 references public Category Category { get; set; }

    0 references public int CategoryId { get; set; }
}
```

ENTITY FRAMEWORK | Demonstração OnStore1 (6/14)

- De seguida vai criar-se uma classe serviço *database context* que vai permitir trabalhar com a base de dados. Por convenção as classes contexto de BD são especificadas na pasta Data.
- Crie a classe `OnStoreContext` com o conteúdo indicado abaixo. Esta classe herda de `DbContext` e funciona como uma sessão de trabalho da BD.
- Nesta classe especificam-se propriedades do tipo `DbSet`. Cada um dos `DbSet` mapeia uma *entity* (tabela) da BD com uma classe de dados (*Model*) da aplicação que está a ser desenvolvida.
- O evento `OnConfiguring()` permite que sejam adicionadas opções ao `DbContext` aquando da sua inicialização. Neste caso, vai definir-se a *ConnectionString* (CS) à base de dados. Note que não é boa prática definir-se a CS em *hardcode*. Numa solução real, guarde as CS num outro local (e.g. `appsettings.json`).
- Para mais informações acerca de *ConnectionStrings*:
 - <https://learn.microsoft.com/en-us/ef/core/miscellaneous/connection-strings>

```
internal class OnStoreContext : DbContext
{
    public DbSet<Product> Products { get; set; } = null;
    public DbSet<Category> Categories { get; set; } = null;
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            @"Data Source=.;Initial Catalog=OnStore1;Integrated Security=True;TrustServerCertificate=True");
    }
}
```

continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (7/14)

- Chegou o momento de criar a base de dados tendo por base as classes de dados (*Models*) definidas na solução. Para isso, vai utilizar-se o *Package Manager Console* (PM) para criar uma *migration*.
- Também se pode utilizar o *Command Line Interface* (CLI) do .NET (e.g. para utilizadores que não usam o Visual Studio). Para utilizar este CLI deve, antes de mais, instalar o `dotnet-ef` como uma ferramenta global.
 - `dotnet tool install -g dotnet-ef`
- Aceda à opção seguinte para criar uma *migration* na solução:
 - Tools | NuGet Package Manager | Package Manager Console
- Para criar uma nova *migration*, insira o comando abaixo no PM.
 - `Add-Migration InitialDBSchema`
- Se pretender criar uma nova *migration* através do CLI utilize o comando abaixo.
 - `dotnet ef migrations add InitialDBSchema`
- Nesta demonstração será utilizado o PM. No entanto, poderá utilizar o CLI para efetuar as mesmas operações. Pesquise os comandos equivalentes na documentação do EF.

```
PM> Add-Migration InitialDBSchema
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
```

continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (8/14)

Entity Framework Core

- Abra a classe gerada automaticamente dentro da pasta *Migrations*, e analise o seu conteúdo para validar se está conforme o pretendido. A classe criada tem o sufixo do nome da *migration* atribuído (neste caso *InitialDBSchema*).

- Depois de validar o conteúdo da classe, pode executar-se a *migration*, executando o comando abaixo.

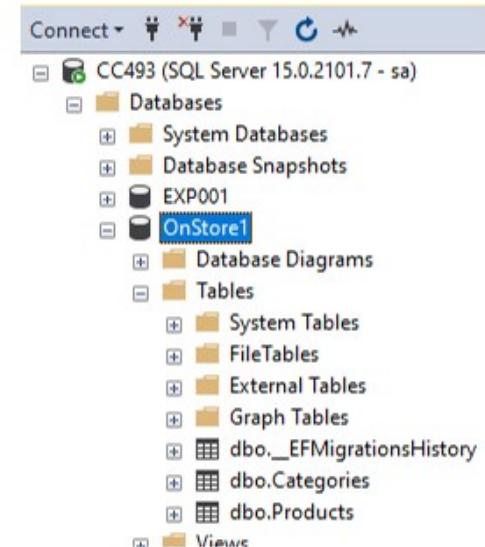
➤ **Update-Database**

➤ **dotnet ef database update** (comando CLI correspondente)

```
PM> Update-Database
Build started...
Build succeeded.
Applying migration '20231008145852_InitialDBSchema'.
Done.
```

- Consulte a estrutura da BD criada (e.g. com o SQL Server Management Studio (SSMS)).

- ✓ Como esperado, as tabelas para guardar os dados dos *Models* foram criadas;
- ✓ Existe uma tabela chamada `__EFMigrationsHistory`. Esta tabela foi criada e é gerida integralmente pelo EF. É com base nos dados guardados nesta tabela, que o EF sabe quais as *migrations* que deve aplicar para fazer o *update* ao *schema* da base de dados.



continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (9/14)

- Durante o desenvolvimento das aplicações é normal que o *schema* da base de dados vá alterando. De seguida, vai simular-se o acréscimo da propriedade `ReleaseYear` na classe `Product`. Altere o código da classe como apresentado na imagem.
- Repare na *Annotation Range* que indica que a propriedade apenas pode conter um valor inteiro entre 2020 e 2030.
- Após a edição da classe, tem que se repercutir esta alteração também no *schema* da BD. A forma de o fazer é adicionando uma nova *migration*.
 - `Add-Migration ReleaseYearAdded`
 - `Update-Database`
- Após executar os comandos acima, consulte a tabela `Products` na BD para verificar a alteração do *schema*.

```
internal class Product
{
    [Column(TypeName = "decimal(6, 2)")]
    public decimal Price { get; set; }

    [Range(2020, 2030)]
    public int ReleaseYear { get; set; }

    public Category Category { get; set; }

    public int CategoryId { get; set; }
}

CC493.OnStore1 - dbo.Products
+ X
| Column Name | Data Type | Allow Nulls |
| Id          | int        |  |
| Description | nvarchar(MAX) |  |
| Price       | decimal(6, 2) |  |
| CategoryId | int        |  |
| ReleaseYear | int        |  |
```

continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (10/14)

- De seguida vai-se simular algumas operações CRUD com a BD. Note que, tratando-se apenas de uma demonstração do EF, não se seguirá nenhum padrão de desenvolvimento ou organização de código. Pretende-se apenas mostrar o funcionamento do EF, mantendo a solução em demonstração o mais simples possível.
- Implemente o código da imagem no ficheiro `Program.cs`. A instrução `using` indica ao .NET para libertar os recursos utilizados pela variável `context`, logo que esta deixe de ser necessária.
- Os métodos `.Add()` apenas adicionam objetos a uma lista em memória. Só quando o método `SaveChanges()` é executado, é que os dados são, realmente, persistidos na BD. Note que o segundo `.Add()` está a ser executado diretamente do `context`. O .NET infere a *Collection* tendo por base o tipo que está a ser adicionado.
- Depois de executar o programa, consulte a BD para garantir que os dados foram guardados na tabela `Categories`. Para executar, utilize o botão *Debug* do Visual Studio ou o comando `dotnet run OnStore1` na linha de comandos (pasta onde está o ficheiro `csproj`).

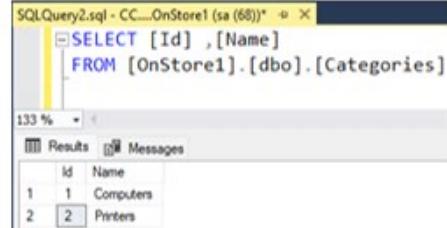
```
using OnStore1.Data;
using OnStore1.Models;

using OnStoreContext context = new OnStoreContext();

// Categories table: insertion test 1
Category category1 = new Category();
category1.Name = "Computers";
context.Categories.Add(category1);

// Categories table: insertion test 2
Category category2 = new Category { Name = "Printers" };
context.Add(category2);

// save data in DB
context.SaveChanges();
```



The screenshot shows a SQL query window titled 'SQLQuery1.sql - CC....OnStore1 (sa (68))'. The query is:

```
SELECT [Id] , [Name]
FROM [OnStore1].[dbo].[Categories]
```

The results pane shows the following data:

	Id	Name
1	1	Computers
2	2	Printers

continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (11/14)

Entity Framework 

- Insira alguns produtos na BD executando o código da imagem. Após executar, valide que os produtos tenham sido criados na BD.
- Para ler dados da base de dados, pode usar a sintaxe *Lambda* ou *LINQ*. Abaixo apresentam-se ambos os exemplos de código. Implemente-os e analise o resultado na consola.

```
using OnStoreContext context = new OnStoreContext();

var products = context.Products
    .Where(p => p.ReleaseYear >= 2022)
    .OrderBy(p => p.Description);

foreach (var p in products)
{
    Console.WriteLine($"Id: {p.Id} - {p.Description} ({p.ReleaseYear}); Price: {p.Price}; ");
}
```

```
context.Products.AddRange(
    new Product { Description = "IBM Computer M823", CategoryId = 1, ReleaseYear = 2021, Price = 982.87m },
    new Product { Description = "HP Computer HY711", CategoryId = 1, ReleaseYear = 2022, Price = 892.43m },
    new Product { Description = "HP Computer HY832", CategoryId = 1, ReleaseYear = 2022, Price = 593.07m },
    new Product { Description = "HP Color Printer 3981", CategoryId = 2, ReleaseYear = 2021, Price = 128.32m },
    new Product { Description = "HP Balck Printer 1192", CategoryId = 2, ReleaseYear = 2022, Price = 329.85m });

context.SaveChanges();
```

continua no slide seguinte...

```
using OnStoreContext context = new OnStoreContext();

var products = from p in context.Products
    where p.ReleaseYear >= 2022
    orderby p.Description
    select p;

foreach (var p in products)
    Console.WriteLine($"Id: {p.Id} - {p.Description} ({p.ReleaseYear}); Price: {p.Price}; ");
```

ENTITY FRAMEWORK | Demonstração OnStore1 (12/14)

- Para alterar um Product, primeiro tem que se criar uma referencia ao registo. No bloco de código apresentado, é carregado o produto com o id=2 com o objetivo de atualizar o respetivo ReleaseYear e preço. O método `FirstOrDefault()` devolve o Product se existir ou null se não existir. Por esse motivo, antes de se utilizar o conteúdo da variável, tem que se verificar se o conteúdo contém um Product.
- Depois de executar o código abaixo, certifique-se que as alterações foram repercutidas na base de dados, ou execute a aplicação para ver o resultado da consola.

```
using OnStoreContext context = new OnStoreContext();

var product = context.Products
    .Where(p => p.Id == 2)
    .FirstOrDefault();

if (product is Product)
{
    product.ReleaseYear = 2023;
    product.Price = 999.99m;
    context.SaveChanges();
}

context.Products
    .Where(p => p.ReleaseYear == 2023)
    .ToList()
    .ForEach(p => Console.WriteLine($"Id: {p.Id} - {p.Description} ({p.ReleaseYear}); Price: {p.Price}; "));
```

Id: 2 - HP Computer HY711 (2023); Price: 999,99;

	Id	Description	Price	CategoryId	ReleaseYear
1	1	IBM Computer M823	982.87	1	2021
2	2	HP Computer HY711	999.99	1	2023
3	3	HP Computer HY832	593.07	1	2022
4	4	HP Color Printer 3981	128.32	2	2021
5	5	HP Balck Printer 1192	329.85	2	2022

continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (13/14)

- Remover um ficheiro na tabela é uma operação similar à anterior. Cria-se uma referencia do registo através do objeto e consome-se o método Remove().
- Depois de executar o código abaixo, certifique-se que a eliminação aconteceu na base de dados.

```
var product = context.Products
    .Where(p => p.Id == 3)
    .FirstOrDefault();

if (product is Product)
{
    context.Remove(product);
    context.SaveChanges();
}

foreach (Product p in context.Products)
    Console.WriteLine($"Id: {p.Id} - {p.Description} ({p.ReleaseYear}); Price: {p.Price}; ");

Id: 1 - IBM Computer M823 (2021); Price: 982,87;
Id: 2 - HP Computer HY711 (2023); Price: 999,99;
Id: 4 - HP Color Printer 3981 (2021); Price: 128,32;
Id: 5 - HP Balck Printer 1192 (2022); Price: 329,85;
```

	Description	Price	CategoryId	ReleaseYear
1	IBM Computer M823	982.87	1	2021
2	HP Computer HY711	999.99	1	2023
4	HP Color Printer 3981	128.32	2	2021
5	HP Balck Printer 1192	329.85	2	2022

continua no slide seguinte...

ENTITY FRAMEWORK | Demonstração OnStore1 (14/14)

Entity Framework Core

- Em EF é também possível enviar *queries* SQL diretamente para o motor de dados. A imagem abaixo apresenta um exemplo simples. O argumento do método `FromSqlInterpolated()` pode ser a execução de uma Stored Procedure (e.g. “`EXECUTE uSP_GetProducts @filterByReleaseYear=2022`”)

```
int releaseYear = 2022;
var products = context.Products
    .FromSqlInterpolated($"SELECT * FROM Products Where ReleaseYear >= {releaseYear}");

foreach (var p in products)
    Console.WriteLine($"Id: {p.Id} - {p.Description} ({p.ReleaseYear}); Price: {p.Price}; ");
```

- Para mais exemplos de utilização de instruções SQL, consultar link abaixo.
➤ <https://learn.microsoft.com/en-us/ef/core/querying/sql-queries>

continua no slide seguinte...

ENTITY FRAMEWORK | *Eager, Explicit, e Lazy Loading*

Entity Framework Core

- As propriedades de navegação, permitem aceder a dados relacionados. No exemplo anterior, o Product tem um Category que é uma entidade relacionada. Da mesma forma, Category tem uma lista de Product, sendo assim uma lista de entidades relacionadas.
- A questão que se coloca é, quando e como carregar estas propriedades de navegação relacionadas. Carregá-las sempre pode ser utilização de tempo e recursos desnecessariamente. Assim, existem 3 tipos de carregamento de dados relacionados.
 - ✓ **Eager loading**, os dados das entidades relacionadas são carregados aquando do carregamento dos dados da entidade principal;
 - ✓ **Explicit loading**, os dados relacionados são carregados de forma explícita posteriormente; e
 - ✓ **Lazy loading**, os dados relacionados são carregados de forma implícita (automática) quando a propriedade é acedida, i.e., quando se carregam os dados principais os dados relacionados não são carregados (são carregados apenas quando são necessários, e automaticamente).

ENTITY FRAMEWORK | *Eager Loading*

- Tendo ainda por base o exemplo da demonstração anterior, implemente o código apresentado na imagem. Repare na utilização do método `Include()` como forma de indicar que se pretende carregar aquela propriedade em específico.
- Se implementar a mesma solução sem especificar o `Include()`, a instrução `p.Category.Name` vai gerar um erro a indicar que `Category` é `null`, i.e., os dados desta entidade auxiliar não foram carregados da BD.
- Pode executar o teste também no carregamento da lista de produtos, a partir da `Category` (imagem abaixo).

```
var categories = context.Categories
    .Include(c => c.Products);

foreach (var category in categories)
{
    Console.WriteLine(category.Name);
    foreach (var product in category.Products)
        Console.WriteLine($" \t[{product.Id}] {product.Description}");
}
```

```
var products = context.Products
    .Where(p => p.ReleaseYear > 2000)
    .Include(p => p.Category)
    .ToList();

foreach (var p in products)
    Console.WriteLine($"[{p.Id}] {p.Description} - Category: { p.Category.Name }");
```

```
Computers
[1] IBM Computer M823
[2] HP Computer HY711
Printers
[4] HP Color Printer 3981
[5] HP Balck Printer 1192
```

ENTITY FRAMEWORK | *Explicit Loading*

Entity Framework Core

- O *Explicit Loading* é uma forma de carregar os dados, depois do carregamento dos dados da entidade principal. Diz-se carregamento explícito, porque a pedido de carregamento tem que ser dado explicitamente.
- O exemplo abaixo carrega num primeiro momento todos os Product. Posteriormente, quando os processa, decide quais deles pretende carregar os dados da Category.

```
var products = context.Products.ToList();

foreach (var p in products)
{
    if (p.ReleaseYear == 2022)
        context.Entry(p)
            .Reference(c => c.Category)
            .Load();

    Console.WriteLine($"[{p.Id}] {p.Description} - " +
        $"Category: { (p.Category is Category) ? p.Category.Name : "N/A" }");
}
```

```
[1] IBM Computer M823 - Category: N/A
[2] HP Computer HY711 - Category: N/A
[4] HP Color Printer 3981 - Category: N/A
[5] HP Balck Printer 1192 - Category: Printers
```

- O exemplo abaixo carrega explicitamente a lista de Product quando a Category == 2.

```
Computers
Printers
[4] HP Color Printer 3981
[5] HP Balck Printer 1192
```

```
var categories = context.Categories.ToList();

foreach (var category in categories)
{
    Console.WriteLine(category.Name);

    if (category.Id == 2)
    {
        context.Entry(category)
            .Collection(p => p.Products)
            .Load();
    }
    else
    {
        category.Products = new List<Product>();
    }

    foreach (var product in category.Products)
        Console.WriteLine($"{product.Id} {product.Description}");
}
```

ENTITY FRAMEWORK | *Lazy Loading*

- O *Lazy Loading* carrega os dados automaticamente, quando a propriedade é acedida. Uma das formas de se ativar o *Lazy Loading* é utilizando o NuGet `Microsoft.EntityFrameworkCore.Proxies`. Atenção que esta forma de carregamento só fica ativada, depois de se ativar no método `OnConfiguring()`.
- Defina as classes como `public`, e as propriedades de navegação como `virtual`, para que o EF possa fazer-lhes *override* quando precisar (ver imagens).
- Instale o NuGet `Microsoft.EntityFrameworkCore.Proxies` e altere o método `OnConfiguring()` conforme apresentado na imagem abaixo.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(@"Data Source=.;Initial Catalog=OnStore1;Integrated Security=True;TrustServerCertificate=True");
}
```

```
public class Category
{
    0 references
    public int Id { get; set; }

    1 reference
    public string? Name { get; set; }

    0 references
    public virtual ICollection<Product> Products { get; set; }
}

public class Product
{
    2 references
    public int Id { get; set; }

    2 references
    public string Description { get; set; } = null!;

    [Column(TypeName = "decimal(6, 2)")]
    0 references
    public decimal Price { get; set; }

    [Range(2020, 2030)]
    0 references
    public int ReleaseYear { get; set; }

    1 reference
    public virtual Category Category { get; set; }

    0 references
    public int CategoryId { get; set; }
}
```

ENTITY FRAMEWORK | Exercícios

Entity Framework Core

- Tendo por base a demonstração anterior, implemente as mesmas funcionalidades utilizando agora um BD Sqlite.

➤ Instale o Nuget package:

`Microsoft.EntityFrameworkCore.Sqlite`

➤ Indique no ficheiro csproj a pasta de trabalho onde deverá ser gravado e lido o ficheiro da base de dados (ver imagem).

➤ Altere o seu código do evento para consumir o método do Sqlite (ver imagem).

- Implemente agora a mesma solução utilizando uma BD PostgreSQL.

➤ O Nuget package a instalar é o `Npgsql.EntityFrameworkCore.PostgreSQL`

➤ Altere o código do evento para consumir o método do PostgreSQL (ver imagem).

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <StartWorkingDirectory>$({MSBuildProjectDirectory})</StartWorkingDirectory>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
  {
    optionsBuilder.UseSqlite("Data source=OnStore1.db");
  }
}
```

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
  optionsBuilder.UseNpgsql("User Id=postgres;Password=myPassword;Host=localhost;Port=5432;Database=OnStore1");
}
```

P. PORTO