Guia definitivo dos Nullable Types no .NET

Lidar com possíveis valores nulos em qualquer linguagem de programação é sempre um desafio e o .NET/C# tem uma ótima forma de te ajudar com isto.

Índice

O problema

Se você atualizou seu projeto para .NET 6 ou superior ou mesmo se começou um projeto recentemente, é possível que tenha visto diversos **warnings** (Avisos) sobre possíveis referências nulas em seu código.

```
warning CS8618: Non-nullable property 'XPTO' must contain a non-null value when exiting constructor
```

Embora este aviso não previna que sua aplicação seja compilada, ele indica que pode haver um possível valor nulo em seu código, que causa o famoso **NullReferenceException**.

O que mudou no .NET 6?

Estes avisos só são apresentados em versões 6 ou superiores do .NET por conta de uma nova configuração adicionada aos arquivos `.csproj`, chamada `<Nullable>`, conforme mostrada abaixo.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable> <!-- 👈-->
  </PropertyGroup>
</Project>
```

Embora seja plenamente possível, remover esta configuração do projeto ou mesmo atribuir o valor **Disable** para ela, a recomendação é que ela seja mantida.

O que é NULL?

Antes de falar sobre como resolver este problema no código, precisamos entender o que é o **NULL** e quais reais problemas temos com eles.

O .NET possui duas categorias de tipos, os tipos de referência (Reference Types) e os tipos de valor (Value Types). É importante frisar que todo tipo de valor é inicializado com um valor padrão diferente de **NULL**.

Na verdade, qualquer tipo pode ser nulo e vamos entender isso mais adiante, mas todo tipo de valor no .NET é inicializado com o que chamamos de valor padrão.

No caso dos números (int, double, float, decimal), este valor é zero, no caso de um booleano o valor é false, no caso de um char o valor é \0 e assim por diante.

Mesmo quando temos tipos complexos como estruturas (que são tipos de valor), nós temos um valor padrão sendo atribuído a elas.

Um bom exemplo é o DateTime, que se nenhum valor for atribuído, o mesmo recebe `1/1/0001 12:00:00 AM`.

Por que um tipo de valor não é nulo?

Se você já teve a oportunidade de fazer nosso curso GRATUITO de C#, aprendeu que os tipos no .NET tem um tamanho específico. Por exemplo, um `int` sempre vai ocupar 32 bits, um `short` sempre vai ocupar 16 bits e assim por diante.

> 💡 **APRENDA C# DE GRAÇA**
> O curso **Fundamentos do C#** conta com 144 aulas divididas em 14 módulos com mais de 12 horas de duração, **TOTALMENTE GRATUITO** e com certificado de conclusão.
>
> 👉 Fundamentos do C# | balta.io

Essa tipagem ajuda muito na alocação de memória, afinal, todo programa vai para lá antes de ser executado e saber que uma determinada estrutura precisa de X bytes na memória, já ajuda a organizar os dados.

Desta forma, não faz sentido algum inicializar um tipo de valor com nulo. Qual a vantagem em dizer ao computador que precisamos de 4 bytes da memória e não colocar nenhum valor lá?

Tipos de referência são os vilões

Então sabemos que os tipos de valor armazenam uma informação, mas e os tipos de referência, aliás o que são eles?

Diferente dos tipos de valor, os tipos de referência armazenam apenas um endereço, um referência para onde os dados possivelmente estarão.

Este comportamento ocorre pois com referências temos um dinamismo maior nos tipos, nos permitindo aumentar e diminuir seu tamanho.

Um ótimo exemplo de tipo de referência são as `strings` que nada mais são do que um **array** de `char`.

Nós podemos ter `strings` (Listas no geral) de qualquer tamanho, desta forma, faz todo sentido trabalharmos com tipo de referência para elas.

Então, na verdade, quando temos uma `string`, temos apenas uma referência para seu conjunto de caracteres que está em algum lugar na memória.

Claro que tem muito mais coisa abaixo desta explicação como constantes, literais estáticas, mas você pegou a ideia.

Resumindo, quando declaramos uma string, dizemos “Olha, do endereço 1 ao 27 estão as informações desta string”.

E qual problema de ter uma referência nula?

O maior problema em ter uma referência nula é tentar acessar seu valor. Sempre que fazemos isto no .NET, recebemos uma exceção do tipo **NullReferenceException**.

Um caso clássico é quando estamos trabalhando com listas e esquecemos de inicializá-las:

```
public class Order
{
  public List<Product> Products { get; set: ]
  // ⚠ Lista não inicializada, vai dar erro
}

var order = new Order();
order.Products.Add(product);
// ❌ NullReferenceException: Products é NULL
```

Isto vale para todo tipo de referência, e como consequência disso, temos que sempre verificar se o objeto está apontando para nulo antes de informar algum valor ou tentar acessar sua informação.

Resolvendo o problema com NULLs

O .NET resolveu incluir a opção que mencionei no começo do artigo para nos avisar de possíveis referências nulas e assim podermos tratá-las ainda em tempo de compilação.

Para seguir adiante, vamos tomar a classe abaixo como exemplo:

```
public class Student
{
  public string Name { get; set; }
  // ⚠ Non-nullable property 'Name' is uninitialized
}
```

Inicializando as propriedades

Conforme vimos, como `Name` é do tipo `string`, precisamos inicializá-lo, e para tipos primitivos, isto é relativamente simples.

```
public class Student
{
  public string Name { get; set; } = string.Empty;
  // ✅ OK
}

public class Student
```

```
{
  public string Name { get; set; } = "André";
  // ✅ OK
}
```

Também podemos fazer a inicialização via construtor, o compilador consegue identificar que a propriedade não foi inicializada imediatamente, mas será quando o objeto for construído.

```
public class Student
{
  public Student()
  {
    Name = string.Empty;
    // ✅ OK
  }

  public string Name { get; set; } // ✅ OK
}
```

É importante salientar que caso tenha múltiplos construtores, você precisa inicializar sua propriedade em todos eles. Então talvez seja mais fácil utilizar o primeiro exemplo desta sessão.

```
public class Student
{
  protected Student()
  {
    // ⚠ Continuará com o Warning pois não foi inicializado aqui.
  }

  public Student(string name)
  {
    Name = name;
    // ✅ OK
  }

  public string Name { get; set; } // ⚠
}
```

Nullable Types

E se por acaso eu quiser que a propriedade `Name` seja nula? Isto é possível e utilizado principalmente quando trabalhamos com **Entity Framework** por exemplo.

Neste caso, o .NET inclui uma opção bem simples e elegante, bastando adicionar `?` ou utilizar `Nullable<T>` na propriedade.

```
public Nullable<DateTime> UpdatedAt { get; set; }
public DateTime? UpdatedAt { get; set; }
```

Tanto faz utilizar `Nullable<DateTime>` ou `DateTime?`, o `?` é apenas um syntax sugar (enfeite ou apelido) para o `Nullable<DateTime>`.

Em resumo, sempre que você quiser deixar explícito que um tipo pode receber um valor nulo, utilize o ? na frente do tipo.

```
public class Student
{
    public Name? Name { get; set; }
    public DateTime? UpdatedAt { get; set; }
}

public class Name
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
}
```

Desta forma você também não terá o aviso do compilador, já que você mesmo explicitou no código que quer receber valores nulos para aquelas propriedades.

Null NOT

Como nada na vida é simples, vamos supor que temos a seguinte situação:

```
public class Student
{
    public Name Name { get; set; }
    // ⚠ Non-nullable property 'Name' is uninitialized
    public DateTime? UpdatedAt { get; set; }
}

public class Name
{
    public Name(string firstName, string lastName)
    {
        if (string.IsNullOrEmpty(firstName) ||
            string.IsNullOrEmpty(lastName))
            throw new ArgumentNullException("First name and last name are required");

        FirstName = firstName;
        LastName = lastName;
    }
```

```
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Neste cenário, temos uma classe que define o nome e utilizamos ela na classe `Student`, cujo o nome ***NÃO PODE SER NULO***.

Para piorar, não conseguimos instanciar um `Name` sem informar o primeiro e último nome, que são obrigatórios devido a falta do construtor sem parâmetros (parameterless) que o `Name` possui.

```
public Name Name { get; set; } = new();
// ❌ Não funciona pois precisamos do primeiro e último nome para instanciar o Name

public Name Name { get; set; } = new("", "");
// ❌ Não podemos inicializar com strings vazias

public Name Name { get; set; } = new("Bruce", "Wayne");
// ❌ Não queremos ter um valor padrão
```

Este cenário é a definição perfeita para o uso do ***NULL NOT*** ou `null!` (Null com uma exclamação na frente).

O ***NULL NOT*** nos permite dizer ao compilador o seguinte:

> *Esta propriedade não pode ser nula mas eu não tenho um valor para ela neste momento. No entanto eu me comprometo a inicializar ela antes de utilizá-la.*

Basicamente nós puxamos a responsabilidade e garantimos que este valor, embora nulo no momento, não será no futuro, que ele será inicializado.

```
public Name Name { get; set; } = null!;
// ✅ OK
```

É importante frisar que o **NULL NOT** não tem relação com o **NOT NULL**. O **NOT NULL** é um comparador que possui implementação e finalidade diferente.

```
if(name is not null)
  ...

if(name != null)
  ...
```

Conclusão

Entender como os valores nulos são criados e onde são utilizados é fundamental, bem como manter as configurações que ajudam o compilador a identificar possíveis referências nulas.

O .NET possui recursos incríveis tanto para manipulação de tipos nulos quanto para tratativa de referências nulas.

Utilize o compilador e as configurações ao seu favor, otimize seu código, evite excesso de condicionais e divirta-se com o conteúdo que aprendeu aqui 🚀

Sobre o autor

André Baltieri

Me dedico ao desenvolvimento de software desde 2003, sendo minha maior especialidade o Desenvolvimento Web.Durante esta jornada pude trabalhar presencialmente aqui no Brasil e Estados Unidos, atender remotamente times da Índia, Inglaterra e Holanda, receber **10x Microsoft MVP** e realizar diversas consultorias em empresas e projetos de todos os tamanhos.