

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/369142860>

# Apostila Desenvolvimento de API RESTful com .NET e C# MÓDULO I: FUNDAMENTOS DE PROGRAMAÇÃO PARA A CAMADA BACK-END

Technical Report · March 2023

---

CITATIONS

0

READS

2,438

1 author:



Bruno C H Silva

Federal University of Ceará

29 PUBLICATIONS 29 CITATIONS

[SEE PROFILE](#)

# **Apostila**

## **Desenvolvimento de API RESTful com .NET e C#**

**MÓDULO I: FUNDAMENTOS DE PROGRAMAÇÃO  
PARA A CAMADA BACK-END**

**Prof. Bruno Honorato**

<https://www.linkedin.com/in/bruno-honorato-3abb01180/>

Este documento visa fornecer uma introdução teórica e prática à programação de uma API na plataforma .NET (leia-se *dotnet*, uma vez que ‘dot’ é o equivalente para a palavra ‘ponto’ em inglês) com C#. Utilizamos também os princípios *Representational State Transfer* (REST) e SOLID no desenvolvimento da nossa API. SOLID é um acrônimo para: ***Single responsibility; Open-closed; Liskov substitution; Interface segregation; Dependency inversion.*** Cada um destes conceitos é abordado neste documento de forma a possibilitar que o leitor saiba aplicá-los no desenvolvimento de uma API RESTful. A estrutura de seções deste documento é a seguinte: introdução conceitual ao .NET; introdução ao C#; conceitos de SOLID; e, criação do projeto de uma API RESTful com C#.

Esta apostila está estruturada no seguinte arranjo de seções:

- 1. Introdução conceitual ao .NET**
- 2. Introdução ao C#**
- 3. Conceitos de SOLID**
- 4. Programando uma API REST em C#**

## 1. Introdução conceitual ao .NET

.NET é uma plataforma de código aberto e sem restrições de uso voltada para o desenvolvimento de *software*. Foi lançada pela Microsoft, em Novembro de 2002, para construir os mais variados tipos de aplicações. Até pouco depois de seu lançamento, a plataforma era conhecida como .NET Framework e seu propósito era o de fornecer um *framework* para desenvolvimento de aplicações *desktop* no Windows.

Atualmente, a plataforma .NET é composta por:

- *Common Language Runtime* (CLR): de forma resumida é um "Execution Engine" contendo tudo que é necessário para executar as aplicações .NET compiladas em *Intermediate Language* (IL). Quando a aplicação .NET é compilada, o compilador não gera diretamente um executável nativo em linguagem de máquina, mas sim, arquivos em uma linguagem intermediária chamada *Microsoft Intermediate Language* (MSIL). O CLR inclui o compilador, o mecanismo de coleta de lixo, etc. Ele recebe os arquivos em MSIL, utiliza um compilador para transformar em código nativo e permite a execução da aplicação.
- *Base Class Library* (BCL): é a biblioteca básica de classes incluída com a plataforma. Quando o .NET é instalado na máquina, toda a BCL é copiada para o *Global Assembly Cache* (GAC). As aplicações .NET quando dependem de um *Assembly* básico procuram no GAC e assim tem acesso a versão da BCL instalada com a plataforma.
- Software de suporte: responsável por inicializar o CLR, prover a interface com o Sistema Operacional, etc.

Atualmente, a plataforma .NET permite que você possa programar aplicativos nas linguagens: C#, F# ou Visual Basic. C# é uma linguagem de programação simples, moderna, orientada a objetos e segura. F# é uma linguagem de programação funcional que também inclui suporte a orientação a objetos e instruções imperativas. *Visual Basic* é uma linguagem fracamente tipada com uma sintaxe simples para criar aplicativos orientados a objetos.

Esteja você trabalhando em C#, F# ou *Visual Basic*, seu código será executado nativamente em qualquer sistema operacional compatível. Em versões anteriores da plataforma, ao optar por .NET, o programador deve decidir uma dentre três implementações desta plataforma:

- .NET Core: é uma implementação multi-plataforma do .NET para programação de sites, servidores e aplicativos de console. Multi-plataforma quer dizer que o código fonte programado poderá ser executado em qualquer Sistema Operacional (SO) que suporte o .NET Core. Destes SO's, destacam-se Windows, Linux e MacOS;
- .NET Framework: oferece suporte ao desenvolvimento de sites, serviços, aplicativos de desktop e outros tipos de aplicações que deverão rodar apenas em Windows;
- *Xamarin/Mono*: é uma implementação .NET otimizada para a execução de aplicativos em todos os principais sistemas operacionais móveis.

Apesar da estratégia de as múltiplas implementações .NET atender ao problema de otimizar a plataforma para um determinado propósito, isso resultou em um novo problema para os projetistas da plataforma .NET: codificar biblioteca de classes reutilizáveis que funcionem em mais de uma plataforma. A solução veio com .NET Standard, um conjunto básico de APIs comuns a todas as implementações .NET.

Cada implementação também pode expor APIs adicionais que são específicas aos sistemas operacionais em que é executada. Por exemplo, o .NET Framework é uma implementação do .NET somente para Windows que inclui APIs para acessar o Registro do Windows.

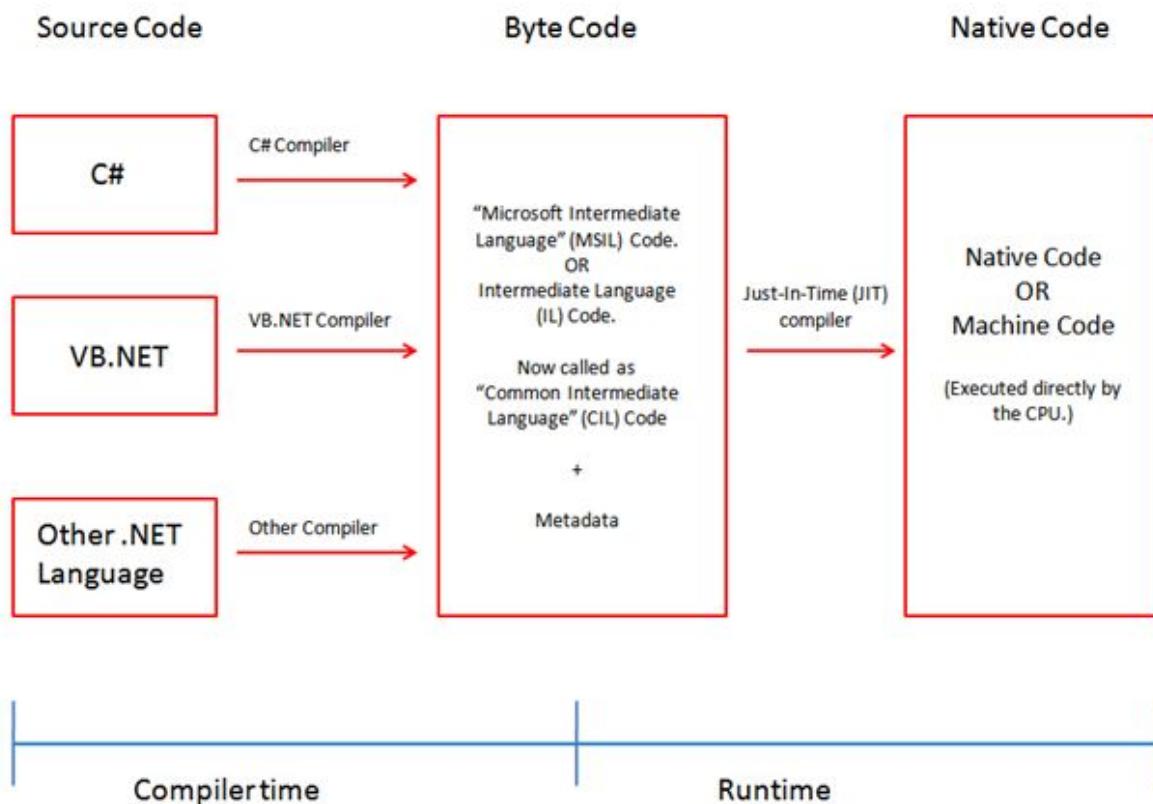
Para estender a funcionalidade, a Microsoft e outros mantêm um ecossistema de pacotes saudável construído em .NET Standard. O NuGet é o gerenciador de pacotes desenvolvido especificamente para .NET. Ele contém mais de 90.000 pacotes. E há também as poderosas *Integrated Development Environments* (IDEs) da Microsoft para se programar com .NET: Visual Studio; e, Visual Studio Code.

## 2. Introdução ao C#

C# (leia-se “C-Sharp”) é uma linguagem de programação desenvolvida pela Microsoft por uma equipe liderada por Anders Hejlsberg e Scott Waltemath, que foi projetada especificamente para a plataforma .NET para ajudar na migração para o .NET de aplicações codificadas em outras linguagens de alto nível. A ideia inicial do projeto do C#

era o de mesclar os recursos de C, C ++ e Java, adaptando os melhores recursos de cada uma destas linguagens e adicionando novos recursos próprios.

Quando o código fonte C# é compilado, o compilador processa seu código para o código multi-plataforma IL. O código IL é transscrito em código de máquina pelo CLR. Este processo é denominado *Just-in-time Compilation* (JIT) e ilustrado como se segue:



Nesta seção, os exemplos de codificação ilustrados foram programados na IDE *Visual Studio*. O instalador do Visual Studio pode ser obtido em:

<https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>

Usuários de SO Linux podem recorrer ao Visual Studio Code:

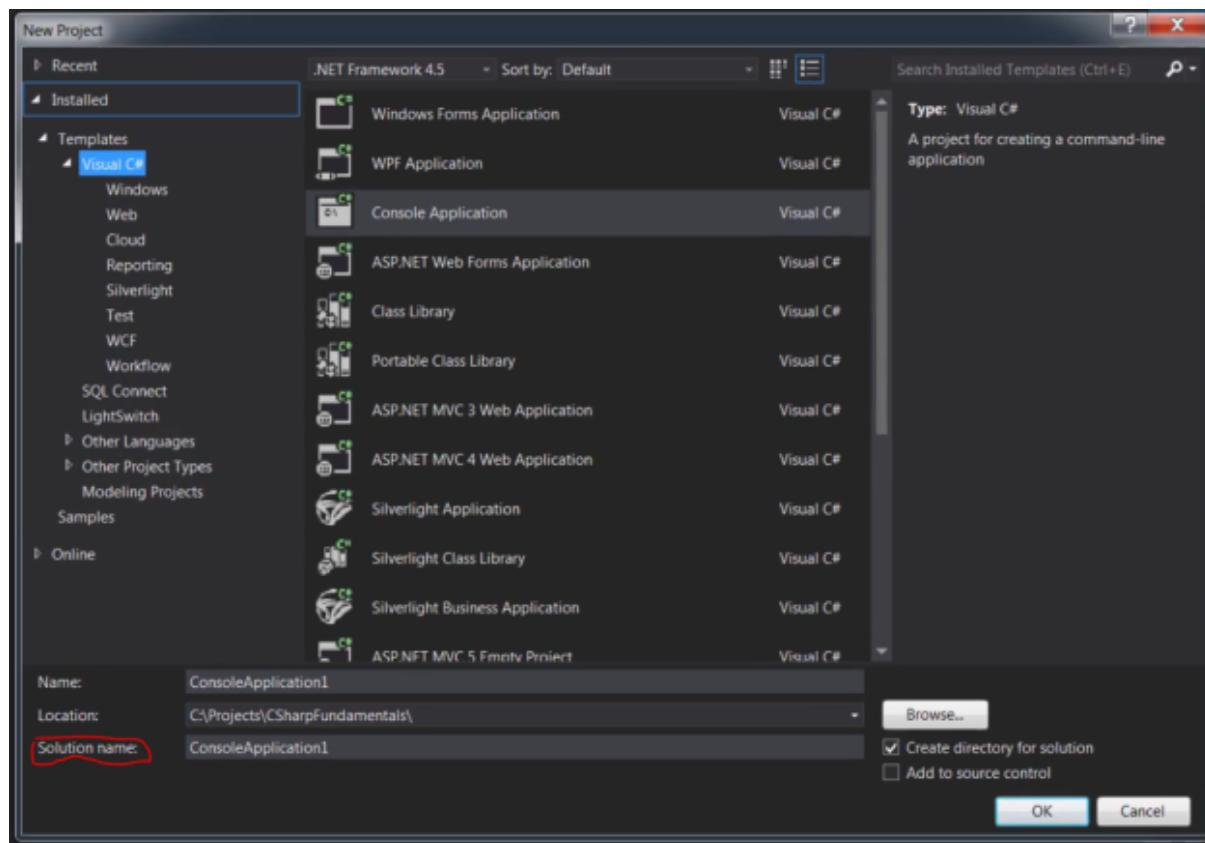
[https://code.visualstudio.com/?wt.mc\\_id=vscom\\_downloads](https://code.visualstudio.com/?wt.mc_id=vscom_downloads)

É necessário também instalar o .NET. Essa implementação da plataforma .NET pode ser obtida:

<https://dotnet.microsoft.com/download>

<https://docs.microsoft.com/en-us/dotnet/core/install/linux-ubuntu>

Seguidas estas orientações de instalação, ao abrir o Visual Studio, criar um novo projeto do tipo Console Application tal qual é ilustrado como se segue:



Aplicações do tipo Console Application são uma ótima ferramenta de aprendizagem, pois não há criação de recursos avançados na geração deste template.

Usuários do SO Linux podem recorrer a este tutorial para configurar o VS Code para programar em C#:

<https://www.youtube.com/watch?v=Ifw-Sp6N9UA>

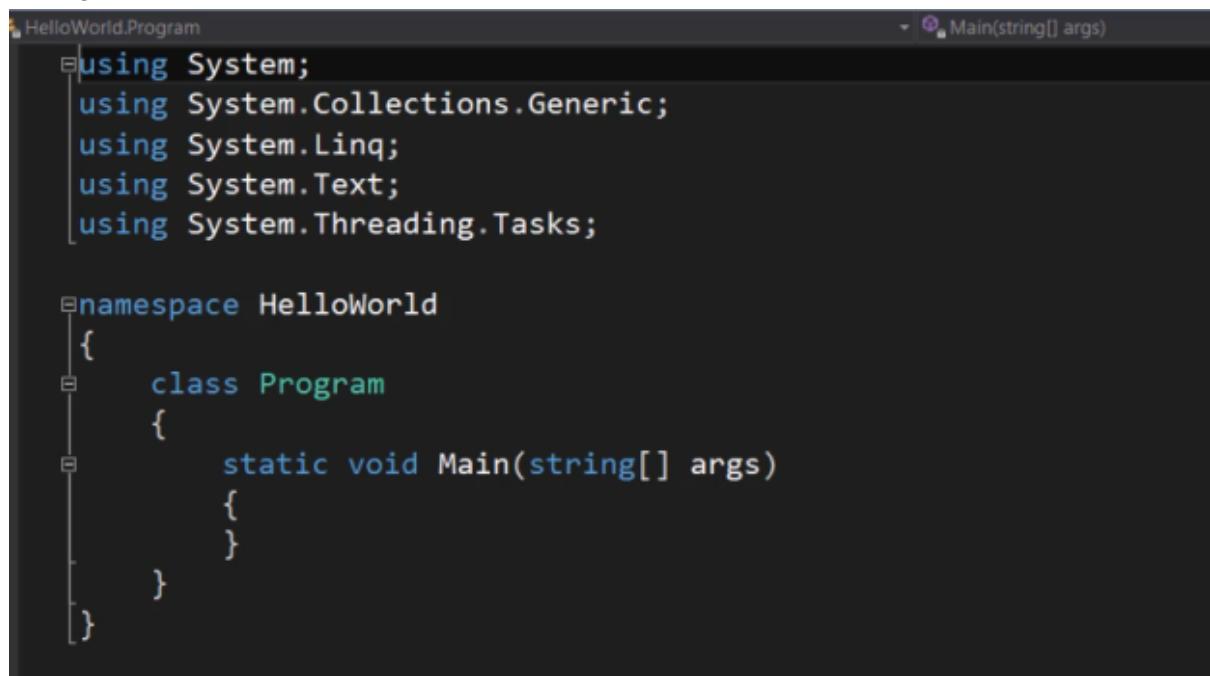
Desenvolvedores de primeira viagem para com o Visual Studio estranham os muitos painéis e ícones exibidos. Mas a verdade é que 90% do tempo, tudo o que você precisa é do editor de código e do explorador de soluções, e se você não conseguir encontrar seu explorador de soluções (*Solution Explorer*), vá para Exibição (ou *View*) > Explorador de soluções.

O *Solution Explorer* contém nosso projeto, neste caso denominado *HelloWorld*, References que faz parte da estrutura .NET usada para adicionar referências às classes que você pode estar usando no aplicativo. App.config é um arquivo no formato *Extensible Markup Language (XML)* onde armazenamos configurações para este aplicativo, ou *strings* de conexão para o banco de dados, ou qualquer configuração que você deseja armazenar para seu aplicativo, e, por último, Program.cs que é uma classe onde iremos começar a escrever o código.

Se você está tendo problemas com o manuseio do VS Studio, esse vídeo ensina como configurar uma aplicação com .NET junto ao Visual Studio Code:

[https://www.youtube.com/watch?v=MAXHPW9-q3g&ab\\_channel=TheSolutionArchitect](https://www.youtube.com/watch?v=MAXHPW9-q3g&ab_channel=TheSolutionArchitect)

Em Program.cs, temos um *namespace* chamado *HelloWorld*, que é criado pelo Visual Studio usando o nome que demos ao nosso primeiro aplicativo de console. Em C#, temos acesso a qualquer classe neste *namespace*, mas se quisermos acessar qualquer classe fora desse *namespace*, precisamos importá-la usando a palavra-chave **using** e chamando o *namespace* onde a classe reside. O conceito de *namespace* é semelhante ao de pacotes do Java, porém, diferentemente dos pacotes, um *namespace* do C# não impõe a necessidade de seguir uma estrutura de arquivos.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

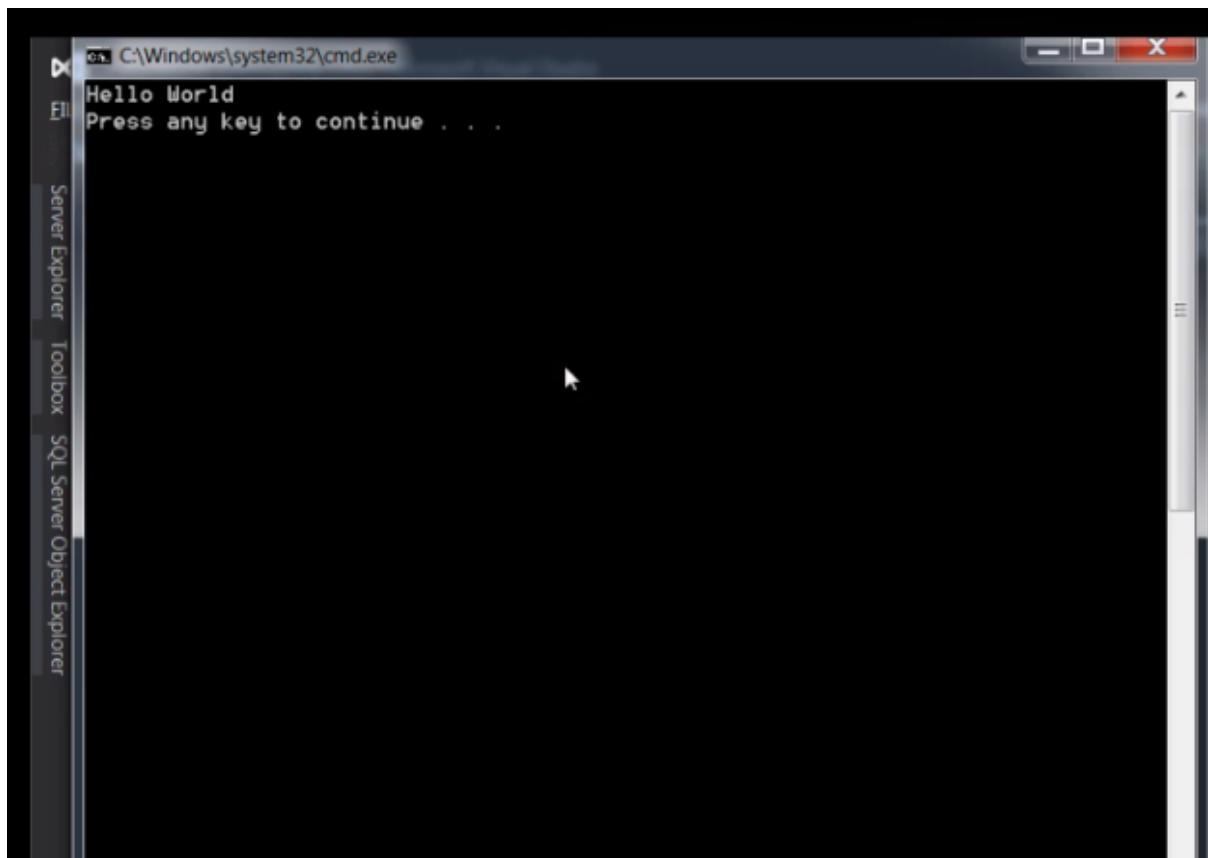
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Ainda sobre Program.cs, temos:

- *System*: este é um *namespace* do .NET que abriga todas essas classes de utilitários básicos e classes de tipos primitivos;
- *System.Collections.Generic*: é usado para trabalhar com coleções de lista e assim por diante;
- *System.Linq*: é usado para trabalhar com dados e realizar consultas nativas;
- *System.Text*: é usado para trabalhar com texto, codificação etc;
- *System.Threading.Task*: é usado para construir aplicativos *multithreading*.

Em C#, membros de classes devem ter seus identificadores sempre começando com a letra maiúscula. Em Java, por exemplo, os nomes dos membros começam com a letra

minúscula. O método **Main** também funciona como ponto de execução da aplicação, assim como C, C++ e Java. Escrevendo o comando **Console.WriteLine ("Hello World")**; dentro do método **Main**, ao executar o programa, deve-se obter a seguinte saída:



A seguir, são apresentados os principais conceitos para se programar em C#. O roteiro é o seguinte: tipos e expressões primitivas; tipos não primitivos; controle de fluxo; listas; trabalhando com datas; trabalhando com cadeias de caracteres; trabalhando com arquivos; classes; Herança; Encapsulamento; Polimorfismo; *interface*; *Generics*; expressões *lambda*; *LINQ*; tipos anuláveis; *Dynamics*; manipulação de exceção; e, programação assíncrona.

## 2.1 Tipos e expressões primitivas

### 2.1.1 Variáveis e constantes

Variáveis são nomes dados a locais de armazenamento na memória. Usamos variáveis para armazenar valores temporariamente na memória. Em C #, para declarar uma variável, começamos com um tipo de dados seguido por um identificador e terminamos com um ponto e vírgula (;) conforme mostrado abaixo

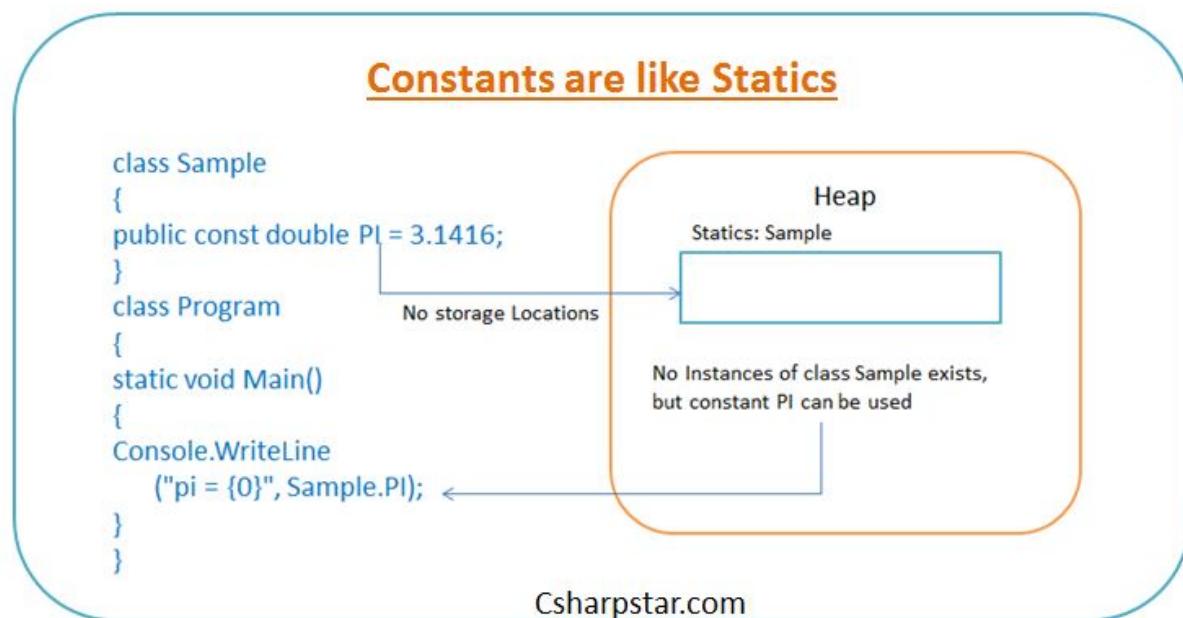
```

int number;
—
int Number = 1;

```

## 2.1.2 Constantes

Constantes são valores imutáveis, i. e., em tempo de execução esses valores não mudam. Imagine que em sua aplicação, você tem este número chamado **Pi = 3,14** usado para calcular a área de um círculo. Esse número deve ser definido como uma constante com a diretiva **const**.



A tipagem primitiva de variáveis e constantes é dada a seguir:

	C# Type	.NET Type	Bytes	Range
Integral Numbers	<b>byte</b>	Byte	1	0 to 255
	<b>short</b>	Int16	2	-32,768 to 32,767
	<b>int</b>	Int32	4	-2.1B to 2.1B
	<b>long</b>	Int64	8	...
Real Numbers	<b>float</b>	Single	4	$-3.4 \times 10^{38}$ to $3.4 \times 10^{38}$
	<b>double</b>	Double	8	...
	<b>decimal</b>	Decimal	16	$-7.9 \times 10^{28}$ to $7.9 \times 10^{28}$
Character	<b>char</b>	Char	2	Unicode Characters
Boolean	<b>bool</b>	Boolean	1	True / False

Eis um código de apoio:

```
using System;

namespace Variables
{
    class Program
    {
        static void Main(string[] args)
        {
            byte number = 2;
            int count = 10;
            float totalPrice = 20.95f;
            char character = 'A';
            string firstName = "Mosh";
            bool isWorking = false;

            Console.WriteLine(number);
            Console.WriteLine(count);
            Console.WriteLine(totalPrice);
            Console.WriteLine(character);
            Console.WriteLine(firstName);
            Console.WriteLine(isWorking);
        }
    }
}
```

(local variable) bool isWorking

Em C #, temos uma palavra-chave chamada **var** que torna as declarações de variáveis mais fáceis. Por exemplo, no código acima, em vez de colocar explicitamente o tipo de dados, posso substituí-los por **var**, que em tempo de execução mapeará para seu tipo de dados correspondente. O exemplo acima com **var** ficaria assim:

```
using System;

namespace Variables
{
    class Program
    {
        static void Main(string[] args)
        {
            var number = 2;
            var count = 10;
            var totalPrice = 20.95f;
            var character = 'A';
            var firstName = "Mosh";
            var isWorking = false;

            struct System.Boolean
            Represents a Boolean value.
            in(number);
            Console.WriteLine(count);
            Console.WriteLine(totalPrice);
            Console.WriteLine(character);
            Console.WriteLine(firstName);
            Console.WriteLine(isWorking);
        }
    }
}
```

Assim como em Java, também podemos processar conversões entre variáveis de tipos diferentes. Para tal, utilizamos a interface da classe **Convert**. Na literatura, chamamos de interface de uma classe os seus métodos. Exemplo de conversão de tipos:

```
Program.cs  C# ConvertTo2
using System;

namespace ConvertTo2
{
    class Program
    {
        static void Main(string[] args)
        {
            int result;
            string str = "3456";
            result = Convert.ToInt32(str);
            Console.WriteLine(result);
            Console.ReadLine();
        }
    }
}
```

C:\Program Files\dotnet\dotnet.exe  
3456

## 2.1.2 Operadores

Os operadores aritméticos são:

### Arithmetic Operators

	Operator	Example
Add	+	a + b
Subtract	-	a - b
Multiply	*	a * b
Divide	/	a / b
Remainder	%	a % b

Os operadores de incremento e decremento são:

	Operator	Example	Same as
Increment	<code>++</code>	<code>a++</code>	<code>a = a + 1</code>
Decrement	<code>--</code>	<code>a--</code>	<code>a = a - 1</code>

Já os operadores relacionais são:

	Operator	Example
Equal	<code>==</code>	<code>a == b</code>
Not Equal	<code>!=</code>	<code>a != b</code>
Greater than	<code>&gt;</code>	<code>a &gt; b</code>
Greater than or equal to	<code>&gt;=</code>	<code>a &gt;= b</code>
Less than	<code>&lt;</code>	<code>a &lt; b</code>
Less than or equal to	<code>&lt;=</code>	<code>a &lt;= b</code>

Temos como operadores lógicos, os seguintes:

	Operator	Example
And	<code>&amp;&amp;</code>	<code>a &amp;&amp; b</code>
Or	<code>  </code>	<code>a    b</code>
Not	<code>!</code>	<code>!a</code>

E por fim, os operadores de atribuição são:

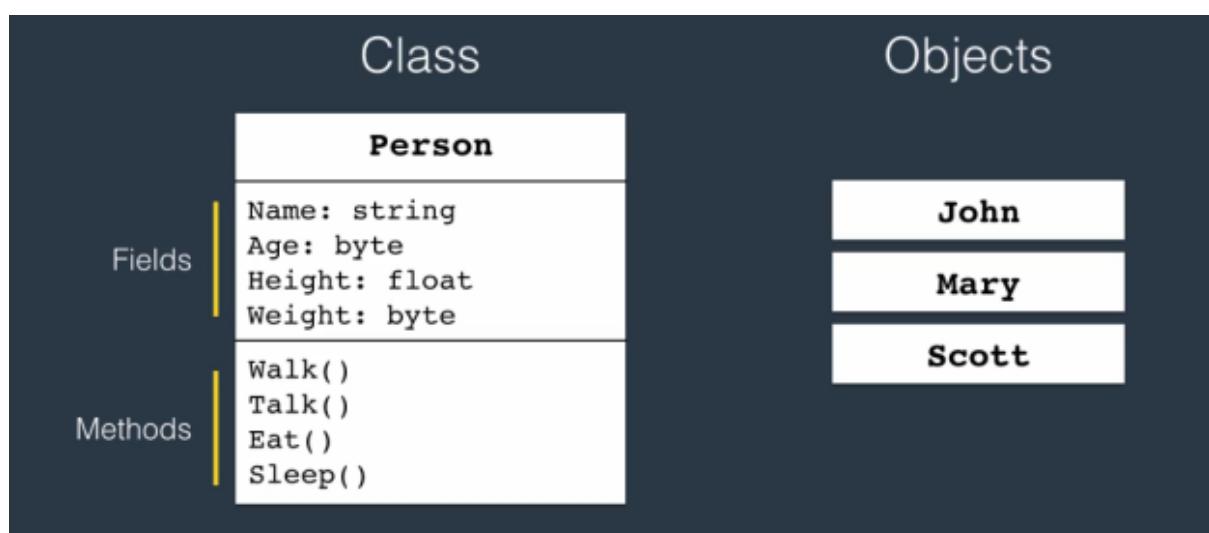
	Operator	Example	Same as
Assignment	=	a = 1	
Addition assignment	+=	a += 3	a = a + 3
Subtraction assignment	-=	a -= 3	
Multiplication assignment	*=	a *= 3	
Division assignment	/=	a /= 3	

## 2.2 Tipos Não Primitivos

Temos os seguintes tipos não primitivos em C#: **class**, **struct**, **array**, **String** e **enum**.

### 2.2.1 Classes

Uma **class** pode ser compreendida como um bloco de construção de um aplicativo. Combinam atributos que irão refletir o estado da respectiva classe ou das instâncias desta classe, e métodos, os quais definem a interface de comunicação e uso da classe e suas instâncias. Lembre-se que instância de classe é sinônimo de objeto da classe. Exemplos:



```
using System;

namespace CSharpFundamentals
{
    public class Person
    {
        public string FirstName;
        public string LastName;

        public void Introduce()
        {
            Console.WriteLine("My name is " + FirstName + " " + LastName);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var john = new Person();
            john.FirstName = "John";
            john.LastName = "Smith";
            john.Introduce();
        }
    }
}
```

## 2.2.2 Struct

Uma **struct** (estrutura) é um tipo com características semelhantes a uma classe. Ele combina campos e métodos relacionados. Em termos de sintaxe, é semelhante, mas a diferença é que em vez de usar a palavra-chave **class**, emprega-se a palavra-chave **struct**.

Use uma **struct** apenas ao criar objetos *lightweight* (ou leves). Isso refletirá em uma otimização de desempenho útil quanto ao consumo de memória pela aplicação. No .NET, todos os tipos primitivos são definidos como estruturas. Eles são pequenos e leves. O maior tipo primitivo não possui um tamanho superior a 16 bytes.

Diferente da linguagem C, é possível definir métodos para a uma **struct**. É possível definir também um método construtor e sobrecrevê-lo. A sobreulação do construtor só será aceita sintaticamente se todas as variáveis da **struct** forem inicializadas, seja por um parâmetro ou por uma instrução dentro do escopo do método.

The screenshot shows a code editor window with a C# file named 'CSharpStructureTutorial.cs'. The code defines a struct 'Employee' with three fields: 'empNumber', 'empName', and 'position'. It has two constructors: one taking three parameters and another taking two parameters. The second constructor has a warning '2 References' above it. A red arrow points from the text 'Error because the 'position' field is not assigned a value.' to the line 'this.empName = empName;' in the second constructor. The code editor interface includes a status bar at the bottom left showing '100 %'.

```
namespace CSharpStructureTutorial
{
    struct Employee
    {
        public string empNumber;
        public string empName;
        public string position;

        1 reference
        public Employee(string empNumber, string empName, string position)
        {
            this.empNumber = empNumber;
            this.empName = empName;
            this.position = position;
        }

        2 References
        public Employee(string empNumber, string empName )
        {
            this.empNumber = empNumber;
            this.empName = empName;
        }
    }
}
```

### 2.2.3 Array

Um **array** é usado para armazenar uma coleção de variáveis do mesmo tipo. É uma estrutura de dados elementar. Exemplo:

```
using System;

namespace CSharpFundamentals
{
    class Program
    {
        static void Main(string[] args)    I
        {
            var numbers = new int[3];
            numbers[0] = 1;

            Console.WriteLine(numbers[0]);
            Console.WriteLine(numbers[1]);
            Console.WriteLine(numbers[2]);

            var flags = new bool[3];
            flags[0] = true;

            Console.WriteLine(flags[0]);
            Console.WriteLine(flags[1]);
            Console.WriteLine(flags[2]);

            var names = new string[3] {"Jack", "John", "Mary"};
        }
    }
}
```

## 2.2.4 String

Uma **String** é uma sequência de caracteres. Em C#, uma **String** é cercada por aspas duplas. Exemplo:

```
namespace CSharpFundamentals
{
    class Program
    {
        static void Main(string[] args)
        {
            var firstName = "Mosh";
            var lastName = "Hamedani";

            var fullName = firstName + " " + lastName;

            var myFullName = string.Format("My name is {0} {1}", firstName, lastName);

            var names = new string[3] { "John", "Jack", "Mary" };
            var formattedNames = string.Join(", ", names);

            var text = @ "Hi John
Look into the following paths
c:\folder1\folder2
c:\folder3\folder4";
            Console.WriteLine(text);
        }
    }
}
```

## 2.2.5 Enum

Um **enum** é um tipo de dados que contém um conjunto de pares nome/valor. Este tipo é usado quando você precisa definir várias constantes relacionadas. Exemplo:

```
0 references
class Program
{
    3 references
    public enum DayOfWeek
    {
        Sunday = 0,
        Monday = 1,
        Tuesday = 2,
        Wednesday = 3,
        Thursday = 4,
        Friday = 5,
        Saturday = 6
    }

    0 references
    static void Main(string[] args)
    {
        foreach (DayOfWeek ObjDayOfWeek in DayOfWeek)
        {
            DoSomething(ObjDayOfWeek);
        }
    }
}

1 reference
private static void DoSomething(DayOfWeek ObjDayOfWeek)
{
    //Processing
}
```

A screenshot of a code editor showing a C# program. The code defines an enum named 'DayOfWeek' with values Sunday through Saturday. In the 'Main' method, there is a foreach loop that iterates over 'DayOfWeek'. A tooltip appears over the 'DayOfWeek' type in the loop, showing the enum definition and an error message: "'Enumerate\_Enum\_csharp.Program.DayOfWeek' is a 'type' but is used like a 'variable'".

## 2.3 Controle de fluxo

O controle de fluxo condicional e de repetição em C# segue o mesmo que o de linguagens como C, C++ e Java. Eis exemplos:

```
using System;

namespace Conditionals
{
    class Program
    {
        static void Main(string[] args)
        {
            int hour = 10;

            if (hour > 0 && hour < 12)
            {
                Console.WriteLine("It's morning.");
            }
            else if (hour >= 12 && hour < 18)
            {
                Console.WriteLine("It's afternoon.");
            }
            else
            {
                Console.WriteLine("It's evening.");
            }
        }
    }
}
```

```
using System;

namespace Iterations
{
    class Program
    {
        static void Main(string[] args)
        {
            for (var i = 1; i <= 10; i++)
            {
                if (i%2 == 0)
                {
                    Console.WriteLine(i);
                }
            }
        }
    }
}
```

```
2
4
6
8
10
Press any key to continue . . .
```

```
using System;

namespace Iterations
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[] {1, 2, 3, 4};

            foreach (var number in numbers)
            {
                Console.WriteLine(number);
            }
        }
    }
}
```

```

using System;

namespace Iterations
{
    class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                Console.WriteLine("Type your name: ");
                var input = Console.ReadLine();

                if (String.IsNullOrWhiteSpace(input))
                    break;

                Console.WriteLine("@Echo: " + input);
            }
        }
    }
}

```

## 2.4 Listas

Esta é uma estrutura de dados elementar fornecida pelo C# para armazenar vários objetos do mesmo tipo. A diferença entre um **array** e uma lista é que o **array** tem tamanho fixo e o tamanho de uma lista é dinâmico. Outro recurso de listas é o suporte a tipos genéricos (do inglês *Generics*). Eis um exemplo:

```

static void Main(string[] args)
{
    var numbers = new List<int>() { 1, 2, 3, 4 };
    numbers.Add(1);
    numbers.AddRange(new int[3] { 5, 6, 7 });

    foreach (var number in numbers)
        Console.WriteLine(number);

    Console.WriteLine();
    Console.WriteLine("Index of 1: " + numbers.IndexOf(1));
    Console.WriteLine("Last Index of 1: " + numbers.LastIndexOf(1));

    Console.WriteLine("Count: " + numbers.Count);

    for (var i = 0; i < numbers.Count; i++)
    {
        if (numbers[i] == 1)
            numbers.Remove(numbers[i]);
    }
    foreach (var number in numbers)
        Console.WriteLine(number);

    numbers.Clear();
    Console.WriteLine("Count: " + numbers.Count);
}

```

## 2.5 Trabalhando com datas

Em C#, a classe básica para se trabalhar com datas é o `DateTime`. Indicasse `TimeSpan` para se trabalhar com tempo (horas, minutos e segundos). Após instanciadas, os objetos dessas classes são imutáveis e modificações devem ocorrer apenas por meio da interface destes objetos. Exemplos de uso:

```
static void Main(string[] args)
{
    var dateTime = new DateTime(2015, 1, 1);
    var now = DateTime.Now;
    var today = DateTime.Today;

    //Console.WriteLine("Hour: " + now.Hour);
    //Console.WriteLine("Minute: " + now.Minute);

    var tomorrow = now.AddDays(1);
    var yesterday = now.AddDays(-1);

    Console.WriteLine(now.ToString("yyyy-MM-dd HH:mm"));
    Console.WriteLine(now.ToShortDateString());
    Console.WriteLine(now.ToLongDateString());
    Console.WriteLine(now.ToString("HH:mm:ss"));
    Console.WriteLine(now.ToString("yyyy-MM-dd HH:mm"));
}

class Program
{
    static void Main(string[] args)
    {
        // Creating
        var timeSpan = new TimeSpan(1, 2, 3);

        var timeSpan1 = new TimeSpan(1, 0, 0);
        var timeSpan2 = TimeSpan.FromHours(1);

        var start = DateTime.Now;
        var end = DateTime.Now.AddMinutes(2);
        var duration = end - start;
        Console.WriteLine("Duration: " + duration);

        // Properties
        Console.WriteLine("Minutes: " + timeSpan.Minutes);
        Console.WriteLine("Total Minutes: " + timeSpan.TotalMinutes);

        // Add
        Console.WriteLine("Add Example: " + timeSpan.Add(TimeSpan.FromMinutes(8)));
    }
}
```

## 2.6 Trabalhando com cadeias de caracteres

Os exemplo a seguir demonstra o uso de métodos clássicos da classe `String`:

```
namespace examples
{
    class Program
    {
        static void Main(string[] args)
        {
            var fullName = "Dennis Sharp ";
            Console.WriteLine("Trim: '{0}'", fullName.Trim());
            Console.WriteLine("ToUpper: '{0}'", fullName.Trim().ToUpper());
            var index = fullName.IndexOf(' ');
            var firstName = fullName.Substring(0, index);
            var lastName = fullName.Substring(index + 1);
            Console.WriteLine("FirstName"+ firstName);
            Console.WriteLine("LastName"+ lastName);
            var names = fullName.Split(' ');
            Console.WriteLine("Firstnamw"+ names[0]);
            Console.WriteLine("LastName"+names[1]);
            Console.WriteLine( firstName.Replace("Dennis", "Dennifs"));
            if(String.IsNullOrEmpty(null))
                Console.WriteLine("invalid");
            var str = "23";
            var age = Convert.ToInt32(str);
            Console.WriteLine(age);
            float price = 78.95f;
            Console.WriteLine(price.ToString("C"));
            Console.ReadLine();
        }
    }
}
```

## 2.7 Trabalhando com arquivos

Em .NET, temos um *namespace* chamado `System.IO`, e é onde estão localizadas as classes com as quais trabalhar com arquivos e diretórios. Os exemplos que se seguem mostram como abrir, editar e obter informações de um arquivo:

```
class Program
{
    static void Main(string[] args)
    {
        var path = @"c:\somefile.jpg";

        File.Copy(@"c:\temp\myfile.jpg", @"d:\temp\myfile.jpg", true);
        File.Delete(path);
        if (File.Exists(path))
        {
            //
        }
        var content = File.ReadAllText(path);

        var fileInfo = new FileInfo(path);
        fileInfo.CopyTo("..."); 
        fileInfo.Delete();
        if (fileInfo.Exists)
        {
            //
        }
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        var path = @"C:\Projects\CSharpFundamentals\HelloWorld\HelloWorld.sln";

        var dotIndex = path.IndexOf('.');
        var extension = path.Substring(dotIndex);

        Console.WriteLine("Extension: " + Path.GetExtension(path));
        Console.WriteLine("File Name: " + Path.GetFileName(path));
        Console.WriteLine("File Name without Extension: " + Path.GetFileNameWithoutExtensio
        Console.WriteLine("Directory Name: " + Path.GetDirectoryName(path));

    }
}

```

## 2.8 Classes

C# fornece o mesmo suporte ao paradigma de orientação a objetos. Desse modo, podemos criar classes simples:

```

namespace examples
{
    0 references
    class Program
    {
        1 reference
        public class Person
        {
            public string Name;
            1 reference
            public void Introduce(string to )
            {
                Console.WriteLine("Hi {0}, I am {1}", to,Name);
            }
        }
        0 references
        static void Main(string[] args)
        {
            var person = new Person();
            person.Name= "Ann";
            person.Introduce("Dennis");
        }
    }
}

```

Podemos também optar pela sobrecarga de métodos:

```

public class Point
{
    public int X;
    public int Y;

    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }

    public void Move(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }

    public void Move(Point newLocation)
    {
        if (newLocation == null)
            throw new ArgumentNullException("newLocation");

        Move(newLocation.X, newLocation.Y);
    }
}

```

Podemos definir *getters* e *setters* de 3 formas:

```

public class Person
{
    private DateTime _birthdate;

    public void SetBirthdate(DateTime birthdate)
    {
        this._birthdate = birthdate;
    }

    public DateTime GetBirthdate()
    {
        return _birthdate;
    }
}

```

```
public class Person
{
    private DateTime _birthdate;
    public DateTime Birthdate
    {
        get { return _birthdate; }
        set { _birthdate = value; }
    }
}
```

```
public class Person
{
    public DateTime Birthdate { get; set; }
}
```

Sendo a sintaxe de ativação de *getters* e *setters* a seguinte:

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            var person = new Person();
            person.Birthdate = new DateTime(1982, 1, 1);
            Console.WriteLine(person.Age);
        }
    }
}
```

Ou seja, se um atributo possui *getters* e *setters*, eles são executados em tempo de execução sem que precisemos explicitar suas chamadas no código.

## 2.9 Herança

Herança é um princípio de orientação a objetos, que permite que classes compartilhem atributos e métodos. Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos. Ao invés do **extends** do Java, o operador de extensão do C# é o : (dois pontos). Exemplo:

```
1 using System.Text;
2 using System.Threading.Tasks;
3
4 namespace Inheritance
5 {
6     1 reference
7     class Vehicle
8     {
9         1 reference
10        public string name { get; set; }
11        0 references
12        public string brand { get; set; }
13
14        1 reference
15        public void move()
16        {
17            Console.WriteLine("vehicle in motion .");
18        }
19
20        0 references
21        public void Stop()
22        {
23            Console.WriteLine("vehicle in stationary state.");
24        }
25    }
26}
```

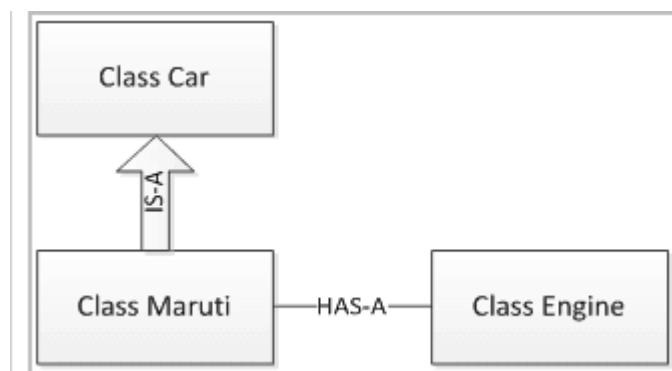
```
4 1 using System.Text;
5 2 using System.Threading.Tasks;
6
7 3 namespace Inheritance
8 4 {
9      1 reference
10     5 class Car:Vehicle
11     {
12         0 references
13         public double speed { get; set; }
14         0 references
15         public double distance { get; set; }
16         0 references
17         public double Time { get; set; }
18
19         0 references
20         public void calculateSpeed()
21         {
22             Console.WriteLine("vThe formular to calculate speed is distance/Time" );
23         }
24     }
25 }
```

```

namespace Inheritance
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            var car = new Car();
            car.name = "Toyota";
            car.move();
            System.Console.ReadLine();
        }
    }
}

```

C# também suporta Composição entre classes. Este é outro tipo de relacionamento que permite que uma classe contenha outra. Por exemplo, um carro tem um motor. Veja a notação UML deste exemplo:



Onde leia-se *IS-A* como *é-um* e *HAS-A* como *tem-um*. A implementação das entidades deste diagrama em classes do C# seriam:

```

2 references
class Engine
{
    1 reference
    public void start()
    {
        Console.WriteLine("Engine Started:");
    }
    0 references
    public void stop()
    {
        Console.WriteLine("Engine Stopped:");
    }
}

```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace examples
{
    class Maruti : Car
    {
        //Maruti extends Car and thus inherits all methods from Car (except final and static)
        //Maruti can also define all its specific functionality
        public void MarutiStartDemo()
        {
            Engine MarutiEngine = new Engine();
            MarutiEngine.start();
        }
    }
}
```

Sobre construtores, há duas coisas que você precisa saber quando se trata de herança. A primeira é que, na instanciação de um objeto, os construtores da classe base são sempre executados primeiro. E a segunda é que os construtores da classe base não são herdados. Exemplos de implementação:

```
public class Vehicle
{
    private string _registrationNumber;

    public Vehicle(string registrationNumber)
    {
        _registrationNumber = registrationNumber;
    }
}
```

Agora, digamos que queremos criar uma classe de carro que deriva de veículo. Usaremos a palavra-chave **base** para indicar essa derivação como mostrado abaixo.

```
public class Car : Vehicle
{
    public Car(string registrationNumber)
        : base(registrationNumber)
    {
        // Initialise fields specific to the Car class
    }
}
```

## 2.10 Encapsulamento

As classes de uma aplicação devem ser projetadas como uma caixa preta. Eles devem ter visibilidade limitada para o lado de fora. A implementação, o detalhe, devem ser ocultados. Em C#, usamos modificadores de acesso (principalmente privados) para fazer isso. Isso é conhecido como *Information Hiding* (e às vezes Encapsulation) na programação orientada a objetos. Os modificadores de acesso do C# são:

**public**: um membro declarado como público pode ser acessado em qualquer lugar. Veja o exemplo abaixo, o método de promoção é visível do lado de fora da classe Cliente

```
public class Customer
{
    public void Promote()
    {
    }
}

...

var customer = new Customer();
customer.Promote();
```

**private**: um membro declarado como privado só pode ser acessado pela classe. Veja o exemplo abaixo

```
public class Customer
{
    private int calculateRating()
    {
    }
}

...

var customer = new Customer();
customer.calculateRating();
```

**protected**: um membro declarado como protegido só pode ser acessado pela classe e suas classes derivadas.

## 2.11 Polimorfismo

*Poli* significa muitos enquanto *morfismo* significa formas, então polimorfismo significa *muitas formas*. No paradigma de orientação a objetos, o conceito de polimorfismo fornece a capacidade de uma classe assumir mais de uma forma.

O conceito de **sobrescrita** de método é um importante recurso do Polimorfismo. A sobrescrita implica em modificar a implementação de um método herdado. Se um método for declarado como **virtual** na classe base, podemos substituí-lo em uma classe derivada. Usando a palavra-chave de substituição como visto a seguir:

```
public class Shape
{
    public virtual void Draw()
    {
        // Default implementation
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        // New implementation
    }
}
```

Essa técnica nos permite alcançar o polimorfismo. O polimorfismo é uma ótima técnica orientada a objetos que permite o uso se livrar de longas instruções de **switch** procedurais (ou condicionais).

Há também o comando **abstract**, utilizado para indicar que não há coesão no domínio da aplicação em se instanciar uma classe, ou, que um método de uma classe deve ser obrigatoriamente sobreescrito pelas subclasses.

```
public abstract class Shape
{
    public abstract void Draw();
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Implementation for Circle
    }
}
```

```
var shape = new Shape(); // Won't compile
```

Portanto, as classes abstratas são inherentemente virtuais e podem fornecer comportamentos polimórficos, conforme discutimos anteriormente.

## 2.12 Interface

Uma **interface** é um tipo da linguagem C# semelhante a uma classe (em termos de sintaxe), mas é fundamentalmente diferente. Uma **interface** é simplesmente uma declaração dos recursos (ou serviços) que uma classe deve fornecer.

Para declarar um **interface** em vez de usar a palavra-chave **class**, usamos a palavra-chave **interface**. Em .NET, todo identificador de *interface* deve começar com a letra I. é uma convenção, conforme visto no exemplo adiante.

É importante salientar que:

- Uma **interface** é puramente uma declaração. Membros de uma **interface** não têm implementação.
- Uma **interface** só pode declarar métodos e propriedades, mas não campos (porque os campos são sobre detalhes de implementação).
- Os membros de uma **interface** não têm modificadores de acesso.

Programar para **interfaces** estimula o desacoplamento entre classes e facilita a testagem e avaliação da qualidade das funcionalidades da nossas classes, pois, para possamos testar unitariamente uma classe, precisamos isolá-la, i. e., precisamos assumir que todas as

outras classes em nosso aplicativo estão funcionando corretamente e ver se a classe em teste está funcionando conforme o esperado.

Testes unitários fazem parte do processo de automatização de testes. Esta prática visa aperfeiçoar a qualidade do nosso código. Com o teste automatizado, escrevemos código para testar nosso próprio código. Isso ajuda a detectar falhas (ou, *bugs*) de forma ágil e eficiente, conforme seguimos modificando o nosso código.

Eis um exemplo de implementação e aplicação de interface:

```
using System;

namespace Extensibility
{
    public interface ILogger
    {
        void LogError(string message);
        void LogInfo(string message);
    }

    public class ConsoleLogger : ILogger
    {
        public void LogError(string message)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(message);
        }

        public void LogInfo(string message)
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine(message);
        }
    }
}
```

No código exposto acima, estamos programando para **interface**, i. e., a nossa classe *ConsoleLogger* fornece as implementações exigidas pela **interface ILogger**. A lógica do exemplo é fornecer uma funcionalidade que possa escrever, junto ao *console*, mensagens operacionais que atestem a normalidade, em verde, e erro, em vermelho, da execução de um determinado processo. A classe *ConsoleLogger* pode gerenciar o contexto de emissão de mensagem de qualquer tipo de processo. Ela pode ser utilizada para postar mensagens de um processo de conexão com arquivos, conexão com banco de dados, gerenciamento de migrações, dentre outros, bastando para tal que *ConsoleLogger* seja injetada em um método de uma classe. No exemplo abaixo, injetamos a relação de dependência entre *DbMigrator* e *ILogger*, i. e., *DbMigrator* precisa de uma implementação de *ILogger* para ser operacionalizada.

```
using System;

namespace Extensibility
{
    public class DbMigrator
    {
        private readonly ILogger _logger;

        public DbMigrator(ILogger logger)
        {
            _logger = logger;
        }

        public void Migrate()
        {
            _logger.LogInfo("Migrationg started at {0}" + DateTime.Now);

            // Details of migrating the database

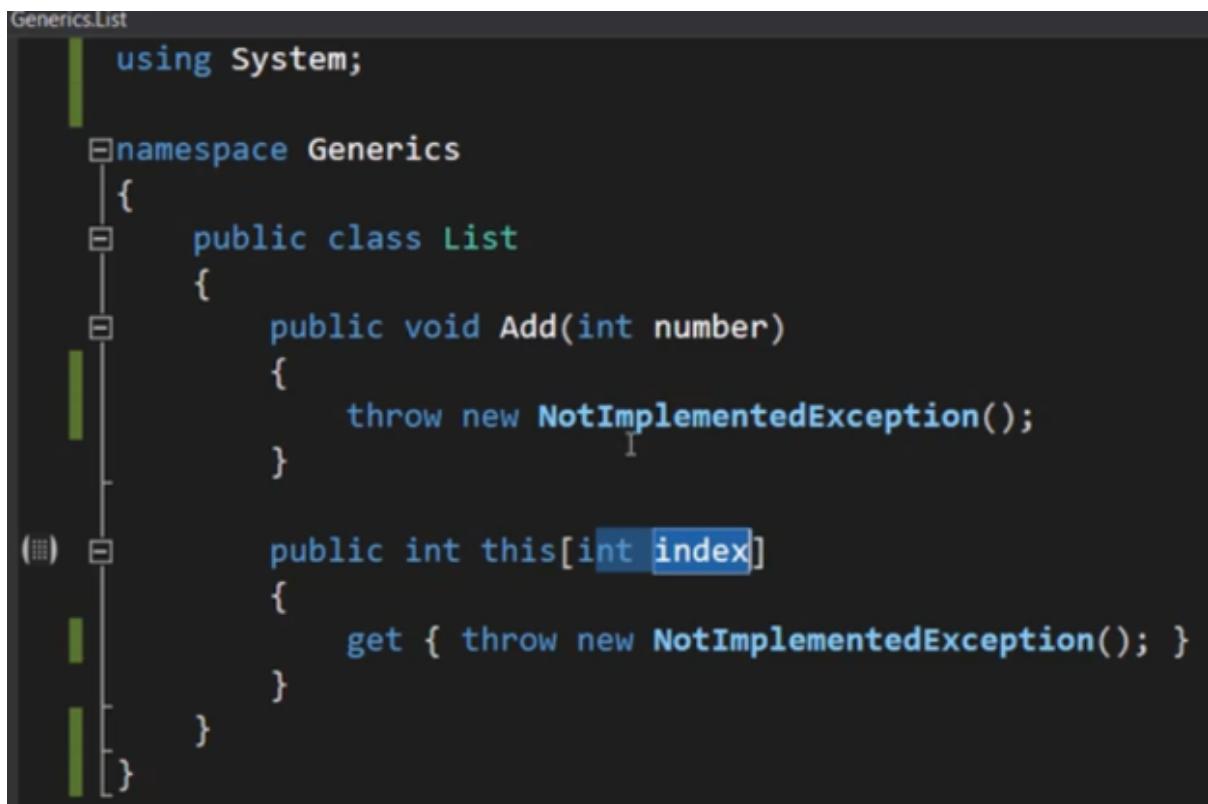
            _logger.LogInfo("Migrationg finished at {0}" + DateTime.Now);
        }
    }
}
```

```
namespace Extensibility
{
    class Program
    {
        static void Main(string[] args)
        {
            var dbMigrator = new DbMigrator(new ConsoleLogger());
            dbMigrator.Migrate();
        }
    }
}
```

## 2.13 Generics

O recurso **Generics** do C# foi projetado como uma solução para o problema de *boxing* e *unboxing*, tal qual problemas relacionados a conversão de tipos que resultam em penalidades para o desempenho da aplicação. **Generics** estimula a reutilização do código, elimina a duplicação de código e garante a segurança de tipagem no processamento de conjuntos de dados.

Para entender isso, digamos que queremos criar uma lista para armazenar uma lista de números. Abaixo está uma implementação simples de uma lista básica. Poderíamos ter um método *Add()* para adicionar um determinado número e um método *Indexer* para retornar o número em um determinado índice.

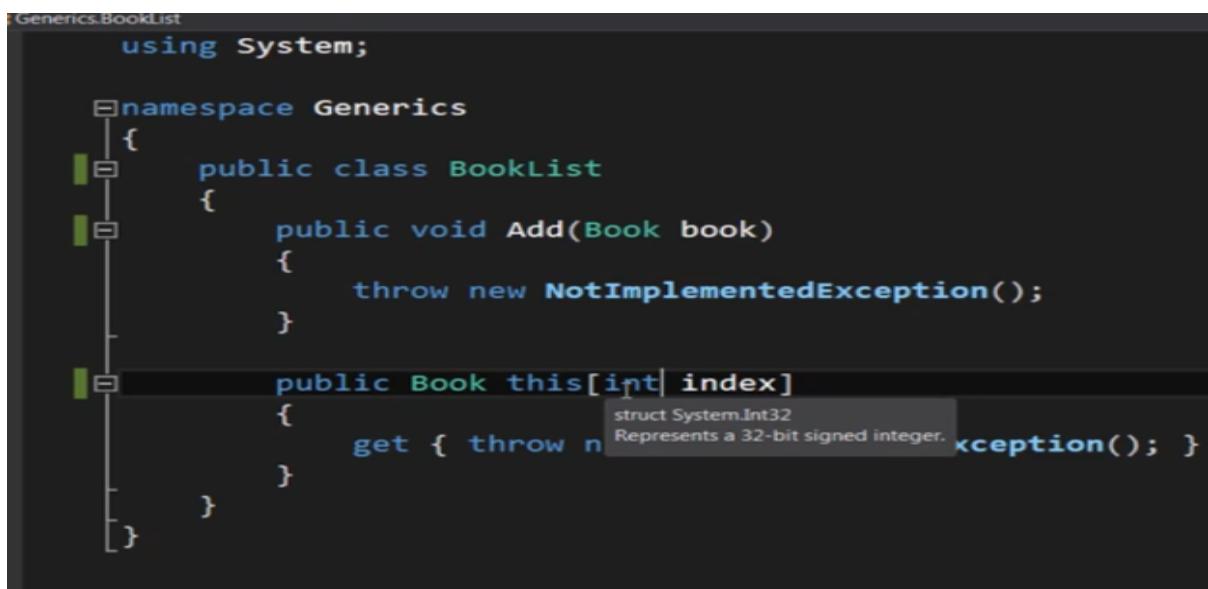


```
Generics.List
using System;

namespace Generics
{
    public class List
    {
        public void Add(int number)
        {
            throw new NotImplementedException();
        }

        public int this[int index]
        {
            get { throw new NotImplementedException(); }
        }
    }
}
```

Agora, se você quiser armazenar um livro (do inglês *book*) na lista, a classe acima não funcionará. Então teríamos que ir e criar outra classe e chamá-la de *BookList* e ela teria um método *Add()* que levaria armazenaria instâncias de *Book* como seu parâmetro e também um indexador do tipo *Book*.



```
Generics.BookList
using System;

namespace Generics
{
    public class BookList
    {
        public void Add(Book book)
        {
            throw new NotImplementedException();
        }

        public Book this[int index]
        {
            get { throw new NotImplementedException(); }
        }
    }
}
```

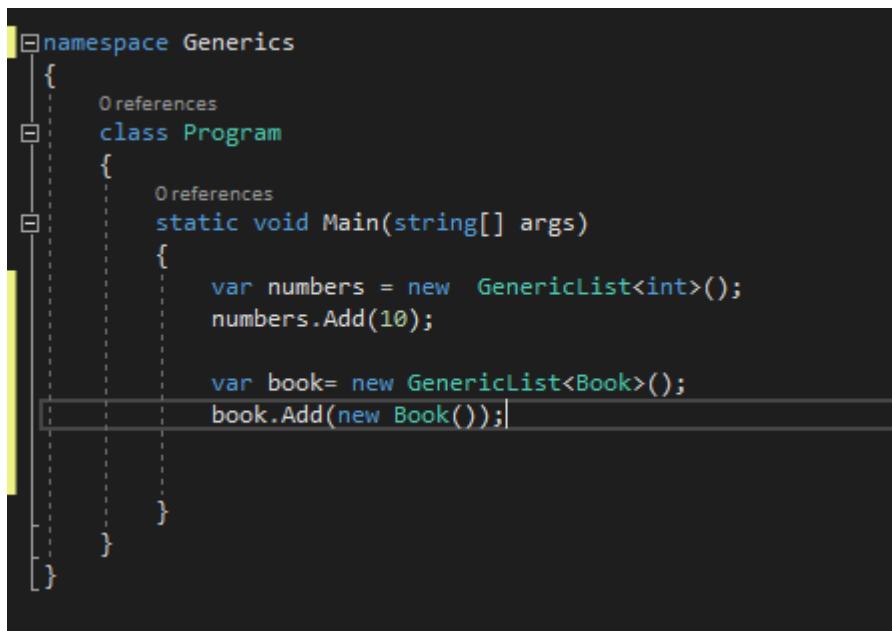
Agora, o problema aqui é que para cada tipo temos que criar uma lista separada e este sistema tem muita duplicação de código e não é produtivo. Mais ainda, se houver um *bug*, temos que corrigi-lo em vários lugares, portanto, uma solução para corrigir isso é usar uma lista de objetos como mostrado abaixo

```
public class GenericList<T>
{
    public void Add(T value)
    {

    }

    public T this[int index]
    {
        get { throw new NotImplementedException(); }
    }
}
```

Agora, nossa classe de lista tem um parâmetro e os parâmetros são especificados como <T>. Mantemos o *Add()* na classe de lista, mas ao invés de passar um objeto da classe *Book* ou qualquer outro parâmetro, passamos T. Desta forma, é fornecido a capacidade de parametrizar o tipo da lista no momento da instanciação.



```
namespace Generics
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new GenericList<int>();
            numbers.Add(10);

            var book= new GenericList<Book>();
            book.Add(new Book());
        }
    }
}
```

## 2.14 Expressões Lambda

Uma expressão lambda (ou, *lambda expression*) é apenas um método anônimo que não possui modificador de acesso (público ou privado), nenhum identificador e nenhuma instrução de retorno. Nós os usamos por conveniência, escrevendo menos código e obtendo o mesmo resultado. Além disso, torna o código mais legível.

Por exemplo, digamos que queremos escrever um método que pegue um número e retorne o quadrado do número. Normalmente, é assim que escreveremos, conforme mostrado abaixo:

```
using System;

namespace LambdaExpressions
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Square(5));
        }

        static int Square(int number)
        {
            return number*number;
        }
    }
}
```

Agora com a expressão lambda podemos conseguir a mesma coisa, mas com menos código. A sintaxe da expressão lambda é **args => expressão**, a qual interpretamos: ‘receba **args** e execute uma expressão’. Portanto, podemos escrever o método **square** acima com a expressão lambda conforme mostrado a seguir:

```
using System;

namespace LambdaExpressions
{
    class Program
    {
        static void Main(string[] args)
        {
            // args => expression
            //number => number*number;

            Func<int, int> square = number => number*number;

            Console.WriteLine(square(5));
        }
    }
}
```

Eis outra uso de expressão lambda:

```

namespace LambdaExpressions
{
    class Program
    {
        static void Main(string[] args)
        {
            var books = new BookRepository().GetBooks();

            var cheapBooks = books.FindAll(book => book.Price < 10);

            foreach (var book in cheapBooks)
            {
                Console.WriteLine(book.Title);
            }
        }
    }
}

```

Acima, definimos o argumento e a lógica de processamento da função *FindAll* com expressão lambda.

## 2.15 LINQ

LINQ significa *Language Integrated Query*. É um dos recursos que acompanha o C# 3.0. Ele oferece a capacidade de consultar objetos em C# nativamente. Com o LINQ, você pode consultar objetos na memória, por exemplo, coleções (LINQ to Objects), bancos de dados (LINQ to Entities), XML (LINQ to XML), conjuntos de dados ADO.NET (LINQ to Data Sets)

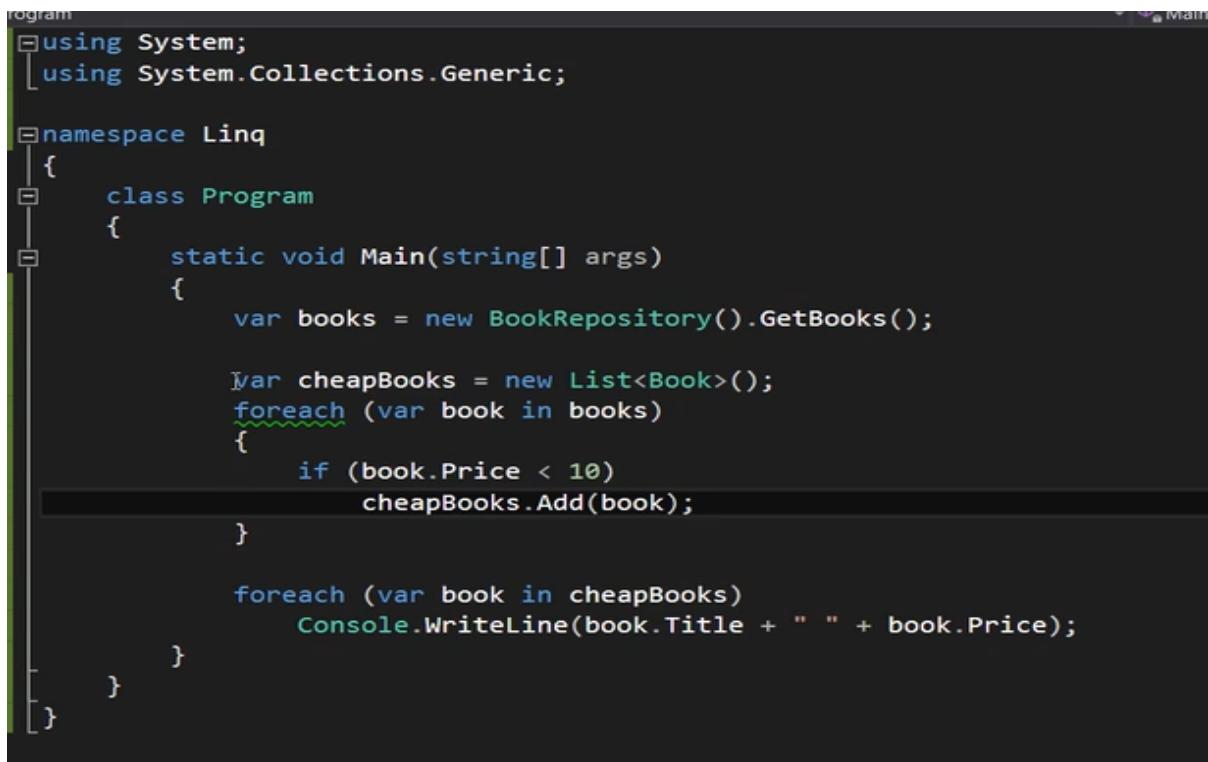
Para entender LINQ, vamos ver uma classe *BookRepository* a qual contempla duas propriedades: title; e, price. *BookRepository* possui em sua interface um método chamado *GetBooks* que retorna um objeto *IEnumerable* de objetos *Book*.

```

namespace Linq
{
    public class BookRepository
    {
        public IEnumerable<Book> GetBooks()
        {
            return new List<Book>
            {
                new Book() {Title = "ADO.Net Step by Step", Price = 5 },
                new Book() {Title = "ASP.NET MVC", Price = 9.99f },
                new Book() {Title = "ASP.NET Web API", Price = 12 },
                new Book() {Title = "C# Advanced Topics", Price = 7 },
                new Book() {Title = "C# Advanced Topics", Price = 9 }
            };
        }
    }
}

```

Em nosso arquivo **Program.cs**, simplesmente instanciamos o *BookRepository* e chamamos *GetBooks*, digamos que desejamos exibir os livros que custam mais do que 10 unidades monetárias, sem usar o LINQ, precisaremos iterar através da coleção de livros para obter aqueles que atendem à condição. Exemplo:



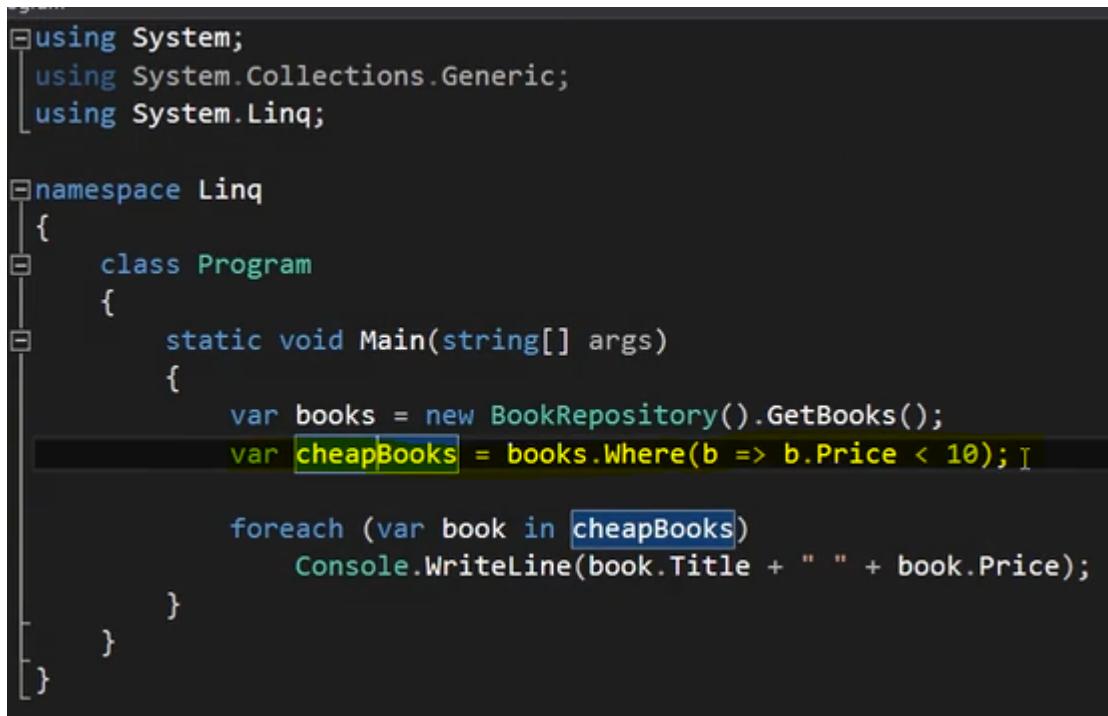
```
program
└── using System;
    └── using System.Collections.Generic;

└── namespace Linq
{
    class Program
    {
        static void Main(string[] args)
        {
            var books = new BookRepository().GetBooks();

            var cheapBooks = new List<Book>();
            foreach (var book in books)
            {
                if (book.Price < 10)
                    cheapBooks.Add(book);
            }

            foreach (var book in cheapBooks)
                Console.WriteLine(book.Title + " " + book.Price);
        }
    }
}
```

Eis como ficaria com LINQ:



```
program
└── using System;
    └── using System.Collections.Generic;
    └── using System.Linq;

└── namespace Linq
{
    class Program
    {
        static void Main(string[] args)
        {
            var books = new BookRepository().GetBooks();
            var cheapBooks = books.Where(b => b.Price < 10); I

            foreach (var book in cheapBooks)
                Console.WriteLine(book.Title + " " + book.Price);
        }
    }
}
```

Poderíamos ainda ordenar os objetos e selecionar apenas o atributo a ser retornado em cada objeto do arranjo:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Linq
{
    class Program
    {
        static void Main(string[] args)
        {
            var books = new BookRepository().GetBooks();
            var cheapBooks = books.Where(b => b.Price < 10).OrderBy(b => b.Title).Select(b => b.Title);

            foreach (var book in cheapBooks)
                Console.WriteLine(book);
                //Console.WriteLine(book.Title + " " + book.Price);
        }
    }
}
```

Outra outra forma de empregar o LINQ para processar uma consulta junto a um conjunto de objetos seria:

```
// LINQ Extension Methods
var cheapBooks = books
    .Where(b => b.Price < 10)
    .OrderBy(b => b.Title)
    .Select(b => b.Title);
```

Ou ainda com o uso dos *Query Operators* do LINQ:

```
// LINQ Query Operators
var cheaperBooks =
    from b in books
    where b.Price < 10
    orderby b.Title
    select b.Title;
```

Digamos que eu queira obter um livro com um título específico, posso usar o método *Single()* para conseguir isso:

```
class Program
{
    static void Main(string[] args)
    {
        var books = new BookRepository().GetBooks();
        var book = books.Single(b => b.Title == "ASP.NET MVC");
        Console.WriteLine(book.Title);
    }
}
```

E quando não temos certeza se esse objeto existe, podemos usar o método `SingleOrDefault()` para evitar que nosso aplicativo falhe porque o nulo padrão será retornado:

```
class Program
{
    static void Main(string[] args)
    {
        var books = new BookRepository().GetBooks();

        // LINQ Extension Methods
        var book = books.SingleOrDefault(b => b.Title == "ASP.NET MVC++");

        Console.WriteLine(book.Title);
    }
}
```

Usamos `First()` para obter o primeiro elemento da lista. E você pode usar o `FirstOrDefault()` quando não tiver certeza se tal elemento existe

```
class Program
{
    static void Main(string[] args)
    {
        var books = new BookRepository().GetBooks();

        // LINQ Extension Methods
        var book = books.First(b => b.Title == "C# Advanced Topics");

        Console.WriteLine(book.Title + " " + book.Price);
    }
}
```

Também temos `Skip()` e `Take()`. Aqui em nosso exemplo, podemos decidir pular os dois primeiros livros e devolver os três últimos:

```
class Program
{
    static void Main(string[] args)
    {
        var books = new BookRepository().GetBooks();

        // LINQ Extension Methods
        var pagedBooks = books.Skip(2).Take(3);

        foreach (var pagedBook in pagedBooks)
        {
            Console.WriteLine(pagedBook.Title);
        }
    }
}
```

Também temos várias funções de agregação no LINQ, assim como no SQL, como *Count()*, *Max()*, *Sum()*, *Min()* etc.

```
// LINQ Extension Methods
var maxPrice = books.Max(b => b.Price);
var minPrice = books.Min(b => b.Price);

Console.WriteLine(maxPrice);
Console.WriteLine(minPrice);

var count = books.Count();

Console.WriteLine(count);
```

## 2.16 Tipos anuláveis

Existem algumas situações em que você precisará tratar valores nulos. Digamos que você esteja trabalhando com o banco de dados e tenha uma tabela chamada clientes e uma coluna que abstraia data de nascimento, sendo esta coluna do tipo *DateTime* e podendo ser nula, já que este dado não é obrigatório. Nesse caso, se você quiser mapear essa tabela para uma classe C#, precisará de um tipo anulável (do inglês *nullable*) para fazer isso.

Qualquer tipo de dados pode ser declarado anulável usando o operador '?'. Para fins de exemplificação, considere que temos um objeto identificado com **date** e do tipo *DateTime* anulável atribuído a um outro objeto identificado como **date2** também do tipo *DateTime*. Ao atribuir **date** para **date2**, chamamos um método *GetValueOrDefault()* na Biblioteca *DateTime* para que, se **date** anulável for nulo, um valor padrão será atribuído ao **date2** sem nosso aplicativo lançar uma exceção. Eis o código desta situação:

```
using System;

namespace NullableTypes
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime? date = new DateTime(2014, 1, 1);
            DateTime date2 = date.GetValueOrDefault();

            Console.WriteLine(date2);

        }
    }
}
```

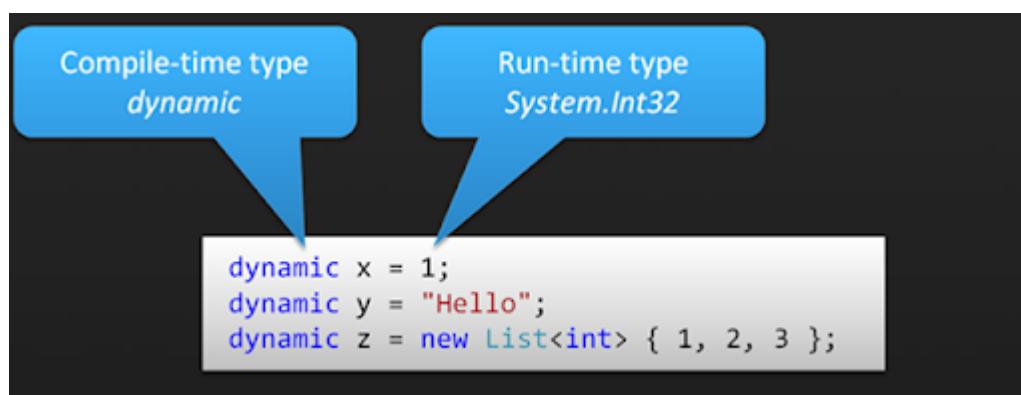
C# fornece também o operador de ‘coalescência nulo’ que é representado usando ‘??’. Eis um exemplo de seu uso:

```
DateTime date2 = date ?? DateTime.Today; I
```

O que o código acima significa é que se **date** não for nulo, atribua **date** a **date2**, se não, atribua a **date2** o valor de *DateTime.Today*. Tipos anuláveis não devem ser utilizados em variáveis ou atributos de tipos primitivos.

## 2.17 Dynamics

**Dynamics** é um dos recursos fornecidos com o C# 4.0 que às vezes confunde alguns dos desenvolvedores do C#. No entanto, é realmente fácil. Normalmente, as linguagens de programação são divididas em linguagens tipadas estaticamente, ou fortemente tipadas, como Java, e linguagens tipadas dinamicamente como Ruby, JavaScript e Python. Agora, qual é realmente a diferença entre esses dois? A diferença é que em linguagens estáticas a resolução de tipos, membros, métodos e propriedades é feita em tempo de compilação. Já nas linguagens que suportam tipagem dinâmica (do inglês *dynamics*) a resolução de tipo, métodos, propriedade ou membros é feita em tempo de execução.



## 2.18 Manipulação de exceção

Em C#, usamos o tratamento de exceções para lidar com situações inesperadas ou excepcionais que surgem do programa que escrevemos. A sintaxe dos comandos é a mesma apresentada no Java, portanto, se desdobra na implementação das instruções:

- **try**: num bloco try, há um trecho de código cuja execução pode resultar em um erro A, B, C, e na ocorrência de um erro, o mesmo pula para o bloco catch;
- **catch**: num bloco catch, é onde deve constar o código para tratar a exceção;
- **finally**: indica o que deve ser feito após o término do bloco try ou de um catch qualquer;
- **throw**: lança (*throw* em inglês) uma exceção;

- **throws**: Utilizado na assinatura de um método B para delegar o tratamento de uma exceção X o tratamento para um método A que invoca o método B, que por sua vez é quem deve estar tratando X.

Eis um exemplo de uso das instruções **try-catch-finally**:

```

using System;
using System.IO;

namespace ExceptionHandling
{
    class Program
    {
        static void Main(string[] args)
        {
            StreamReader streamReader = null;
            try
            {
                streamReader = new StreamReader(@"c:\file.zip");
                var content = streamReader.ReadToEnd();
                throw new Exception("Oops");
            }
            catch (Exception ex)
            {
                Console.WriteLine("Sorry, an unexpected error occurred.");
            }
            finally
            {
                if (streamReader != null)
                    streamReader.Dispose();
            }
        }
    }
}

```

## 2.19 Programação Assíncrona

*Asynchronous JavaScript and XML* (Ajax) é o uso metodológico de tecnologias como Javascript e XML, suportada por navegadores, para tornar páginas Web mais interativas com o usuário, utilizando-se de solicitações assíncronas de informações. Antes da disponibilização do Ajax, era quase impossível alterar uma pequena parte do conteúdo de uma página web sem recarregar a página inteira. O cenário decaia com uma conexão lenta com a internet.

Com Ajax, podemos fazer qualquer alteração sem recarregar uma página inteira ou sem tocar em outro elemento que integre a página web. Ajax pode ser compreendido como uma técnica assíncrona para troca de dados entre o servidor e o cliente. Portanto, o objetivo principal do Ajax é ativar um método de servidor e trocar dados do servidor sem obstruir qualquer atividade que esteja sendo realizada pelo cliente.

Por exemplo, quando você acessa seu e-mail e clica na caixa de spam, apenas a porção da página web é atualizada para elencar os e-mails recebidos e avaliados como spam. Ao

clicar na caixa spam e disparar a funcionalidade para renderizar a caixa de spam também não inviabiliza a funcionalidade observar chegar um novo e-mail na caixa principal. Ao clicar na caixa de spam, mesmo que a renderização da caixa de spam seja demorada, o processo de aviso de recebimento de novas mensagens não cessa. Isso só é possível por causa das chamadas assíncronas do Ajax.

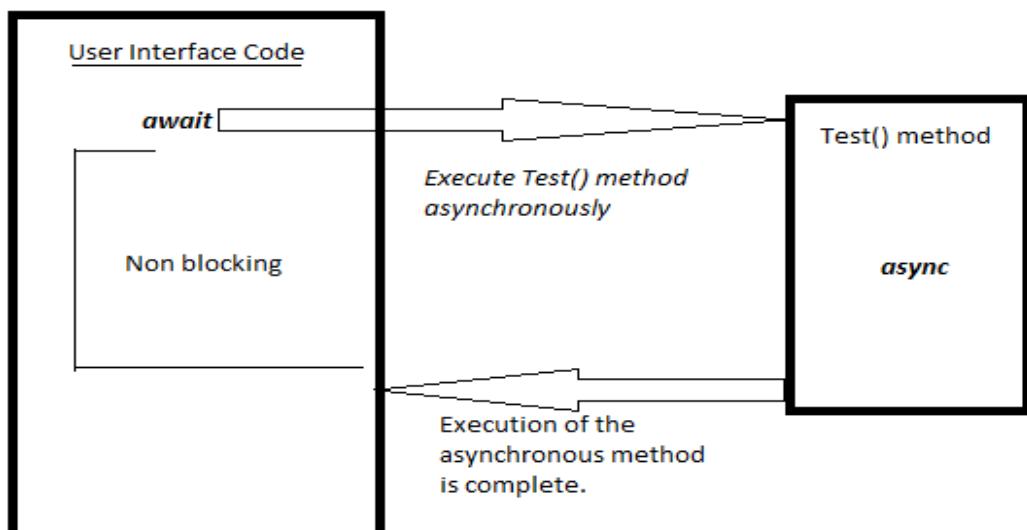
Para que o servidor possa suportar as chamadas assíncronas com Ajax de uma página web, a aplicação que responde pela camada do servidor deve ser programada em linguagem que suporte programação assíncrona. No C# 5.0, a Microsoft introduziu a programação assíncrona para fornecer aos desenvolvedores a oportunidade de escrever métodos assíncronos.

No modelo de execução síncrona, o programa é executado linha por linha, um de cada vez. Quando uma função é chamada, a execução do programa deve esperar até que a função retorne. Já no modelo assíncrono, quando um método é chamado, a execução do programa continua para a próxima linha, sem esperar que a função seja concluída.

Na plataforma .NET, versão 4.5 em diante, a Microsoft introduziu um novo modelo de programação assíncrona, que foi chamado de modelo assíncrono à base de tarefas (ou **Tasks**). Existem duas palavras-chave principais usadas para utilizar estas funcionalidades:

- **async**: esta palavra-chave qualifica uma função como uma função assíncrona. Portanto, se uma função tiver `async` especificado em sua frente, essa função será assíncrona.
- **await**: ao invocar um método que retorna uma `Task <string>`, precisamos colocar a palavra-chave `await` na frente do método.

Eis a arquitetura do modelo de programação assíncrona em C#:



#### Asynchrony in .NET Framework

O comando **await** induz a pensar no estado de espera, i. e., que um método A que invoca outro método B e utiliza **await** para invocar B deve esperar o término da execução de B para proceder com a execução das próximas instruções. Porém, a semântica do **await** é apenas dizer ao compilador que essa operação é cara e pode demorar um pouco para ser executada. Nesse caso, em vez de bloquear a *thread* corrente, o Sistema Operacional simplesmente retornará o controle imediatamente para o chamador do método e uma *thread* aparte será criada para executar o método assíncrono invocado com **await**. Eis um exemplo de código de chamadas a métodos assíncronos com **await**:

```
public MainWindow()
{
    InitializeComponent();
}

private void Button_Click(object sender, RoutedEventArgs e)
{
    DownloadHtmlAsync("http://msdn.microsoft.com");
}

public async Task DownloadHtmlAsync(string url)
{
    var webClient = new WebClient();
    var html = await webClient.DownloadStringTaskAsync(url);

    using (var streamWriter = new StreamWriter(@"c:\projects\result.html"))
    {
        await streamWriter.WriteLine(html);
    }
}
```

No exemplo acima, temos um método que suporta um evento disparado quando o usuário clica em um botão para obter o *download* de uma página HTML. Estamos usando o método assíncrono para garantir que, durante o *download*, a janela não congele e possa ser movida de um lugar para outro na tela.

Como marcamos **DownloadStringTaskAsync(url)** com o operador **await**, o método retornará a execução da *thread* principal imediatamente ao evento **Button\_Click()** que chamou o método **DownloadHtmlAsync**. Eventualmente, quando a *thread* criada pela invocação de **DownloadStringTaskAsync(url)** terminar de executar no escopo de **DownloadHtmlAsync**, a execução continuará em assíncrono **DownloadhtmlAsync()**, resultando na execução do **StreamWriter** para gerar o arquivo **result.html**. Nas libs da plataforma .NET, para cada método síncrono, há um método assíncrono correspondente.

### 3. Conceitos de SOLID

Os princípios SOLID, propostos por Robert C. Martin (Uncle Bob) em 2003, visam definir cinco princípios que formam as iniciais do termo citado. Combinando as práticas recomendadas por estes princípios, a tendência é que se construam sistemas sobre o paradigma orientado a objetos mais fáceis de evoluir, manter, testar e reutilizar. A seguir, é definido cada conceito e apresentados exemplos de códigos. No fim desta seção, é apresentado o conceito de **Code Bad Smells**.

#### 3.1 Single Responsibility Principle (SRP)

Este conceito representa o S do acrônimo SOLID. De acordo com o **Single Responsibility Principle**, classes precisam ser coesas, possuindo uma única responsabilidade. Isso irá gerar um equilíbrio de métodos e atributos para cada classe, não permitindo que cresçam de forma desproporcional, evitando adquirir funcionalidades que não estejam de acordo com o propósito da classe.

Conforme o SRP, você deve evitar as **God Class**, i. e., classes com muitas responsabilidades. O ideal é uma classe ter apenas uma responsabilidade. Em outras palavras, uma classe deve ter apenas um único motivo para mudar. O exemplo é imaginar um sistema que conta com o processamento de **ordens** de pagamento de uma loja. Uma ordem possui um arranjo de itens. Um exemplo de classe para abstrair a entidade **ordem** seria:

```
0 references
class Ordem
{
    0 references
    public void CalcularTotal() { }

    0 references
    public void GetItens() { }

    0 references
    public void GetItensCount() { }

    0 references
    public void AddItem() { }

    0 references
    public void DeleteItem() { }

    0 references
    public void PrintOrder() { }

    0 references
    public void ShowOrder() { }

    0 references
    public void Load() { }

    0 references
    public void Save() { }

    0 references
    public void Update() { }

    0 references
    public void Delete() { }
}
```

Como podemos ver na Figura posta acima, a classe **Ordem** possui três blocos de métodos. Cada bloco corresponde a uma responsabilidade distinta: o primeiro bloco abstrai a

responsabilidade sobre o controle do arranjo de itens; o segundo bloco abstrai a responsabilidade sobre a exibição dos itens de uma instância de **Ordem**; e, o terceiro bloco abstraí a responsabilidade por contemplar a funcionalidade de gerenciamento de uma instância de **Ordem** em um banco de dados.

Temos então que a classe **Ordem** fere o SRP, ou o S do SOLID, pois ela possui mais de uma responsabilidade. A forma correta de implementar a entidade **Ordem** seria então criar uma classe para cada responsabilidade inerente a esta entidade. A implementação se daria da seguinte forma:

```
0 references
class Ordem {
}

0 references
public void CalcularTotal() { }

0 references
public void GetItens() { }

0 references
public void GetItensCount() { }

0 references
public void AddItem() { }

0 references
public void DeleteItem() { }
}

0 references
class OrdemViewer {
    0 references
    public void PrintOrder() { }

    0 references
    public void ShowOrder() { }
}

0 references
class OrdemRepository {
    0 references
    public void Load() { }

    0 references
    public void Save() { }

    0 references
    public void Update() { }

    0 references
    public void Delete() { }
}
```

Assim, teríamos uma classe responsável por cada responsabilidade inerente à entidade **Ordem**. Isso satisfaz também outra boa prática de Projeto de Software: *cada classe deve ter apenas um motivo para mudar*.

## 3.2 Open/Closed Principle (OCP)

Este princípio diz que classes precisam ser abertas para extensão e fechadas para modificação, ou seja, não podem ser modificadas com frequência. Se recursos precisam ser adicionados, devemos estender e não incrementar, mantendo intacto o código fonte original. Ao violar este princípio, classes podem se tornar complexas ao longo das suas modificações, por acumularem diversas estruturas condicionais e caminhos distintos para executar novas funcionalidades.

A consequência é que tais classes estarão propensas a sofrerem constantes manutenções. Para evitar que isso ocorra, é necessário construir abstrações, que podem ser

representadas utilizando herança ou implementação de interfaces. Para melhor compreender este conceito, considere o código que se segue:

```
3 references
class ContratoCLT
{
    1 reference
    public float Salario() { return 1000; }

}

3 references
class Estagio
{
    1 reference
    public float Bolsa() { return 500; }
}

0 references
class FolhaPagamento
{
    protected float Saldo = 5000;

    0 references
    public void Calcular(Object input)
    {

        if (typeof(ContratoCLT).IsInstanceOfType(input))
        {
            ContratoCLT contrato = (ContratoCLT)input;
            this.Saldo -= contrato.Salario();
        }
        else if (typeof(Estagio).IsInstanceOfType(input))
        {
            Estagio estagio = (Estagio)input;
            this.Saldo -= estagio.Bolsa();
        }
    }
}
```

Este exemplo atende a uma funcionalidade de calcular o pagamento da folha de uma empresa. Inicialmente, foi levantado que a empresa teria duas modalidades de contratação: contrato CLT; e, bolsa de estágio. No decorrer do desenvolvimento, foi identificado que a empresa também poderia contratar via PJ. Segundo o viés de codificação observado no exemplo, mais uma cláusula condicional teria de ser criada. Essa tendência se repetiria para cada novo tipo de contrato identificado, levando a classe FolhaPagamento a ter de ser modificada a cada novo requisito identificado quanto às formas de contrato. Além disso, ao incrementar uma classe que já funcionava e havia sido validada, estaríamos correndo o risco de criar bugs em um recurso da aplicação que já funcionava. Portanto, seguindo o princípio OCP, o código supracitado seria este:

```

4 references
interface IRemuneravel
{
    4 references
    float Remuneracao();
}

0 references
class ContratoCLT : IRemuneravel
{
    4 references
    public float Remuneracao() { return 1000; }

}

0 references
class Estagio : IRemuneravel
{
    4 references
    public float Remuneracao() { return 500; }

}

0 references
class ContratoPJ : IRemuneravel
{
    4 references
    public float Remuneracao() { return 1200; }
}

```

```

0 references
class FolhaPagamento
{
    protected float Saldo = 5000;

    0 references
    public void calcular(IRemuneravel remuneravel)
    {
        this.Saldo -= remuneravel.Remuneracao();
    }
}

```

A ideia da codificação seguindo o OCP é o de separar o comportamento extensível em uma **Interface** (ou seja, uma abstração) e inverter as dependências. No exemplo geral, o comportamento extensível é: a cada nova modalidade de contrato que se apresenta, devemos codificar novas funcionalidades. Segundo a codificação seguindo OCP, não depende mais da classe **FolhaPagamento** fornecer a lógica para o processamento do cálculo do saldo. A lógica agora é fornecida por cada classe que decidiu ser **IRemuneravel**. Ou seja, invertemos a dependência referente a suportar uma funcionalidade. Neste contexto, fica claro que com abstrações bem definidas, fica mais seguro expandir o código para criar novas funcionalidades.

### 3.3 Liskov Substitutive Principle (LSP)

Este princípio prevê que as subclasses de herança precisam ser construídas atendendo critérios, para evitar conflitos com suas superclasses. Um critério consiste nos métodos das subclasses serem obrigados a estabelecer pré-condições de igual ou menor rigidez que os métodos das suas superclasses. E as pós-condições com igual ou maior rigidez. As

subclasses também não podem lançar exceções que não são lançadas pelas superclasses. Estes critérios foram construídos para que superclasses possam representar o comportamento da herança diante das classes externas, sem que suas subclasses produzam efeitos colaterais inesperados.

O LSP foi definido pela cientista da computação Barbara Liskov em 1987. Em termos mais simples, este princípio prega que uma classe derivada deve ser substituível pela sua classe base sem que seja necessário alterar as propriedades do software. É basicamente a implementação de métodos polimórficos observando que a assinatura deste método deve ser respeitada e o comportamento esperado neste método deve ser entregue. Vejamos o código que se segue:

```
2 references
class A
{
    2 references
    public virtual void Operacao()
    {
        Console.WriteLine("Imprimindo mensagem A");
    }
}

0 references
class B : A
{
    2 references
    public override void Operacao()
    {
        Console.WriteLine("Imprimindo mensagem B");
    }
}

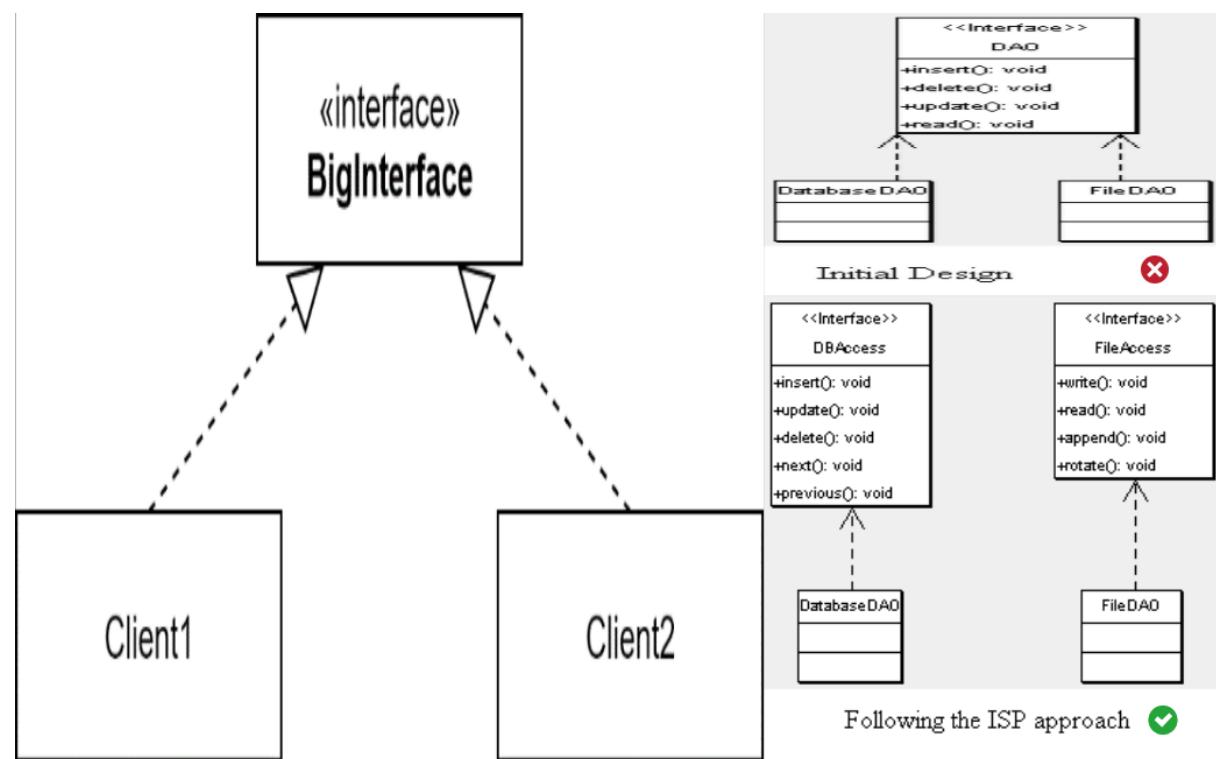
0 references
class Processador
{
    0 references
    public void Processar(A input)
    {
        input.Operacao();
    }
}
```

Neste exemplo, temos que **B** estende **A** e modifica o comportamento herdado. Na classe **Processador**, podemos passar tanto uma instância de **A** quanto de **B** que o programa continua a funcionar da forma esperada. Neste exemplo, ferir o LSP seria fazer a sobreescrita do método **Operacao** em **B** e não implementar a lógica da operação nesta sobreescrita, que é imprimir uma mensagem. Outra forma de ferir o LSP neste exemplo seria modificar a assinatura do método **Operacao** em **B** que foi definida em **A**. Esse princípio assegura o bom uso do Poli

### 3.4 Interface Segregation Principle (ISP)

Este princípio defende que interfaces sejam coesas, e que as assinaturas de seus métodos representem um único comportamento. A intenção é que as interfaces sejam estáveis, evitando que subclasses implementem algum método de forma indesejada.

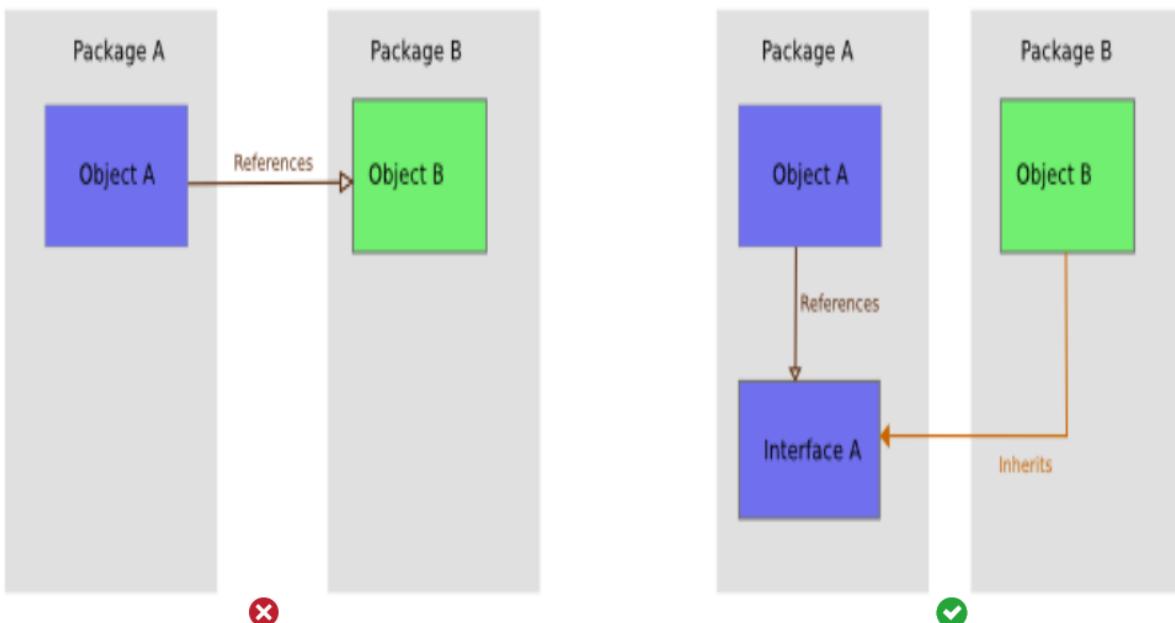
Em termos simples, o ISP prega que não devemos ter **interfaces** genéricas, i. e., não se pode ter uma **interface** cuja uma classe não tenha que implementar todos os seus métodos. A ideia é a de segregar **interfaces**, ou seja, isolar comportamentos e responsabilidades para que não se observe no código uma classe **A** que implemente uma **interface Y** sem que **A** forneça implementações concretas para todos os métodos de **Y**. A Figura que se segue ilustra como seria o projeto funcional com e sem a aplicação do ISP.



### 3.5 Dependency Inversion Principle (DIP)

É inevitável que classes possuam dependências entre elas. Contudo, o acoplamento precisa ser o menor possível entre as classes envolvidas. O baixo acoplamento fará com que modificações em uma classe não sejam propagadas para suas dependências, facilitando a manutenção. Portanto, é preferencial que uma classe possua acoplamento apenas com interfaces ou superclasses, utilizando suas subclasses polimorficamente.

Em outras palavras, o DIP prega que nossas classes devem depender de abstrações e não de implementações concretas, i. e., devemos separar o comportamento extensível por trás de uma interface e inverter as dependências (conceito este explicado na seção anterior). Com isso, teremos que módulos de alto nível não dependerão de módulos de baixo nível, e sim que ambos dependerão de abstrações. A Figura que se segue ilustra a ideia.



Se ainda não conseguiu entender este princípio, observe os exemplos de código apresentados na seção 3.2. Vemos que no exemplo 1 (sem aplicação do OCP) exposto na seção 3.2, temos duas classes que tratam de um contexto específico, **ContratoCLT** e **Estagio**. Cada uma trata de um tipo de remuneração. A terceira classe, **FolhaPagamento**, tem um propósito mais geral que as outras duas: ela calcula a folha de pagamento com base nos tipos de remuneração dispostos na aplicação. Neste contexto, podemos interpretar também que **FolhaPagamento** possui um nível de complexidade a mais que as outras classes que abstraem os tipos de remuneração. Vemos que para que o método de cálculo de **FolhaPagamento** seja operacionalizado é necessário referenciar outras classes. Temos aqui um exemplo de acoplamento (ou dependência), e por **FolhaPagamento** referenciar classes concretas, temos um forte acoplamento, pois modificações em classes concretas devem, naturalmente, serem propagadas pelas classes que as referenciam, i. e., toda vez que surgir uma nova modalidade de forma pagamento, precisaremos modificar **FolhaPagamento**. Ainda neste contexto, podemos interpretar que um componente de maior nível de complexidade, **FolhaPagamento**, depende de componentes de baixo nível, **Estagio** e **ContratoCLT**. Uma última observação sobre **FolhaPagamento** é que esta classe pode ser interpretada como uma **entidade volátil**. Quando não conseguimos ter uma noção sobre o futuro de uma classe, podemos considerá-la como uma **entidade volátil**. Entidades voláteis são, geralmente, módulos compostos por rotinas ou algoritmos que estão em processo de desenvolvimento e que, por sua natureza, sofrem mudanças frequentes, e se uma classe for depender de outra, ela deve depender sempre de outro módulo mais estável do que ela mesma.

É nesse tipo de contexto que o DIP entra em ação. Observe que no exemplo 2 (aplicação do OCP) da seção 3.2 invertemos as dependências. **FolhaPagamento** está associada a uma abstração, **IRemuneravel**, que dificilmente será modificada e agora as classes de baixo nível é que passam a depender da abstração para serem operacionalizadas.

“Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.” (ROBERT 'UNCLE BOB' MARTIN, 1996)

“Os sistemas mais flexíveis são aqueles em que as dependências de código-fonte se referem apenas a abstrações e não a itens concretos.” (MARTIN 2019)



### 3.6 Code Bad Smells

Para projetar classes eficientes, sistemas que adotam o paradigma POO deveriam seguir os princípios SOLID citados. Contudo, os desenvolvedores podem violar alguns desses princípios por vários motivos, seja pela urgência na entrega do sistema, adoção de uma solução deficiente no projeto das classes do sistema, ou simplesmente pelo desconhecimento quanto aos princípios citados. Estas violações resultam no surgimento de algumas anomalias nas classes desses sistemas. Várias dessas anomalias foram catalogadas e apelidadas de Code Bad Smells (ou Code Smells).

Em linhas gerais, a presença de anomalias nas classes de um sistema pode apresentar danos como o decremento da comprehensibilidade e manutenibilidade. Além disso, tais sistemas podem se tornar mais propensos a falhas, e sofrer mais mudanças no decorrer do tempo. Segue a lista das Code Smells mais conhecidas:

- a) **Class Data Should be Private** - Classes que expõe seus atributos, permitindo que o comportamento da classe seja manipulado externamente, quebrando o encapsulamento da classe. Este Code Smell afeta diretamente o princípio D do SOLID.

- b) **Complex Class** - Classes que são difíceis de testar e manter, por possuírem diversas estruturas condicionais e caminhos de execução distintos. Este Code Smell afeta diretamente o princípio O do SOLID.
- c) **Functional Decomposition** - Classes que declaram muitos atributos e implementam poucos métodos, fazendo pouco uso dos recursos de POO. Este Code Smell afeta diretamente o princípio O do SOLID.
- d) **God Class** - Classes que controlam muitos outros objetos do sistema. Geralmente possuem baixa coesão, pois acumulam diversas responsabilidades, e possuem diversas linhas de código-fonte. Este Code Smell afeta diretamente os princípios S e O do SOLID.
- e) **Lazy Class** - Classes que possuem poucas funcionalidades, não justificando a sua existência. Este Code Smell não afeta diretamente nenhum princípio SOLID, já que a classe infectada sequer deveria existir.
- f) **Long Method** - Métodos que são desnecessariamente longos. Portanto, poderiam ser divididos, gerando outros métodos. Este Code Smell afeta diretamente o princípio O do SOLID.
- g) **Refused Parent Bequest** - Ocorre quando uma subclasse herda comportamentos indesejados das suas superclasses, ou necessitam implementar métodos também indesejados de interfaces ou classes abstratas. Isto ocorre pela falta de compatibilidade entre superclasse e subclasses, gerando uma série de restrições para determinadas subclasses, e consequentemente, obrigando o desenvolvedor a conhecer sua arquitetura interna para utilizá-la. Este Code Smell afeta diretamente os princípios ISP e LSP;
- h) **Spaghetti Code** - Classes que não representam nenhum tipo de comportamento para o sistema. Costumam possuir apenas alguns métodos longos sem parâmetros. Este Code Smell afeta diretamente os princípios S e O do SOLID.

## 4. Programando uma API REST em C#

Uma API pode ser compreendida como um conjunto de rotinas e padrões estabelecidos e documentados em uma aplicação **W**, para que outras aplicações consigam utilizar as funcionalidades desta aplicação **W**, sem precisar conhecer detalhes da implementação de **W**. Desta forma, APIs permitem interoperabilidade entre aplicações.

Interoperabilidade é a capacidade de um sistema (informatizado ou não) de se comunicar de forma transparente (ou o mais próximo disso) com outro sistema (semelhante ou não). Para um sistema ser considerado interoperável, é muito importante que ele trabalhe com padrões abertos ou por uma ontologia (modelo de dados que representa um conjunto de conceitos dentro de um domínio e os relacionamentos entre estes).

Este fator reforça a importância de seguir padrões na hora de codificar uma API. Um dos pontos a se observar durante a codificação é que os dados fornecidos pela API devem ser de fácil representação e compreensão tanto para humanos quanto para máquinas. Para fins

de exemplificação, considere estes formatos de serialização (ou codificação estrutural) de dados:

XML	JSON	YAML
<pre>&lt;Servers&gt;   &lt;Server&gt;     &lt;name&gt;Server1&lt;/name&gt;     &lt;owner&gt;John&lt;/owner&gt;     &lt;created&gt;123456&lt;/created&gt;     &lt;status&gt;active&lt;/status&gt;   &lt;/Server&gt; &lt;/Servers&gt;</pre>	<pre>{   Servers: [     {       name: "Server1",       owner: "John",       created: "123456",       status: "active"     }   ] }</pre>	<pre>Servers:   - name: Server1     owner: John     created: 123456     status: active</pre>

As três representações são válidas e seu valor semântico é o mesmo para a máquina. O que muda é a legibilidade e facilidade de interpretação de cada modelo para o programador. O formato XML é o mais extenso, exigindo um esforço extra por parte de quem está codificando. Já o formato *JavaScript Object Notation* (JSON) é algo mais simplificado de se escrever. Por fim, o formato *YAML Ain't Markup Language* (YAML) é praticamente como escrevemos na nossa linguagem natural.

O *Hypertext Transfer Protocol* (HTTP) é o protocolo recorrentemente utilizado para garantir a interoperabilidade entre uma API e demais aplicações ou clientes. A fim de aperfeiçoar este protocolo, foi proposto um conjunto de princípios conhecidos como REST. O REST foi introduzido em 2000 na tese de doutoramento de Roy Fielding, um dos principais autores das especificações do protocolo HTTP. Roy Fielding, descreve o REST como sendo um estilo de arquitetura de software que define um conjunto de princípios e restrições a serem usados para a criação de serviços web. Os serviços web que estão em conformidade com o estilo arquitetural REST, são chamados serviços Web RESTful. É importante realçar que o REST não é um protocolo mas sim um estilo de arquitetura. A seguir serão listados os princípios fundamentais e restrições de REST:

- **Client-Server:** cada mensagem HTTP contém toda a informação necessária para compreender o pedido. Isso ajuda a estabelecer uma arquitetura distribuída, dando suporte à evolução independente da lógica do lado do cliente e do servidor.
- **Stateless:** cada pedido do cliente ao servidor deve conter todas as informações necessárias para perceber o pedido, e não pode tirar proveito de nenhum contexto armazenado no servidor. Por outras palavras, cada pedido ao servidor deve ser entendido como a primeira e não deve ter relação com o anterior. Desta forma é muito mais fácil escalar o sistema.
- **Cache:** as restrições de *cache* exigem que a resposta de um pedido sejam implicitamente ou explicitamente rotulados como armazenáveis em *cache* ou não. Se uma resposta for armazenável em *cache*, a *cache* do cliente poderá reutilizar os dados da resposta para pedidos posteriores equivalentes.

- **Desenvolvimento em camadas:** a arquitetura REST permite que o sistema seja desenvolvido em camadas, por exemplo, um sistema pode implementar APIs no servidor A, armazenar os dados no servidor B e autenticar pedidos no servidor C. Um cliente normalmente não precisa conhecer se está conectado diretamente ao servidor final ou a um intermediário ao longo do caminho
- **Uniform Interface:** o princípio que caracteriza um sistema REST em particular é ter uma interface uniforme (*Uniform Interface*), que sustenta que os métodos de interação entre os componentes do sistema devem ser regulados por convenções uniformes. De fato, Fielding indica quatro restrições que devem ser atendidas para satisfazer esse princípio:
  - **Identificação de Recursos:** um recurso é um objeto ou a representação de algo significativo no domínio do sistema. A interação com estes recursos é feita através das APIs. Exemplos de entidades: um carro, um equipamento, um utilizador e qualquer outra entidade que possa ser abstraída de um determinado contexto. O conceito de recurso é, portanto, muito semelhante ao de um objecto no mundo da programação orientada a objectos. A identificação do recurso deve ser feita utilizando-se o conceito de *Uniform Resource Identifier* (URI), que é um dos padrões utilizados pela web. URI são frequentemente referidas como rotas web. Alguns exemplos de URIs:
    - <https://exemplo.com/utilizadores>
    - <https://exemplo.com/utilizadores/5>
    - <https://exemplo.com/equipamentos>
  - **Manipulação de Recursos:** um recurso pode ser representado de várias formas como por exemplo *HyperText Markup Language* (HTML), XML, JSON ou mesmo como um arquivo *Joint Photographic Experts Group* (JPEG). Isto significa que os clientes interagem com os recursos por meio das suas representações, que é uma forma poderosa de manter os conceitos de recursos abstraídos das suas interacções.
  - **Mensagem Auto-descritiva:** cada pedido à API deve conter todas as informações que o serviço precisa para executar o pedido e cada resposta que o serviço retorna contém todas as informações que o cliente precisa para entender a resposta. Os sistemas REST também operam com a noção de um conjunto restrito de tipos de mensagens que são totalmente compreendidos pelo cliente e pelo servidor. O documento que define o HTTP descreve oito tipos de mensagens (Métodos HTTP) que podem ser enviados aos servidores HTTP. Seis deles são amplamente utilizados hoje. São eles: *GET*, *HEAD*, *OPTIONS*, *PUT*, *POST* e *DELETE*. Os três primeiros são mensagens *somente leitura*. Os três últimos são mensagens de atualização. Existem regras bem definidas de como os clientes e servidores se devem comportar ao usar estas mensagens. Os nomes e significados dos elementos de metadados das mensagens (cabeçalhos HTTP) também estão bem definidos. Os sistemas REST entendem e seguem essas regras com muito cuidado. A Tabela posta abaixo, mostra a relação entre operações Create,

Read, Update e Delete (ou, CRUD) e o HTTP. Com o REST, o HTTP passou a ter uma diretiva (ou, verbo, no jargão do REST) para suportar cada operação CRUD.

Operação	Método HTTP
Create	POST
Read	GET
Update	PUT, PATCH
Delete	DELETE

Para exemplificar o princípio de **Mensagem Auto-descritiva**, considere o seguinte arranjo de rotas:

- Rota GET: `http://www.exemplo.io/usuarios`
- Rota DELETE: `http://www.exemplo.io/usuarios/john`
- Rota POST: `http://www.exemplo.io/usuarios -data {nome: mary}`

Quando acessamos a primeira rota, a qual representa uma operação GET, a API responde retornando todos os usuários do site. Já, no segundo caso, quando acessamos a segunda rota, o usuário *john* é removido da aplicação. E por fim, no último exemplo, utiliza-se uma rota POST, em que percebemos o envio de dados extras para cadastrar um novo usuário com o atributo nome definido como *mary*. A simples adoção de verbos específicos para rotas (ou URLs) e por padrões de interoperabilidade de dados resultou na simplificação e melhoria de legibilidade do código de uma API.

Como se pode observar a arquitetura REST tem inúmeras vantagens tais como, permitir a evolução independente dos diferentes componentes do desenvolvimento de um sistema: se, por exemplo, hoje o *front-end* de um sistema REST é escrito em React e posteriormente a equipe de desenvolvimento decidir trocar por uma outra é possível substituir o componente do *front-end* sem modificar uma linha de código de *back-end* (vice-versa também é possível). A separação entre cliente e servidor tem uma vantagem óbvia: o sistema pode escalar sem muitos problemas. Independência em relação a linguagem de programação e tecnologias específicas: um sistema REST é sempre independente do tipo de plataforma ou programação: adapta-se ao tipo de linguagem ou plataformas usadas, o que oferece liberdade considerável na escolha da linguagem ou estrutura para a implementação de um componente.

A seguir, abordamos o padrão arquitetural *Model-View-Controller* (MVC) e a criação de uma API Web MVC com C# utilizando o Visual Studio.

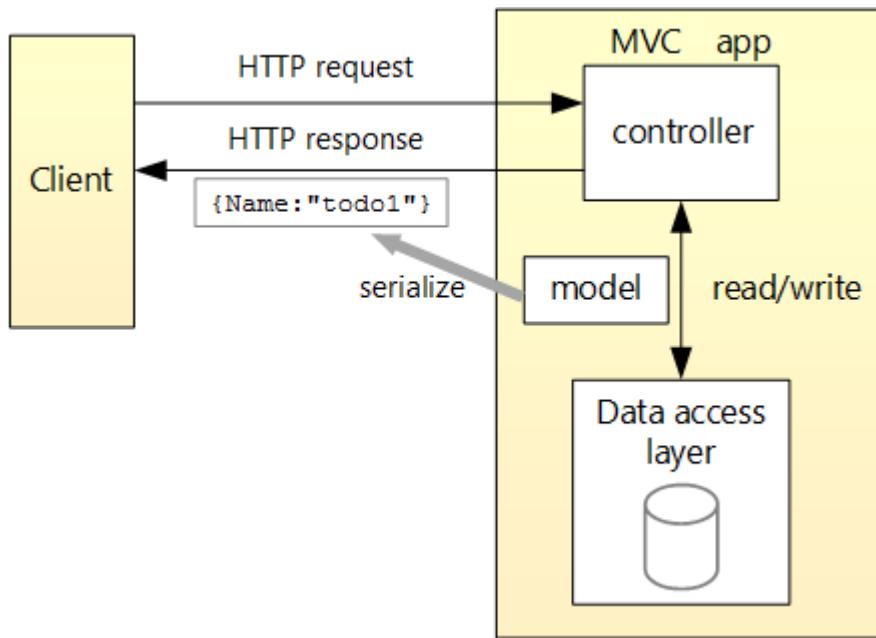
## 4.1 Model View Controller

É comum delimitar as aplicações web em três camadas seguindo o padrão de desenvolvimento MVC:

- A camada de Modelo (*Model*) cuida da persistência dos objetos no servidor;

- A camada de Visão (*View*) é encarregada de apresentar telas, controlar a navegação e a interface com o usuário;
- A camada de Controle (*Controller*) implementa regras de negócio e intermedia a comunicação entre as outras duas camadas.

A Figura que se segue ilustra a disposição e interoperabilidade entre essas camadas:



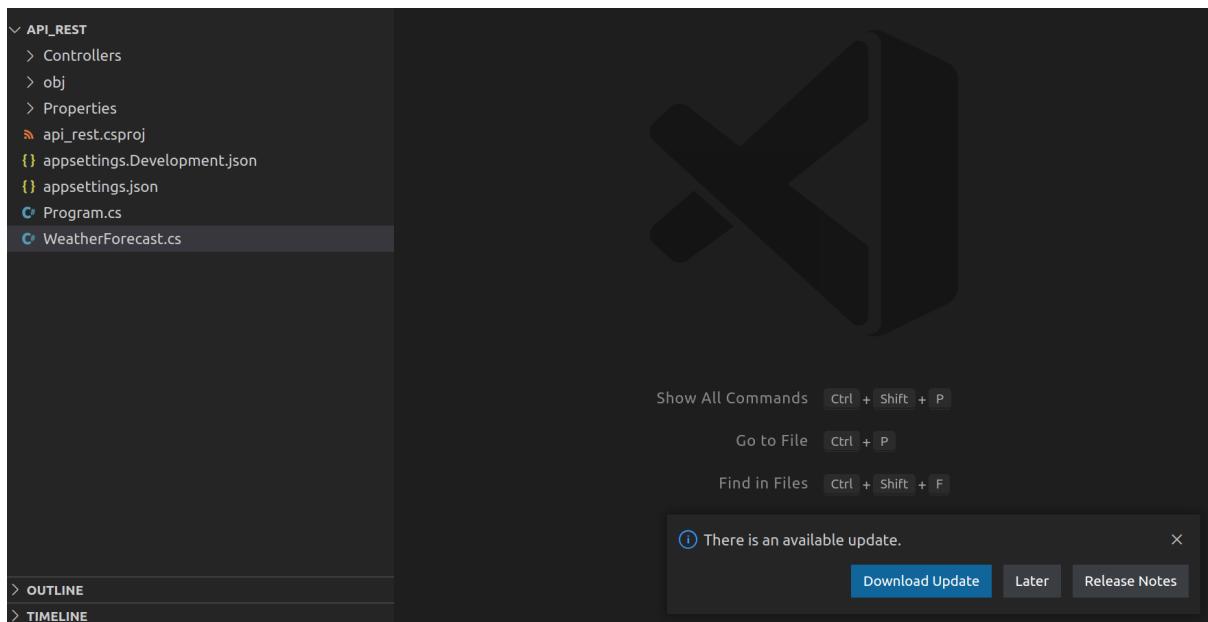
Normalmente, cada interação com o usuário faz com que se altere o estado da aplicação. Como consequência da interação com o usuário, a camada de Visão dispara um evento a ser tratado pela camada de Controle. Ocorre que tecnologias diferentes têm sido empregadas para construir sistemas web e tais tecnologias diferem em como se dá a comunicação entre cliente e servidor.

## 4.2 Criando uma API REST

Com o .NET instalado, abra seu prompt de comando, navegue até uma pasta a qual você utiliza como workspace usando o comando **cd** e digite a seguinte sequência de comandos:

```
dotnet new webapi -o api_rest
cd api_rest
code -r ../api_rest
```

O seu projeto será criado com a seguinte estrutura:

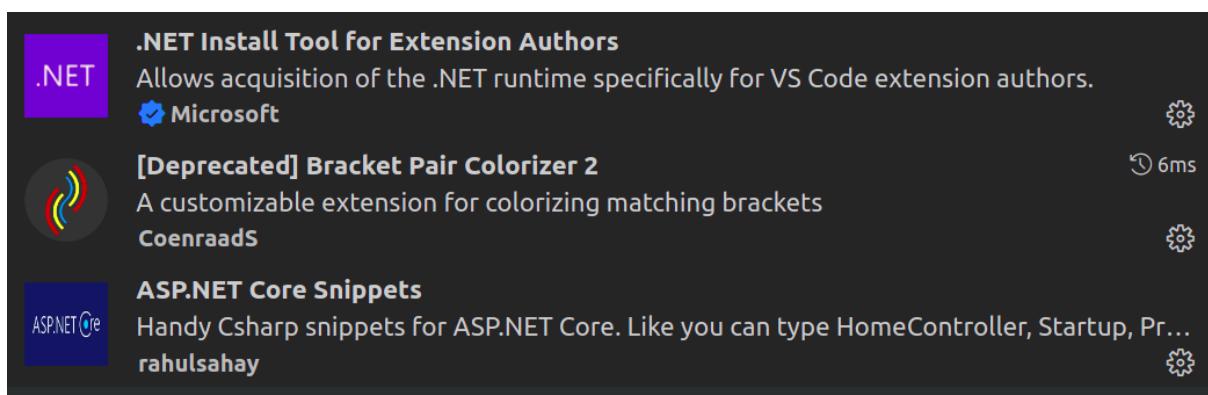


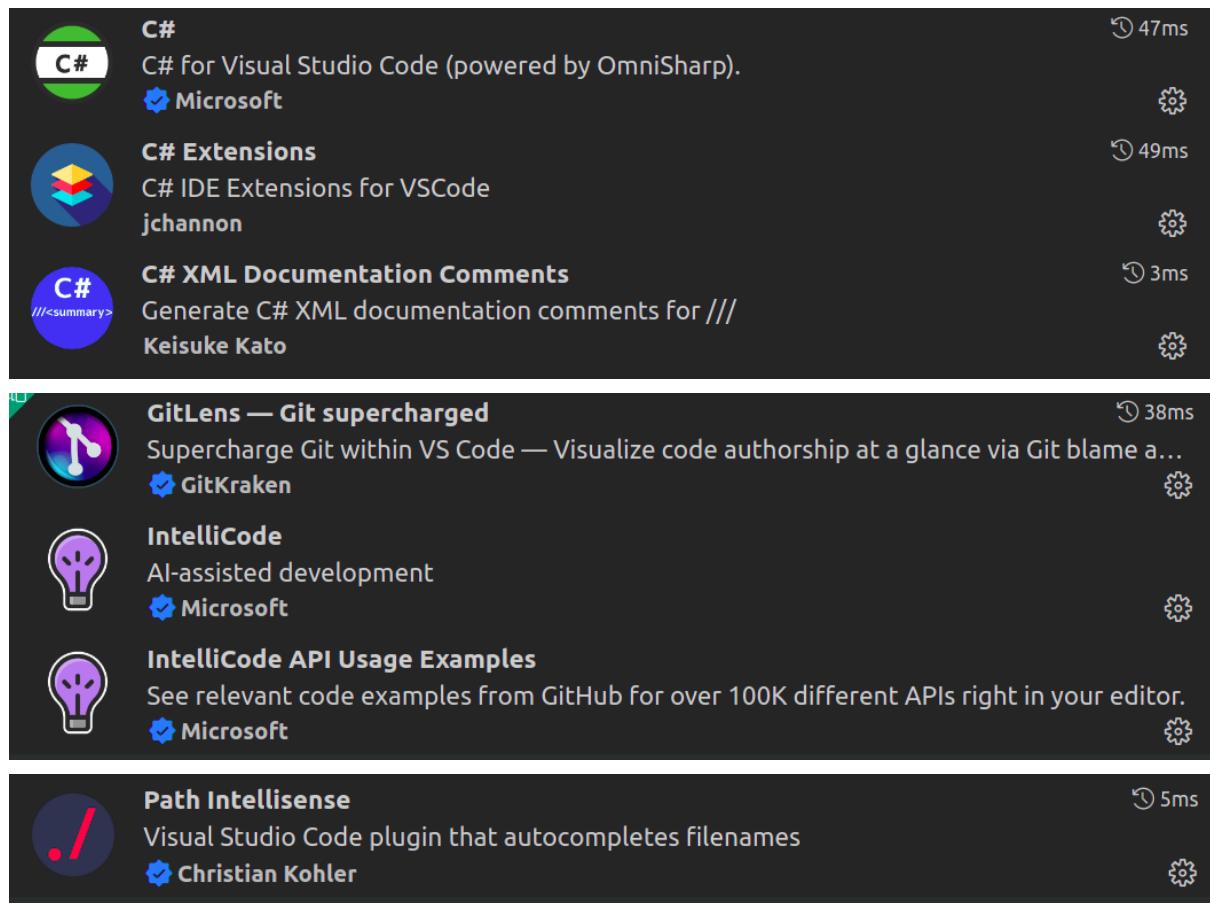
Se você tiver dificuldade em reproduzir o exemplo acima, eis alguns vídeos de versões mais recentes das tecnologias abordadas para se criar um API RESTful:

[https://www.youtube.com/watch?v=AtZ9X\\_3hE6w](https://www.youtube.com/watch?v=AtZ9X_3hE6w)

Quando a aplicação é iniciada, o método **Main**, da classe **Program**, é chamado. Ele cria um *host* padrão usando a configuração de inicialização, expondo o aplicativo via HTTP por meio de uma porta específica (por padrão, porta 5000 para HTTP e 5001 para HTTPS). Um aplicativo ASP.NET Core pode dispor ainda de um grupo de middlewares (pequenas partes do aplicativo anexadas ao core do aplicativo, que manipulam solicitações e respostas) que podem ser configurados em **Program** ou em uma classe à parte (denominada em versões anteriores como classe **Startup**).

Atualmente, alguns plugins que podem agregar bastante a sua produtividade ao programar com .NET e C# no VS Code são elencados como se seguem:





Dê uma olhada na classe **WeatherForecastController** dentro da pasta *Controllers*. Ele expõe métodos que serão chamados quando a API receber solicitações por meio de <https://localhost:7181/weatherforecast> (o host <https://localhost:7181> pode variar). Teste a sua aplicação executando **dotnet run** via prompt de comando no diretório raiz da sua aplicação. Se tudo correu bem, o prompt do VS Code exibir uma saída semelhante a:

```
brunochs@brunochs-Dell-G15-5510:~/workspace/development/fitbank/cybercamp/api_rest$ dotnet run
Compilando...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7181
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5173
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /home/brunochs/workspace/development/fitbank/cybercamp/api_rest/
^info: Microsoft.Hosting.Lifetime[0]
```

Considerando que o host gerado foi <https://localhost:7181>, abra seu browser e navegue conforme a seguinte URI:

<https://localhost:7181/weatherforecast>

Se tudo correu bem, você deverá ver um JSON contendo dados que abstraem informações sobre previsão climática (do inglês weather forecast).

Nas próximas seções, vamos implementar um exemplo de API para um mercado. Seguiremos os conceitos de Padrões de Projeto que manterão o aplicativo simples e fácil de manter. Escrever código que pode ser entendido e mantido por você mesmo não é tão difícil, mas você deve ter em mente que trabalhará como parte de uma equipe. Se você não se importar em como escreve seu código, o resultado será catastrófico para você e a seus colegas de equipe, ocasionando assim constantes dores de cabeça.

A seguir, temos as seguintes seções: implementando a camada de domínio; implementando a camada de controle; implementando a camada de serviços; implementando os repositórios do domínio; implementando uma camada de persistência com ORM; configurando o mecanismo de injeção de dependência; criando um *Resource* para *Category*; criando uma rota POST para criar novas instâncias *Category*; validando uma requisição com body usando Model State; mapeando o *SaveCategoryResource*; aplicando o Padrão de Projeto Request-Response para registrar novas instâncias de *Category*; lógica de Banco de Dados e o Padrão de Projeto Unity of Work; testando a rota POST com um software API Client; implementando uma rota PUT para atualização de dados; implementando uma rota DELETE para remoção de dados.

Os exemplos de implementação que se seguem foram implementados com a seguinte configuração: .NET 6; AutoMapper, versão compatível com .NET 6; e, o VS Code 1.67 (2022). As versões do Visual Studio e do .NET Core podem ser baixadas diretamente pela web. O AutoMapper pode ser instalado direto no projeto usando o NuGet. Para aprender a usar o NuGet, veja este vídeo:

[https://www.youtube.com/watch?v=2r4eTq-AexQ&ab\\_channel=HackdBytes](https://www.youtube.com/watch?v=2r4eTq-AexQ&ab_channel=HackdBytes)

Apesar de as seções subsequentes apresentarem a implementação de uma API com .NET 6, o seguinte repositório:

[https://github.com/brunocastrohs/teaching/tree/main/backend\\_for\\_beginners](https://github.com/brunocastrohs/teaching/tree/main/backend_for_beginners)

Dispõe de implementações da **api\_rest** para as versões 2.2, 3.1, 5 e 6 do .NET em diferentes pastas. Basta clonar o repositório que for compatível com a versão do .NET disponível em sua máquina e estudar as seções subsequentes para entender como implementar a API seguindo conceitos importantes de Padrões de Projeto.

#### 4.2.1 Implementando a camada de domínio

Vamos começar escrevendo a camada de Domínio. Essa camada engloba, além da camada de Modelo, outros componentes importantes sobre o gerenciamento e persistência de dados da nossa API. Esta camada terá nossas classes de modelos, as classes que representarão nossos produtos e categorias, bem como repositórios e interfaces de serviços.

Criamos então uma nova pasta chamada *Domain*. Dentro da nova pasta de domínio, crie outra chamada *Models*. O primeiro modelo que temos que adicionar a esta pasta é a *Category*. Inicialmente, será uma classe simples Plain Old CLR Object (POCO). Isso significa que a classe terá apenas propriedades para descrever suas informações básicas. Até aqui, o seu projeto deve ter a seguinte estrutura.

The screenshot shows the Visual Studio interface with the 'EXPLORER' and 'Category.cs' tabs selected. The 'Category.cs' file is open in the editor, displaying the following code:

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5
6  namespace Dws.Note_one.Api.Domain.Models
7  {
8      public class Category
9      {
10          public int Id { get; set; }
11
12          public string Name { get; set; }
13
14          public IList<Product> Products { get; set; } = new List<Product>();
15
16
17
18      }
19
20 }

```

A classe *Category* possui uma propriedade *Id*, para identificar a categoria, e uma propriedade *Name*. Também temos uma propriedade de produtos. Este último será usado pelo *Entity Framework Core*, o *Object Relational Mapper* (ORM) que a maioria dos aplicativos ASP.NET Core usa para persistir dados em um banco de dados, para mapear o relacionamento entre categorias e produtos. Também faz sentido pensar em termos de paradigma de orientação a objetos, uma vez que uma categoria tem muitos produtos relacionados. Também temos que criar o modelo *Product*. Na mesma pasta, adicione uma nova classe de *Product*.

```

using Dws.Note_one.Api.Domain.Helpers;

namespace Dws.Note_one.Api.Domain.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public short QuantityInPackage { get; set; }
        public EUnitOfMeasurement UnitOfMeasurement { get; set; }

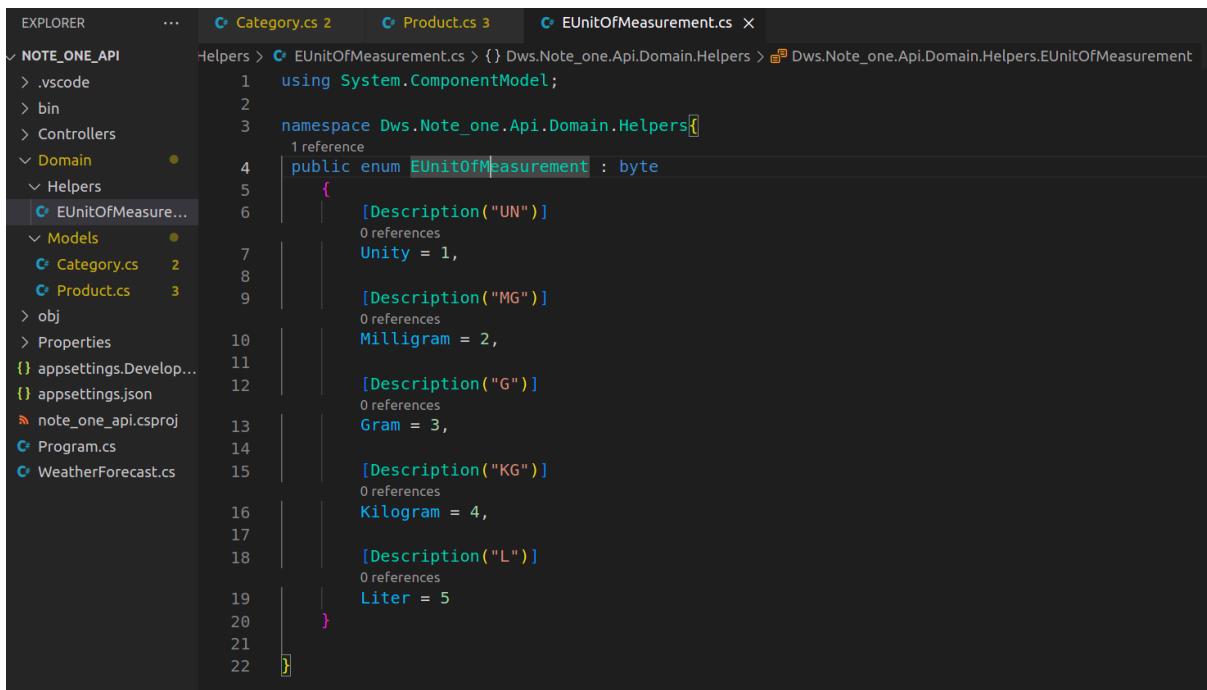
        public int CategoryId { get; set; }
        public Category Category { get; set; }
    }
}

```

}

Uma instância de *Product* também possui propriedades para o Id e o nome. Há também a propriedade *QuantityInPackage*, que informa quantas unidades do produto temos em um pacote (exemplo: quantos biscoitos há em um pacote) e uma propriedade *UnitOfMeasurement*. Este é representado por um tipo *Enum*, que representa uma enumeração das unidades de medida possíveis. As duas últimas propriedades, *CategoryId* e *Category*, serão usadas pelo ORM para mapear o relacionamento entre produtos e categorias. Indica que um produto possui uma, e apenas uma categoria.

Agora, vamos definir a última parte de nossos modelos de domínio, o *Enum EUnitOfMeasurement*. Por convenção, Enums não precisa começar com um “E” antes de seus nomes, mas em algumas bibliotecas e estruturas você encontrará esse prefixo como uma forma de distinguir enums de interfaces e classes.



```
EXPLORER      ...   Category.cs 2   Product.cs 3   EUnitOfMeasurement.cs X
NOTE_ONE_API
  > .vscode
  > bin
  > Controllers
  > Domain
    > Helpers
      EUnitOfMeasurement.cs
    > Models
      Category.cs 2
      Product.cs 3
  > obj
  > Properties
  > appsettings.Develop...
  > appsettings.json
  note_one_api.csproj
  Program.cs
  WeatherForecast.cs

Helpers > EUnitOfMeasurement.cs > {} Dws.Note_one.Api.Domain.Helpers > Dws.Note_one.Api.Domain.Helpers.EUnitOfMeasurement
1  using System.ComponentModel;
2
3  namespace Dws.Note_one.Api.Domain.Helpers
4  {
5      public enum EUnitOfMeasurement : byte
6      {
7          [Description("UN")]
8          Unity = 1,
9          [Description("MG")]
10         0 references
11         Milligram = 2,
12         [Description("G")]
13         0 references
14         Gram = 3,
15         [Description("KG")]
16         0 references
17         Kilogram = 4,
18         [Description("L")]
19         0 references
20         Liter = 5
21     }
22 }
```

#### 4.2.2 Implementando a camada de controle

Na pasta *Controllers*, adicione uma nova classe chamada *CategoryController*. Por convenção, todas as classes nesta pasta que terminam com o sufixo “Controller” se tornarão controladores de nossa aplicação. Isso significa que eles vão lidar com solicitações e respostas. Você deve estender sempre da classe *Controller*, definida no namespace *Microsoft.AspNetCore.Mvc*.

```
EXPLORER ... CategoryController.cs
Controllers > CategoryController.cs > {} Dws.Note_one.Api.Controllers > Dws.Note_one.Api.Controllers.CategoryController
1  using Microsoft.AspNetCore.Mvc;
2
3  namespace Dws.Note_one.Api.Controllers
4  {
5      [Route("/api/[controller]")]
6      public class CategoryController : Controller
7      {
8      }
9
10 }
```

Lembre-se que um namespace consiste em um grupo de classes, interfaces, enums e estruturas relacionadas. Você pode pensar nisso como algo semelhante a módulos da linguagem Javascript ou pacotes de Java.

Vamos começar a lidar com solicitações GET. Em primeiro lugar, como estamos desenvolvendo uma API com princípios REST, quando alguém solicita dados da rota (ou *endpoint*) /api/category por meio do verbo GET, a API precisa retornar todas as categorias. Podemos criar um serviço de categoria para esse fim.

Conceitualmente, um serviço é basicamente uma classe ou interface que define métodos para lidar com alguma lógica de negócios. É uma prática comum em muitas linguagens de programação diferentes criar serviços para lidar com a lógica de negócios, como autenticação e autorização, pagamentos, fluxos de dados complexos, armazenamento em cache e tarefas que requerem alguma interação entre outros serviços ou modelos.

Usando serviços, podemos isolar o tratamento de solicitação e resposta da lógica real necessária para completar tarefas. O serviço que vamos criar inicialmente definirá um único comportamento, ou método: um método de listagem. Esperamos que esse método retorne todas as categorias existentes no banco de dados.

Para definir um comportamento esperado para algo em C# (e em outras linguagens orientadas a objetos, como Java, por exemplo), definimos uma **Interface**. Uma **Interface** informa como algo deve funcionar, mas não implementa a lógica real para o comportamento. A lógica é implementada em classes que implementam a **Interface**.

Na pasta Domain, crie um novo diretório chamado Services e uma subpasta /Services. Em /Services, adicione uma **Interface** chamada *ICategoryService*.

The screenshot shows the VS Code interface. The Explorer pane on the left displays the project structure for 'NOTE\_ONE\_API'. The editor pane on the right shows the code for 'ICategoryService.cs'.

```
1  using System.Collections.Generic;
2  using System.Threading.Tasks;
3  using Dws.Note_one.Api.Domain.Models;
4
5  namespace Dws.Note_one.Api.Domain.Services.IServices
6  {
7      3 references
8      public interface ICategoryService
9      {
10         1 reference
11         Task<IEnumerable<Category>> ListAsync();
12     }
13 }
```

As implementações do método *ListAsync* devem retornar de forma assíncrona um arranjo de categorias. A classe *Task*, encapsulando o retorno, indica assincronia. Precisamos pensar em um método assíncrono devido ao fato de que temos que esperar o banco de dados completar alguma operação para retornar os dados, e esse processo pode demorar um pouco. Observe também o sufixo “*async*”. É uma convenção que indica que nosso método deve ser executado de forma assíncrona.

**Interfaces** nos permitem abstrair o comportamento desejado da implementação real. Valendo-se dos princípios de injeção e inversão de dependência, podemos implementar tipos abstratos, como **Interfaces**, e isolá-los de outros componentes. Basicamente, ao usar injeção de dependência, você define alguns comportamentos usando uma **Interface**. Em seguida, você cria uma classe que implementa a **Interface**. Finalmente, você vincula as referências da **Interface** à classe que criou.

Vamos continuar implementando nossa API e você entenderá por que usar essa abordagem. Altere o código *CategoryController* da seguinte forma:

The screenshot shows the Visual Studio Code interface with the 'EXPLORER' tab selected. In the left sidebar, there's a tree view of the project structure under 'NOTE\_ONE\_API'. The 'Controllers' folder contains 'CategoryController.cs'. The main editor area displays the code for 'CategoryController.cs'.

```
1  using Microsoft.AspNetCore.Mvc;
2  using Dws.Note_one.Api.Domain.Models;
3  using Dws.Note_one.Api.Domain.Services.IServices;
4
5  namespace Dws.Note_one.Api.Controllers
6  {
7      [Route("/api/[controller]")]
8      public class CategoryController : Controller
9      {
10         private readonly ICategoryService _categoryService;
11
12         public CategoryController(ICategoryService categoryService)
13         {
14             _categoryService = categoryService;
15         }
16
17         [HttpGet]
18         public async Task<IEnumerable<Category>> GetAllAsync()
19         {
20             var categories = await _categoryService.ListAsync();
21             return categories;
22         }
23     }
24 }
25
26 }
```

Foi definido um construtor para o *CategoryController* e ele recebe uma instância de *ICategoryService*. Isso significa que a instância pode ser qualquer coisa que implemente a interface de serviço. Eu armazeno esta instância em um campo privado, somente leitura *\_categoryService*. Usaremos este campo para acessar os métodos de implementação de nosso serviço de categoria.

A propósito, o prefixo de sublinhado (utilizado em *\_categoryService*) é outra convenção comum para denotar um campo. Esta convenção, em especial, não é recomendada pela diretriz de convenção de nomenclatura oficial do .NET, mas é uma prática muito comum como forma de evitar o uso da palavra-chave “this” para distinguir campos de classe de variáveis locais.

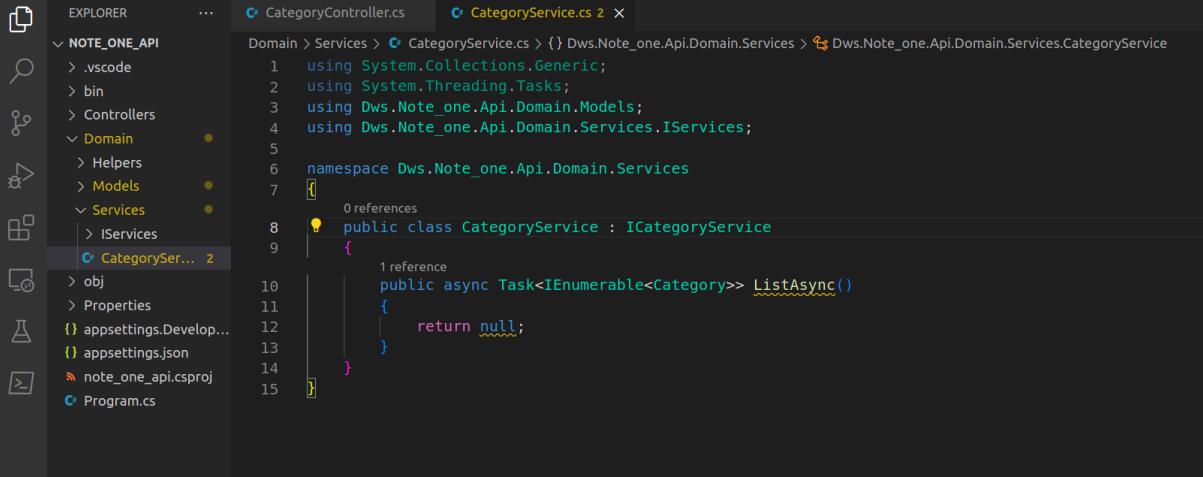
Abaixo do construtor, defini o método que tratará as solicitações de / api / categorias. O atributo *HttpGet* diz ao pipeline do ASP.NET Core para usá-lo para lidar com solicitações GET (esse atributo pode ser omitido, mas é melhor escrevê-lo para facilitar a legibilidade).

O método usa nossa instância de serviço de categoria para listar todas as categorias e, em seguida, retorna as categorias ao cliente. O pipeline da estrutura lida com a serialização de dados para um objeto JSON. O tipo *IEnumerable <Category>* informa à estrutura que queremos retornar uma enumeração de categorias, e o tipo *Task*, precedido pela palavra-chave *async*, informa ao pipeline que esse método deve ser executado de forma assíncrona. Finalmente, quando definimos um método assíncrono, temos que usar a palavra-chave *await* para tarefas que podem demorar um pouco.

Com a estrutura preliminar da nossa API definida, precisamos agora realmente implantar o serviço (ou Services) de *Category*.

## 4.2.3 Implementando a camada de serviços

Em Services, criada em Domain, vamos colocar todas as implementações de serviços. Dentro da nova pasta, adicione uma nova classe chamada *CategoryService*. Altere o código da seguinte forma:



```
1  using System.Collections.Generic;
2  using System.Threading.Tasks;
3  using Dws.Note_one.Api.Domain.Models;
4  using Dws.Note_one.Api.Domain.Services.IServices;
5
6  namespace Dws.Note_one.Api.Domain.Services
7  {
8      public class CategoryService : ICategoryService
9      {
10         public async Task<IEnumerable<Category>> ListAsync()
11         {
12             return null;
13         }
14     }
15 }
```

É simplesmente o código básico para a implementação da interface, mas ainda não lidamos com nenhuma lógica. Vamos pensar em como o método de listagem deve funcionar. Precisamos acessar o banco de dados e retornar todas as categorias, então precisamos retornar esses dados ao cliente. Uma classe de serviço não é uma classe que deve lidar com o acesso a dados. Existe um padrão chamado Padrão de Repositório que é usado para gerenciar dados de bancos de dados.

Ao usar o Padrão de Repositório, definimos classes de repositório, que basicamente encapsulam toda a lógica para lidar com o acesso aos dados. Esses repositórios expõem métodos para listar, criar, editar e excluir objetos de um determinado modelo, da mesma forma que você pode manipular coleções. Internamente, esses métodos se comunicam com o banco de dados para realizar operações CRUD, isolando o acesso ao banco de dados do restante da aplicação.

Nosso serviço precisa se comunicar com um repositório de categorias, para obter a lista de objetos. Conceitualmente, um serviço pode “conversar” com um ou mais repositórios ou outros serviços para realizar operações.

## 4.2.4 Implementando os repositórios do domínio

Dentro da pasta Domain, crie uma nova pasta chamada Repositories e IRepository. Em seguida, adicione uma nova interface chamada ICategoryRepository. Defina a interface da seguinte forma:

The screenshot shows the Visual Studio Code interface. On the left is the Explorer sidebar with project files like .vscode, bin, Controllers, Domain (Helpers, Models, Category.cs, Product.cs), Repositories, IRepositories, ICategoryRepository.cs (selected), CategoryRepository.cs, Services, obj, Properties, appsettings.Development.json, appsettings.json, note\_one\_api.csproj, and Program.cs. The main area is a code editor showing the file ICategoryRepository.cs:

```
Domain > Repositories > IRepositories > ICategoryRepository.cs > {} Dws.Note_one.Api.Domain.Repositories
1  using System.Collections.Generic;
2  using System.Threading.Tasks;
3  using Dws.Note_one.Api.Domain.Models;
4
5  namespace Dws.Note_one.Api.Domain.Repositories.IRepositories
6  {
7      public interface ICategoryRepository
8      {
9          Task<IEnumerable<Category>> ListAsync();
10     }
11 }
```

O código inicial é basicamente idêntico ao código da interface de *Services* em *Domain*. Tendo definido a interface, podemos voltar à classe de serviço e concluir a implementação do método de listagem, usando uma instância de *ICategoryRepository* para retornar os dados:

The screenshot shows the Visual Studio Code interface. On the left is the Explorer sidebar with project files like .vscode, bin, Controllers, Domain (Helpers, Models, Category.cs, Product.cs), Repositories, IServices, CategoryService.cs (selected), Services, obj, Properties, appsettings.Development.json, appsettings.json, note\_one\_api.csproj, and Program.cs. The main area is a code editor showing the file CategoryService.cs:

```
Domain > Services > CategoryService.cs > ...
1  using System.Collections.Generic;
2  using System.Threading.Tasks;
3  using Dws.Note_one.Api.Domain.Models;
4  using Dws.Note_one.Api.Domain.Services.IServices;
5  using Dws.Note_one.Api.Domain.Repositories.IRepositories;
6
7  namespace Dws.Note_one.Api.Domain.Services
8  {
9      public class CategoryService : ICategoryService
10     {
11         private readonly ICategoryRepository _categoryRepository;
12
13         public CategoryService(ICategoryRepository categoryRepository)
14         {
15             this._categoryRepository = categoryRepository;
16         }
17
18         public async Task<IEnumerable<Category>> ListAsync()
19         {
20             return await _categoryRepository.ListAsync();
21         }
22     }
23 }
```

Agora temos que implementar a lógica real do repositório de categorias. Antes de fazer isso, temos que pensar em como vamos acessar o banco de dados.

## 4.2.5 Implementando uma camada de persistência com ORM

Usaremos o Entity Framework Core (ou EF Core para simplificar) como mecanismo de ORM para abstrair a interoperabilidade com um banco de dados. Esta estrutura vem com ASP.NET Core como seu ORM padrão e expõe uma API amigável que nos permite mapear classes de nossos aplicativos para tabelas de banco de dados.

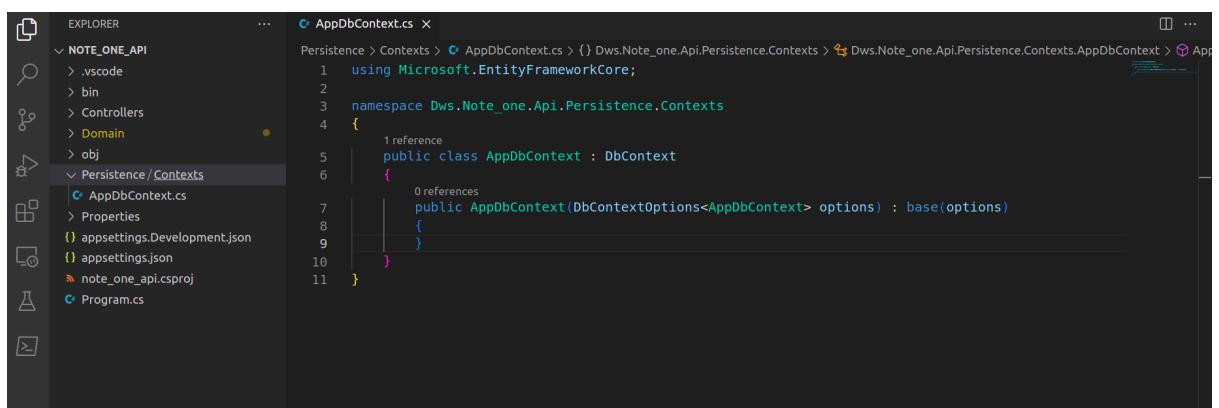
Para instalar o pacote EF Core ao seu projeto, vá ao terminal do VS Code e no diretório raiz da sua aplicação, execute os comandos:

```
dotnet add package Microsoft.EntityFrameworkCore  
dotnet add package Microsoft.EntityFrameworkCore.Relational  
dotnet add package Microsoft.EntityFrameworkCore.InMemory
```

```
brunochs@brunochs-Dell-G15-5510:~/workspace/academic/dws/note_one_api$ dotnet add package Microsoft.EntityFrameworkCore
```

O EF Core também nos permite projetar nosso aplicativo primeiro e, em seguida, gerar um banco de dados de acordo com o que definimos em nosso código. Essa técnica é chamada de *code-first*. Usaremos a abordagem *code-first* para gerar um banco de dados (neste exemplo, na verdade, vou usar um banco de dados in-memory, mas você será capaz de alterá-lo facilmente para uma instância de servidor SQL Server ou MySQL, por exemplo).

Na pasta raiz da API, crie um novo diretório chamado *Persistence*. Este diretório terá tudo o que precisamos para acessar o banco de dados, como implementações de repositórios. Dentro da nova pasta, crie um novo diretório chamado **Context** e, em seguida, adicione uma nova classe chamada *AppDbContext*. Esta classe deve herdar *DbContext*, uma classe que EF Core usa para mapear seus modelos para tabelas de banco de dados. Altere o código da seguinte maneira:



O construtor que adicionamos a esta classe é responsável por passar a configuração do banco de dados para a classe base por meio de injeção de dependência. Você verá em um momento como isso funciona.

Agora, temos que criar duas propriedades DbSet. Essas propriedades são conjuntos (coleções de objetos exclusivos) que mapeiam modelos para tabelas de banco de dados.

Além disso, temos que mapear as propriedades dos modelos para as respectivas colunas da tabela, especificando quais propriedades são chaves primárias, quais são as chaves estrangeiras, os tipos de coluna, etc. Podemos fazer isso substituindo o método OnModelCreating, usando um recurso chamado API Fluent para especificar o mapeamento do banco de dados. Altere a classe *AppDbContext* da seguinte maneira:

```
using Microsoft.EntityFrameworkCore;
using Dws.Note_one.Api.Domain.Models;

namespace Dws.Note_one.Api.Persistence.Contexts
{
    public class AppDbContext : DbContext
    {
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }

        public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);

            builder.Entity<Category>().ToTable("Categories");
            builder.Entity<Category>().HasKey(p => p.Id);
                builder.Entity<Category>().Property(p => p.Id).IsRequired().ValueGeneratedOnAdd();
                builder.Entity<Category>().Property(p => p.Name).IsRequired().HasMaxLength(30);
                builder.Entity<Category>().HasMany(p => p.Products).WithOne(p => p.Category).HasForeignKey(p => p.CategoryId);

            builder.Entity<Category>().HasData
            (
                new Category { Id = 100, Name = "Fruits and Vegetables" },
                // Id set manually due to in-memory provider
                new Category { Id = 101, Name = "Dairy" }
            );

            builder.Entity<Product>().ToTable("Products");
            builder.Entity<Product>().HasKey(p => p.Id);
```

```

        builder.Entity<Product>().Property(p =>
p.Id).IsRequired().ValueGeneratedOnAdd();
        builder.Entity<Product>().Property(p =>
p.Name).IsRequired().HasMaxLength(50);
        builder.Entity<Product>().Property(p =>
p.QuantityInPackage).IsRequired();
        builder.Entity<Product>().Property(p =>
p.UnitOfMeasurement).IsRequired();
    }
}

}

```

Especificamos para quais tabelas nossos modelos devem ser mapeados. Além disso, definimos as chaves primárias, usando o método HasKey, as colunas da tabela, usando o método Property e algumas restrições, como IsRequired, HasMaxLength e ValueGeneratedOnAdd, tudo com expressões lambda de uma "forma fluente" (métodos de encadeamento). Dê uma olhada no seguinte trecho de código:

```

builder.Entity<Category>()
    .HasMany(p => p.Products)
    .WithOne(p => p.Category)
    .HasForeignKey(p => p.CategoryId);

```

Aqui, especificamos uma relação entre as tabelas. Dizemos que uma categoria tem muitos produtos e definimos as propriedades que irão mapear essa relação (Products, da classe Category, e Category, da classe Product). Também definimos a chave estrangeira (CategoryId).

Tendo implementado a classe de contexto do banco de dados, podemos implementar o repositório de categorias. Adicione a pasta `Repositories` dentro da pasta `Domain`, uma nova classe chamada `BaseRepository`.

```

    BaseRepository.cs
    Domain > Repositories > BaseRepository.cs > ...
    1  using Dws.Note_one.Api.Persistence.Context;
    2
    3  namespace Dws.Note_one.Api.Domain.Repositories
    4  {
    5      0 references
    6      public abstract class BaseRepository
    7      {
    8          1 reference
    9          protected readonly AppDbContext _context;
    10
    11         0 references
    12         public BaseRepository(AppDbContext context)
    13         {
    14             _context = context;
    15         }
    16     }
    17 }

```

Esta classe é apenas uma classe abstrata que todos os nossos repositórios herdarão. Uma classe abstrata é uma classe que não possui instâncias diretas. Você tem que criar classes diretas para criar as instâncias.

O *BaseRepository* recebe uma instância de nosso *AppDbContext* por meio de injeção de dependência e expõe uma propriedade protegida (uma propriedade que só pode ser acessível pelas classes filhas) chamada *\_context*, que dá acesso a todos os métodos de que precisamos para lidar com as operações do banco de dados.

Adicione uma nova classe na mesma pasta chamada *CategoryRepository*. Agora vamos realmente implementar a lógica do repositório:

```

    CategoryRepository.cs
    Domain > Repositories > CategoryRepository.cs > ...
    1  using System.Collections.Generic;
    2  using System.Threading.Tasks;
    3  using Microsoft.EntityFrameworkCore;
    4  using Dws.Note_one.Api.Domain.Models;
    5  using Dws.Note_one.Api.Domain.Repositories.IRepositories;
    6  using Dws.Note_one.Api.Persistence.Context;
    7
    8  namespace Dws.Note_one.Api.Domain.Repositories
    9  {
    10     0 references
    11     public class CategoryRepository : BaseRepository, ICategoryRepository
    12     {
    13         0 references
    14         public CategoryRepository(AppDbContext context) : base(context)
    15         {
    16         }
    17
    18         1 reference
    19         public async Task<IEnumerable<Category>> ListAsync()
    20         {
    21             return await _context.Categories.ToListAsync();
    22         }
    23     }
    24 }

```

O repositório herda o *BaseRepository* e implementa *ICategoryRepository*. Observe como é simples implementar o método de listagem. Usamos o banco de dados Categories definido para acessar a tabela de categorias e, em seguida, chamamos o método de extensão *ToListAsync*, que é responsável por transformar o resultado de uma consulta em uma coleção de categorias.

O EF Core traduz nossa chamada de método para uma consulta SQL, da maneira mais eficiente possível. A consulta só é executada quando você chama um método que irá transformar seus dados em uma coleção ou quando você usa um método para obter dados específicos.

Agora temos uma implementação limpa do controlador de categorias, o serviço e o repositório. Separamos interesses, criando classes que fazem apenas o que deveriam fazer.

A última etapa antes de testar o aplicativo é vincular nossas interfaces às respectivas classes usando o mecanismo de injeção de dependência ASP.NET Core.

#### 4.2.6 Configurando o mecanismo de injeção de dependência

Na pasta raiz do aplicativo, abra a classe *Startup*. Esta classe é responsável por definir todos os tipos de configurações quando o aplicativo é iniciado.

Os métodos *ConfigureServices* e *Configure* são chamados em tempo de execução pelo pipeline da estrutura para configurar como o aplicativo deve funcionar e quais componentes ele deve usar.

Dê uma olhada no método *ConfigureServices*. Neste método, temos apenas uma linha, que configura o aplicativo para usar o pipeline MVC, o que basicamente significa que o aplicativo vai lidar com solicitações e respostas usando classes de controlador (há mais coisas acontecendo aqui nos bastidores, mas é o que você precisa saber por enquanto).

Podemos usar o método *ConfigureServices*, acessando o parâmetro *services*, para configurar nossas ligações de dependência. Limpe o código da classe removendo todos os comentários e altere o código da seguinte maneira:

```
using Dws.Note_one.Api.Persistence.Context;
using Dws.Note_one.Api.Domain.Repositories;
using Dws.Note_one.Api.Domain.Repositories.IRepositories;
using Dws.Note_one.Api.Domain.Services;
using Dws.Note_one.Api.Domain.Services.IServices;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
```

```
builder.Services.AddControllers();
builder.Services.AddDbContext<AppDbContext>(options => {
    options.UseInMemoryDatabase("groceries-api-in-memory");
});
builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
builder.Services.AddScoped<ICategoryService, CategoryService>();
// Learn more about configuring Swagger/OpenAPI at 
https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Observe o seguinte trecho de código:

```
builder.Services.AddDbContext<AppDbContext>(options => {
    options.UseInMemoryDatabase("groceries-api-in-memory");
});
```

Aqui, configuramos o contexto do banco de dados. Dizemos ao .NET para usar nosso AppDbContext com uma implementação de banco de dados na memória, que é identificada pela string passada como um argumento para nosso método. Normalmente, o provedor de memória é usado quando escrevemos testes de integração, mas estou usando-o aqui para simplificar. Dessa forma, não precisamos nos conectar a um banco de dados real para testar o aplicativo.

A configuração dessas linhas configura internamente nosso contexto de banco de dados para injeção de dependência usando um tempo de vida com escopo definido.

O tempo de vida com escopo informa ao pipeline do ASP.NET Core que sempre que ele precisa resolver uma classe que recebe uma instância de AppDbContext como um argumento do construtor, ele deve usar a mesma instância da classe. Se não houver instância na memória, o pipeline criará uma nova instância e a reutilizará em todas as classes que precisam dela, durante uma determinada solicitação. Dessa forma, você não precisa criar manualmente a instância da classe quando precisar usá-la.

A técnica de injeção de dependência nos oferece muitas vantagens, como:

- Reutilização de código;
- Melhor produtividade, pois quando temos que mudar a implementação, não precisamos nos preocupar em mudar uma centena de lugares onde você usa esse recurso;
- Você pode facilmente testar o aplicativo, pois podemos isolar o que temos para testar usando simulações (implementação falsa de classes), onde temos que passar interfaces como argumentos de construtor;
- Quando uma classe precisa receber mais dependências por meio de um construtor, você não precisa alterar manualmente todos os lugares onde as instâncias estão sendo criadas (isso é incrível!).

Depois de configurar o contexto do banco de dados, também vinculamos nosso serviço e repositório às respectivas classes. Agora, observe este outro trecho de código:

```
builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
builder.Services.AddScoped<ICategoryService, CategoryService>();
```

Aqui também usamos um tempo de vida com escopo porque essas classes precisam usar internamente a classe de contexto do banco de dados. Faz sentido especificar o mesmo escopo neste caso.

Agora que configuramos nossos vínculos de dependência, temos que fazer uma pequena mudança na classe Program, para que o banco de dados propague corretamente nossos dados iniciais. Esta etapa é necessária apenas ao usar o provedor de banco de dados na memória.

```
using Dws.Note_one.Api.Persistence.Context;
using Dws.Note_one.Api.Domain.Repositories;
using Dws.Note_one.Api.Domain.Repositories.IRepositories;
using Dws.Note_one.Api.Domain.Services;
using Dws.Note_one.Api.Domain.Services.IServices;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
```

```

// Add services to the container.

builder.Services.AddControllers();
builder.Services.AddDbContext<AppDbContext>(options =>
{
    options.UseInMemoryDatabase("groceries-api-in-memory");
});
builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
builder.Services.AddScoped<ICategoryService, CategoryService>();
// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

var scope = app.Services.CreateScope();
using (var context = scope.ServiceProvider.GetService<AppDbContext>())
{
    context.Database.EnsureCreated();
}

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

```

Foi necessário alterar a configuração de Program para garantir que nosso banco de dados será "criado" quando o aplicativo for iniciado, pois estamos usando um provedor in-memory. Sem essa mudança, as categorias que queremos semear não serão criadas.

Com todos os recursos básicos implementados, é hora de testar nosso endpoint de API. Basta navegar com o seu prompt de comandos até a pasta em que está o root do seu projeto e executar o comando **dotnet run**. Feito isto, a saída no prompt deve ser algo como:

```
Info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 2.2.4-servicing-10062 initialized 'AppDbContext' using provider 'Microsoft.EntityFrameworkCore.InMemory' with options: StoreName=supermarket
- api-in-memory
Info: Microsoft.EntityFrameworkCore.Update[30100]
      Saved 2 entities to in-memory store.
Info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\bruno\AppData\Local\ASP.NET\DataProtection-Keys' as key repository and Windows DPAPI to encrypt keys at rest.
Hosting environment: Development
Content root path: C:\workspace\api_rest\api_rest
Now listening on: https://localhost:5001
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Para testar a rota GET que implementamos em nosso CategoryController, basta acessar a rota (ou *endpoint*) considerando o host do seu http-server produzido ao rodar a aplicação: (<https://localhost:5001/api/category>). Em um browser, a saída deve ser essa:



Aqui, vemos os dados que adicionamos ao banco de dados quando configuramos o contexto do banco de dados. Esta saída confirma que nosso código está funcionando. Você criou um endpoint de API com a diretiva, ou verbo, GET.

#### 4.2.7 Criando um Resource para Category

Observe que na resposta JSON da rota GET testada e ilustrada na seção anterior há uma propriedade extra: um *array* de produtos.

```
1
2   {
3     "id": 100,
4     "name": "Fruits and Vegetables",
5     "products": []
6   },
7   {
8     "id": 101,
9     "name": "Dairy",
10    "products": []
11 }
```

O array de produtos está presente em nossa resposta JSON atual porque nosso modelo de Category tem uma propriedade de produtos, necessária para EF Core para mapear corretamente os produtos de uma determinada categoria.

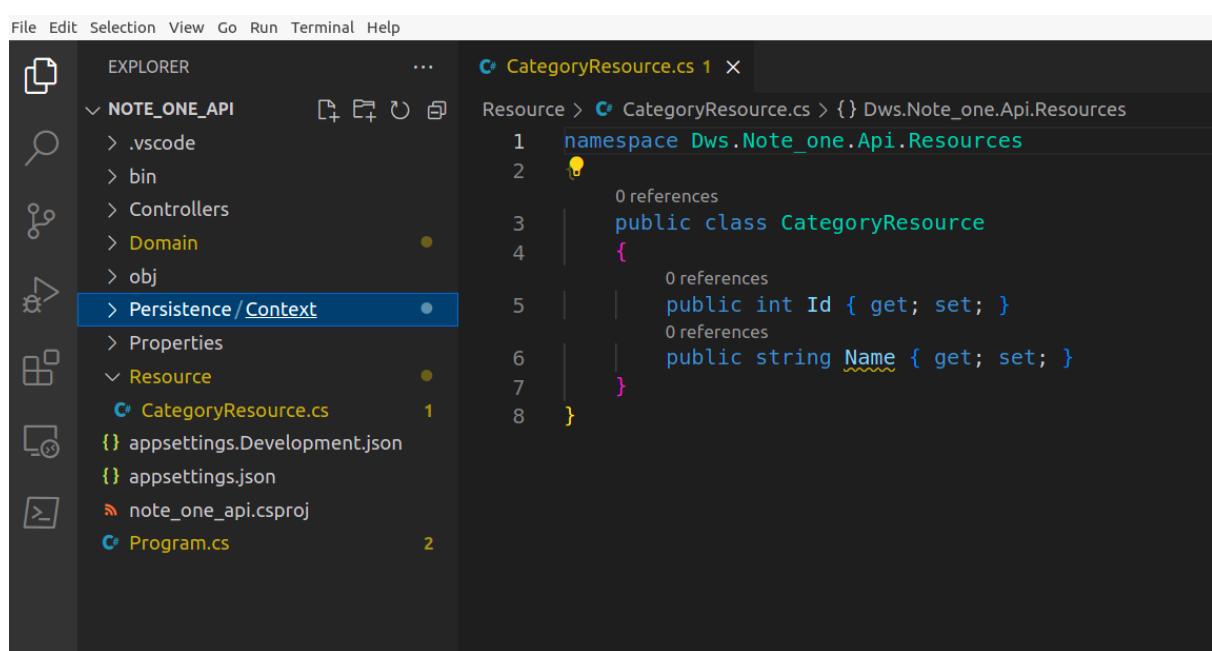
Não queremos essa propriedade em nossa resposta, mas não podemos alterar nossa classe de modelo para excluir essa propriedade. Isso faria com que o EF Core lançasse erros quando tentássemos gerenciar os dados das categorias, e também quebraria o design do nosso modelo de domínio porque não faz sentido ter uma categoria de produto que não tenha produtos.

Para retornar dados JSON contendo apenas os identificadores e nomes das categorias de supermercado, temos que criar uma classe de recursos, i. e., uma classe Resource.

Uma classe Resource é uma classe que contém apenas informações básicas que serão trocadas entre aplicativos cliente e rotas de API, geralmente na forma de dados JSON, para representar algumas informações específicas.

Todas as respostas das rotas de uma API devem retornar um Resource. É uma má prática retornar a representação do modelo real como a resposta, pois ela pode conter informações que o aplicativo cliente não precisa ou que não tem permissão para ter (por exemplo, um modelo de usuário pode retornar informações da senha do usuário , o que seria um grande problema de segurança).

Precisamos de uma classe Resource para representar apenas nossas categorias, sem os produtos. Sendo assim, implementaremos a nossa classe `CategoryResource`. Se a API ainda estiver rodando, pressione Ctrl + C junto ao *prompt* para interromper a sua execução. Na pasta raiz da API, crie uma nova pasta chamada **Resources**. Lá, adicione uma nova classe chamada `CategoryResource`:



```
File Edit Selection View Go Run Terminal Help
EXPLORER ...
NOTE_ONE_API ...
    .vscode
    bin
    Controllers
    Domain
    obj
    Persistence_Context
        .
    Properties
    Resource ...
        CategoryResource.cs 1
    appsettings.Development.json
    appsettings.json
    note_one_api.csproj
    Program.cs 2
CategoryResource.cs 1 X
Resource > CategoryResource.cs > {} Dws.Note_one.Api.Resources
1 namespace Dws.Note_one.Api.Resources
2
3     0 references
4         public class CategoryResource
5             0 references
6                 public int Id { get; set; }
7                     0 references
8                 public string Name { get; set; }
9 }
```

Temos que mapear nossa coleção de modelos de *Category*, que é fornecido por nosso *CategoryService*, para uma coleção de *CategoryResource*.

Usaremos uma biblioteca chamada *AutoMapper* para lidar com o mapeamento entre objetos. *AutoMapper* é uma biblioteca muito popular no mundo .NET e é usada em muitos projetos comerciais e de código aberto. Para adicionar ao *AutoMapper* ao repositório de bibliotecas importadas para o projeto, use o **NuGet** ou abra o seu prompt, navegue até o root do seu projeto e execute o comando que se segue:

```
dotnet add package AutoMapper  
dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection
```

Para usar o *AutoMapper*, temos que fazer duas coisas:

- Configurá-lo conforme a injeção de dependência que desejamos impor;
- Criar uma classe que dirá ao *AutoMapper* como lidar com o mapeamento de classes.

Para tal, a primeira orientação é: abra a classe **Program.cs**. No método *ConfigureServices*, após a ultima linha, adicione o seguinte código:

```
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies())  
;
```

Esta linha lida com todas as configurações necessárias do *AutoMapper*, como registrá-lo para injeção de dependência e escanear o aplicativo durante a inicialização para configurar perfis de mapeamento.

Agora, no diretório raiz, adicione uma nova pasta chamada *Mapping* e, em seguida, adicione uma classe chamada **ModelToResourceProfile**. O fonte desse classe é dado a seguir:

The screenshot shows a code editor with a dark theme. On the left is the file explorer showing a project structure with folders like *NOTE\_ONE\_API*, *Controllers*, *Domain*, and *Mapping*. Inside *Mapping*, the file *ModelToResourceProfile.cs* is selected. The right pane shows the code for *ModelToResourceProfile.cs*:

```
EXPLORER ... ModelToResourceProfile.cs X  
NOTE_ONE_API  
  .vscode  
  bin  
  Controllers  
  Domain  
  Mapping  
    ModelToResourceProfile.cs  
    obj  
    Persistence  
    Properties  
    Resource  
      CategoryResource.cs  
  appsettings.Development.json  
  appsettings.json  
  note_one_api.csproj  
  Program.cs  
  1  
  2  
  3  
  4  
  5  
  6  
  7  
  8  
  9  
  10  
  11  
  12  
  13  
  14  
Mapping > ModelToResourceProfile.cs > {} Supermarket.API.Mapping  
1  using AutoMapper;  
2  using Dws.Note_one.Api.Domain.Models;  
3  using Dws.Note_one.Api.Resources;  
4  namespace Supermarket.API.Mapping  
5  {  
6  }  
0 references  
public class ModelToResourceProfile : Profile  
{  
} 0 references  
public ModelToResourceProfile()  
{  
  CreateMap<Category, CategoryResource>();  
}  
}
```

A classe **ModelToResourceProfile** herda **Profile**, um tipo de classe que o *AutoMapper* usa para verificar como nossos mapeamentos funcionarão. No construtor, criamos um mapa entre a classe do modelo **Category** e a classe **CategoryResource**. Visto que as propriedades das classes têm os mesmos nomes e tipos, não precisamos usar nenhuma configuração especial para elas.

A etapa final para operacionalizar o *AutoMapper* na nossa aplicação consiste em alterar o **CategoryController** para usar o *AutoMapper* para lidar com o mapeamento de nossos objetos:

```
using Microsoft.AspNetCore.Mvc;
using AutoMapper;
using Dws.Note_one.Api.Domain.Models;
using Dws.Note_one.Api.Resource;
using Dws.Note_one.Api.Domain.Services.IServices;
using Dws.Note_one.Api.Mapping;

namespace Dws.Note_one.Api.Controllers
{
    [Route("/api/[controller]")]
    public class CategoriesController : Controller
    {
        private readonly ICategoryService _categoryService;
        private readonly IMapper _mapper;

        public CategoriesController(ICategoryService categoryService,
IMapper mapper)
        {
            _categoryService = categoryService;
            _mapper = mapper;
        }

        [HttpGet]
        public async Task<IEnumerable<CategoryResource>> GetAllAsync()
        {
            var categories = await _categoryService.ListAsync();
            var resources = _mapper.Map<IEnumerable<Category>,
IEnumerable<CategoryResource>>(categories);

            return resources;
        }
    }
}
```

```
}
```

Note que o construtor de **CategoryController** foi alterado para receber uma instância de implementação **IMapper**. Estes métodos de **Interface** são usados para usar acionar o mapeamento do *AutoMapper*.

Repare também que o método *GetAllAsync* foi alterado para mapear nossa enumeração de **Category** para uma enumeração de **CategoryResource** usando o método *Map*. Este método recebe uma instância da classe ou coleção que queremos mapear e, por meio de definições de tipo genérico, define para qual tipo de classe ou coleção deve ser mapeado.

Observe que mudamos facilmente a implementação sem ter que adaptar a classe de serviço ou repositório, simplesmente injetando uma nova dependência (**IMapper**) no construtor. A injeção de dependência torna seu aplicativo sustentável e fácil de mudar, já que você não precisa quebrar toda a implementação de seu código para adicionar ou remover recursos.

Agora, via prompt, inicie a API novamente usando o comando ‘**dotnet run**’ e acesse `http://localhost:5000/api/category` (*lembre-se que o seu host pode ser diferente de localhost:5000*) para ver a nova resposta JSON. Com a rota GET bem definida segundo o contexto da aplicação, vamos partir para a criação de uma nova rota para criar categorias: o POST.

#### 4.2.8 Criando uma rota POST para criar novas instâncias Category

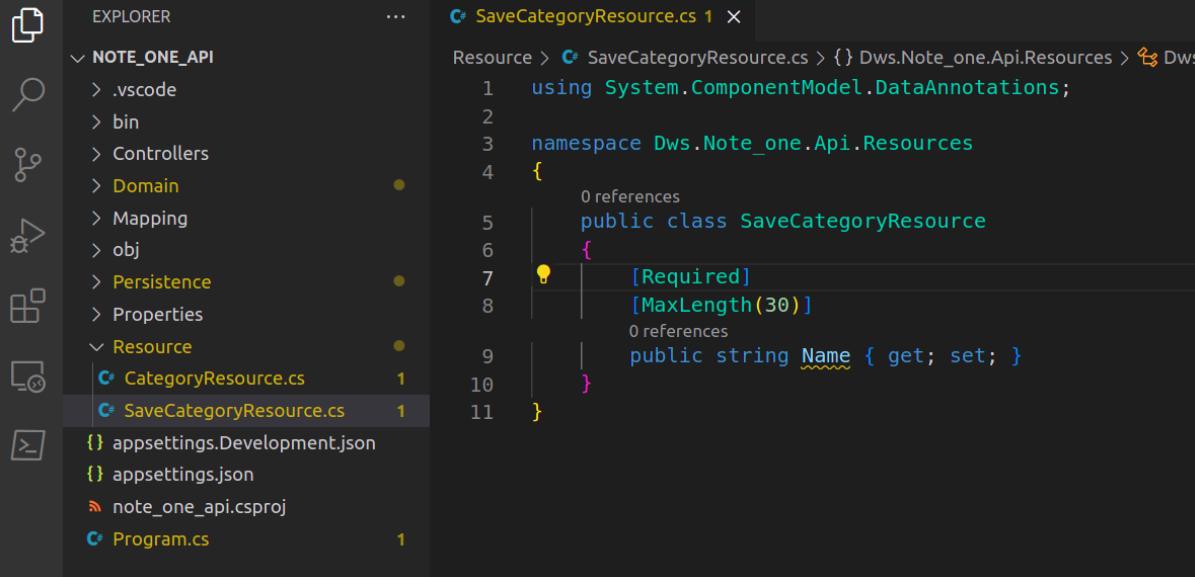
Ao lidar com a criação de novas instâncias de uma determinada entidade, devemos nos preocupar com muitas coisas, tais como:

- Validação de dados e integridade de dados;
- Autorização para criar instâncias;
- Manipulação de erros;
- *Logging* de falhas e êxitos.

Neste Módulo, não será abordado como lidar com autenticação e autorização. No Módulo II, haverá uma seção sobre como implementar facilmente esses recursos via autenticação de token utilizando o formato JSON. Além disso, há uma estrutura muito popular chamada **ASP.NET Identity** que fornece soluções integradas em relação à segurança e ao registro de usuários que você pode usar em seus aplicativos. Inclui provedores para trabalhar com EF Core, como um *IdentityDbContext* integrado.

Antes de criar a nova rota POST, precisamos de um novo *Resource* o qual mapeará os dados que os aplicativos clientes enviam para esta rota. Já que estamos criando uma nova categoria, não temos um ID ainda (visto que o ID é gerado automaticamente e em tempo de execução), e isso significa que precisamos de um recurso que representa uma **Category** contendo apenas seu nome.

Na pasta *Resources*, adicione uma nova classe chamada **SaveCategoryResource**:

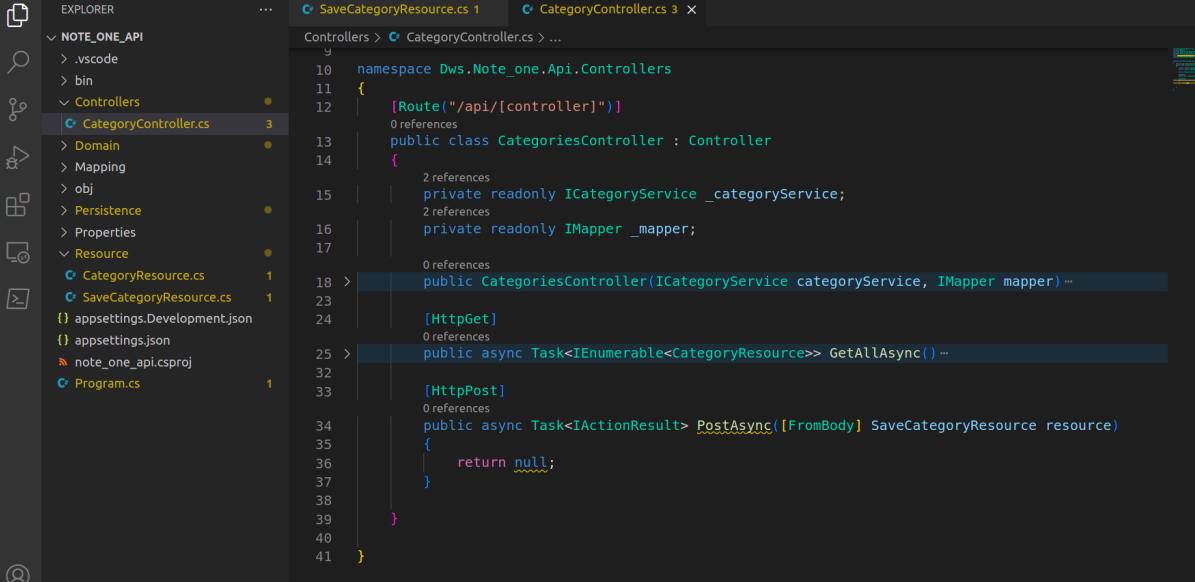


The screenshot shows the VS Code interface with the Explorer sidebar on the left. In the Resources folder, there are files: .vscode, bin, Controllers, Domain, Mapping, obj, Persistence, Properties, Resource, CategoryResource.cs, and SaveCategoryResource.cs. The SaveCategoryResource.cs file is selected and open in the editor. The code defines a class SaveCategoryResource with a required string property Name.

```
1  using System.ComponentModel.DataAnnotations;
2
3  namespace Dws.Note_one.Api.Resources
4  {
5      public class SaveCategoryResource
6      {
7          [Required]
8          [MaxLength(30)]
9          public string Name { get; set; }
10     }
11 }
```

Observe os atributos *Required* e *MaxLength* aplicados sobre a propriedade *Name*. Esses atributos são chamados de anotações de dados (ou *data annotations*). O *pipeline* do ASP.NET Core usa esses metadados para validar solicitações e respostas. Como os nomes sugerem, o atributo nome de *Category* é obrigatório e pode ter no máximo 30 caracteres.

Agora vamos definir a forma da nova rota da API. Adicione o seguinte código ao controlador de categorias:



The screenshot shows the VS Code interface with the Explorer sidebar on the left. In the Controllers folder, there are files: CategoryController.cs, SaveCategoryResource.cs, and Program.cs. The CategoryController.cs file is selected and open in the editor. It contains a CategoriesController class with a GetAllAsync method annotated with [HttpGet] and a PostAsync method annotated with [HttpPost].

```
10  namespace Dws.Note_one.Api.Controllers
11  {
12      [Route("/api/[controller]")]
13      public class CategoriesController : Controller
14      {
15          private readonly ICategoryService _categoryService;
16          private readonly IMapper _mapper;
17
18          public CategoriesController(ICategoryService categoryService, IMapper mapper)
19          {
20              _categoryService = categoryService;
21              _mapper = mapper;
22          }
23
24          [HttpGet]
25          public async Task<IEnumerable<CategoryResource>> GetAllAsync()
26          {
27              return await _categoryService.GetAll();
28          }
29
30          [HttpPost]
31          public async Task<IActionResult> PostAsync([FromBody] SaveCategoryResource resource)
32          {
33              if (resource == null)
34              {
35                  return null;
36              }
37
38              var category = _categoryService.Create(resource);
39              var mappedCategory = _mapper.Map<Category>(category);
40
41              return Ok(mappedCategory);
42          }
43      }
44  }
```

Usando a anotação *[HttpPost]*, dizemos ao ASP.NET Core que este método responde pelo verbo POST da rota de Category. Observe o tipo de resposta deste método, **Task<IActionResult>**. Os métodos presentes nas classes do controlador são chamados de ações e têm essa assinatura porque podemos retornar mais de um resultado possível depois que o aplicativo executa a ação.

Nesse caso, se o nome da categoria for inválido ou se algo der errado, temos que retornar uma resposta de código 400 (solicitação incorreta), contendo geralmente uma mensagem de erro que os aplicativos cliente podem usar para tratar o problema, ou podemos ter um 200 resposta (sucesso) com dados se tudo correr bem.

Existem muitos tipos de tipos de ação (ou **ActionResult**) que você pode usar como resposta, mas geralmente podemos usar essa interface e o ASP.NET Core usará uma classe padrão para isso.

O atributo *FromBody* informa ao ASP.NET Core para analisar os dados do corpo da requisição em nossa nova classe de recurso. Isso significa que quando um JSON contendo o nome da categoria é enviado ao nosso aplicativo, o *framework* o analisa automaticamente para a nossa nova classe.

Agora, vamos implementar nossa lógica de rota. Temos que seguir alguns passos para criar com sucesso uma nova **Category**:

1. Primeiro, temos que validar a requisição recebida. Se a requisição da rota POST for inválida, devemos retornar uma resposta de requisição incorreta contendo as mensagens de erro;
2. Então, se a solicitação for válida, temos que mapear nosso novo *Resource* para nossa classe de modelo de **Category** usando *AutoMapper*;
3. Precisamos também ativar o *Service* de **Category**, dizendo para salvar nossa nova instância de **Category**. Se a lógica de salvamento for executada sem problemas, ele deve retornar uma resposta contendo nossos novos dados de categoria. Caso contrário, deve nos dar uma indicação de que o processo falhou e uma mensagem de erro potencial;
4. Finalmente, se houver um erro, retornamos uma mensagem de requisição incorreta. Caso contrário, mapeamos nosso novo modelo de **Category** para um **CategoryResource** e retornamos uma resposta de sucesso ao cliente, contendo os novos dados de **Category**.

Parece complicado, mas é realmente fácil implementar essa lógica usando a arquitetura de serviço que estruturamos para nossa API. Vamos começar validando a solicitação recebida.

#### 4.2.9 Validando uma requisição com body usando Model State

Os *Controllers* do ASP.NET Core têm uma propriedade chamada *ModelState*. Esta propriedade é preenchida durante a execução da requisição antes de chegar à execução da

ação. É uma instância de *ModelStateDictionary*, uma classe que contém informações como se a solicitação é válida e mensagens de erro de validação em potencial.

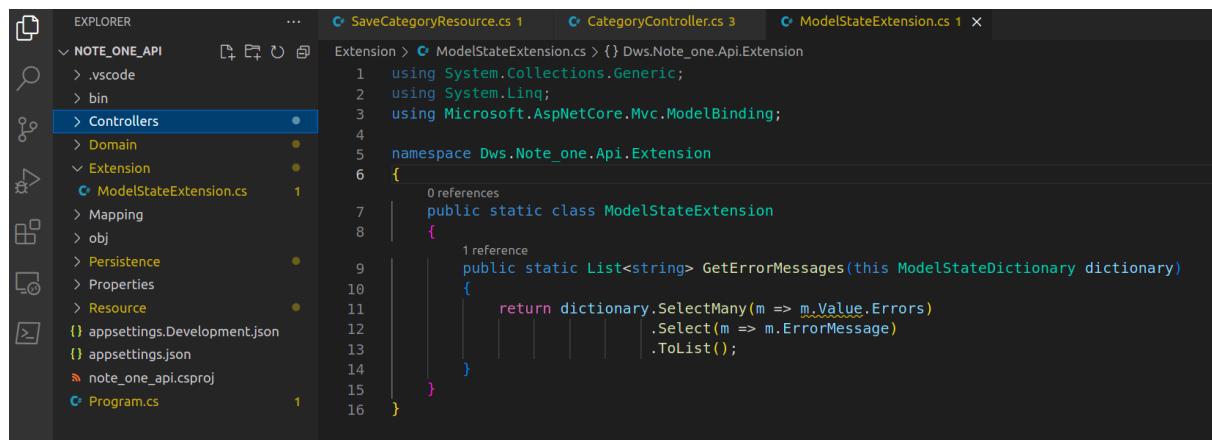
Altere o código do método do verbo POST da seguinte forma:

```
[HttpPost]
0 references
public async Task<IActionResult> PostAsync([FromBody] SaveCategoryResource resource)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState.GetErrorMessages());
    else
        return null;
}
```

O código verifica se o estado do modelo (neste caso, os dados enviados no corpo da solicitação) é inválido, verificando nossas anotações de dados. Se não for, a API retorna uma solicitação incorreta (com código de status 400) e as mensagens de erro padrão que nossos metadados de anotações forneceram.

O método *ModelState.GetErrorMessages()* ainda não foi implementado. É um método de extensão (um método que estende a funcionalidade de uma classe ou interface já existente) que será implementado para converter os erros de validação em strings simples para retornar ao cliente.

Adicione uma nova pasta *Extensions* na raiz de nossa API e, em seguida, adicione uma nova classe **ModelStateExtension**:



Todos os métodos de extensão em **ModelStateExtension** devem ser estáticos, assim como nas classes onde são declarados. Lembre-se de quando você estudou o paradigma de orientação a objetos e leu sobre membros estáticos. Membros estáticos não manipulam dados de instância específicos e são carregados apenas uma vez, quando o aplicativo é iniciado.

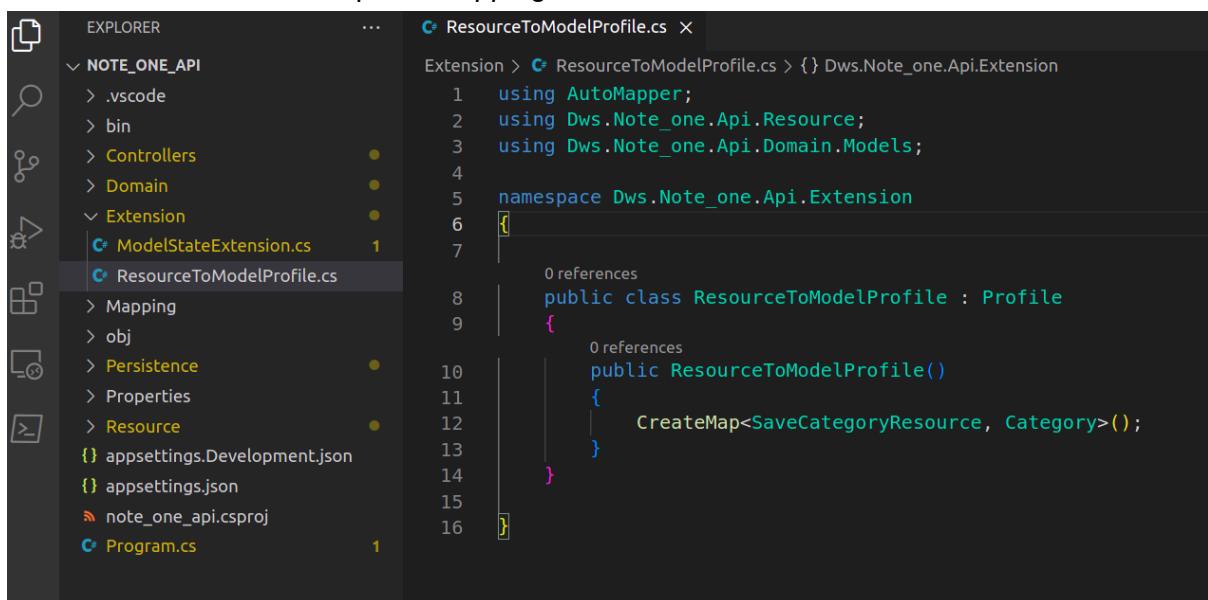
A palavra-chave **this** na frente da declaração do parâmetro **ModelStateDictionary** diz ao compilador C# para tratá-la como um método de extensão. O resultado é que podemos chamá-lo como um método normal desta classe, pois incluímos a respectiva diretiva **using** onde queremos usar a extensão.

A extensão usa consultas LINQ, um recurso muito útil do .NET que nos permite consultar e transformar dados usando expressões encadeadas. As expressões aqui transformam os métodos de erro de validação em uma lista de strings contendo as mensagens de erro.

Importe o namespace `<nome da sua aplicação>.Extensions` para o CategoryController de ir para a próxima etapa. Vamos continuar implementando nossa lógica de rota POST mapeando nosso **SaveCategoryResource** para uma classe de modelo de **Category**.

#### 4.2.10 Mapeando o SaveCategoryResource

Já definimos um perfil de mapeamento para transformar *Models* em *Resources*. Agora precisamos de um novo perfil que faça o inverso. Adicione uma nova classe `ResourceToModelProfile` à pasta `Mapping`:

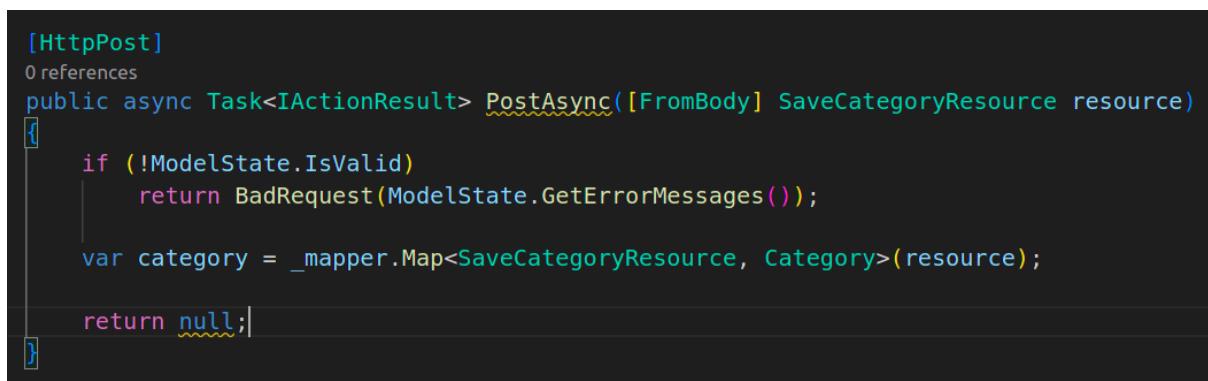


The screenshot shows the Visual Studio interface. On the left, the Solution Explorer displays the project structure with files like `ModelStateExtension.cs`, `ResourceToModelProfile.cs`, and `Program.cs`. On the right, the code editor shows the `ResourceToModelProfile.cs` file:

```
using AutoMapper;
using Dws.Note_one.Api.Resource;
using Dws.Note_one.Api.Domain.Models;

namespace Dws.Note_one.Api.Extension
{
    public class ResourceToModelProfile : Profile
    {
        public ResourceToModelProfile()
        {
            CreateMap<SaveCategoryResource, Category>();
        }
    }
}
```

Nada de novo aqui. Graças à magia da injecção de dependência, o *AutoMapper* registrará automaticamente este perfil quando o aplicativo for iniciado, e não temos que mudar nenhum outro lugar para usá-lo. Agora, em `PostAsync` de `CategoryController`, podemos mapear nosso novo recurso para a respectiva classe de modelo:



```
[HttpPost]
public async Task<IActionResult> PostAsync([FromBody] SaveCategoryResource resource)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState.GetErrorMessages());

    var category = _mapper.Map<SaveCategoryResource, Category>(resource);

    return null;
}
```

## 4.2.11 Aplicando o Padrão de Projeto Request-Response para registrar novas instâncias de Category

Agora temos que implementar uma das lógicas mais importantes: salvar uma nova instância **Category**. Essa operação será desenvolvida em **CategoryService** conforme a nossa filosofia de desenvolvimento.

A lógica de salvamento pode falhar devido a problemas na conexão com o banco de dados, ou talvez porque alguma regra de negócio interna que avalie como inválido os dados submetidos pela aplicação cliente.

Se algo der errado, não podemos simplesmente lançar um erro, porque isso poderia interromper a API e o aplicativo cliente não saberia como lidar com o problema. Além disso, possivelmente precisaríamos ter algum mecanismo de *logging* para registrar o erro.

O contrato do método de salvamento, ou seja, a assinatura do método e o tipo de resposta, precisa nos indicar se o processo foi executado corretamente. Se o processo correr bem, receberemos os dados da categoria. Caso contrário, devemos receber, pelo menos, uma mensagem de erro informando por que o processo falhou.

Sendo assim, podemos implementar essa operação aplicando o Padrão de Projeto *Request-Response*. Esse Padrão encapsula nossos parâmetros de solicitação e resposta em classes como uma forma de encapsular informações que nossos *Services* usarão para processar alguma tarefa e retornar informações para a classe que está usando o respectivo *Service*.

Esse padrão nos dá algumas vantagens, como:

- Se precisarmos alterar nosso serviço para receber mais parâmetros, não precisamos quebrar sua assinatura;
- Podemos definir um contrato padrão para nossa solicitação e / ou respostas;
- Podemos lidar com a lógica de negócios e possíveis falhas sem interromper o processo de aplicação e não precisaremos usar toneladas de blocos *try-catch*.

Vamos criar um tipo de resposta padrão para nossos métodos de serviços que lidam com alterações de dados. Para cada solicitação deste tipo, queremos saber se a solicitação é executada sem problemas. Se falhar, queremos retornar uma mensagem de erro ao cliente.

Na pasta *Domain*, dentro de *Services*, adicione um novo diretório chamado *Communication*. Adicione uma nova classe chamada **BaseResponse**.

The screenshot shows the VS Code interface. The Explorer pane on the left displays the project structure of 'NOTE\_ONE\_API' with files like .vscode, bin, Controllers, Domain, Models, Repositories, Services, Communication, BaseResponse.cs, IServices, CategoryService.cs, Extension, Mapping, obj, Persistence, Properties, Resource, appsettings.Development.json, appsettings.json, note\_one\_api.csproj, and Program.cs. The Editor pane on the right shows the code for 'BaseResponse.cs'.

```
1  using System.Collections.Generic;
2  using System.Threading.Tasks;
3  using Dws.Note_one.Api.Domain.Models;
4  using Dws.Note_one.Api.Domain.Services.IServices;
5  using Dws.Note_one.Api.Domain.Repositories.IRepositories;
6
7  namespace Dws.Note_one.Api.Domain.Services.Communication
8  {
9      public abstract class BaseResponse
10     {
11         public bool Success { get; protected set; }
12         public string Message { get; protected set; }
13
14         public BaseResponse(bool success, string message)
15         {
16             Success = success;
17             Message = message;
18         }
19     }
20 }
21 }
```

**BaseResponse** é uma classe abstrata que nossos tipos de resposta herdarão. A abstração define uma propriedade *Success*, que dirá se as solicitações foram concluídas com sucesso, e uma propriedade *Message*, que terá a mensagem de erro se algo falhar.

Observe que essas propriedades são necessárias e apenas as super-classes podem definir esses dados porque as subclasses precisam passar essas informações por meio do método construtor. Apesar de recorrermos ao *subclasseamento* aqui, essa prática de vem sendo contestada desde os anos 2000.

Se você perceber que um serviço ou aplicativo crescerá e mudará com frequência, evite abusar de *subclasseamento*. Não é uma boa prática definir subclasses, pois esta prática pode levar ao acoplamento do seu código, dificultando assim futuras modificações e manutenções. Prefira usar composição em vez de herança.

Dando continuidade a implementação, adicione uma nova classe chamada **SaveCategoryResponse** em *Communication*:

```
SaveCategoryResponse.cs 1
Domain > Services > Communication > SaveCategoryResponse.cs > ...
1  using Dws.Note_one.Api.Domain.Models;
2
3  namespace Dws.Note_one.Api.Domain.Services.Communication
4  {
5
6      2 references
7      public class SaveCategoryResponse : BaseResponse
8      {
9          1 reference
10         public Category Category { get; private set; }
11
12         2 references
13         private SaveCategoryResponse(bool success, string message, Category category) : base(success, message)
14         {
15             Category = category;
16
17             /// <summary>
18             /// Creates a success response.
19             /// </summary>
20             /// <param name="category">Saved category.</param>
21             /// <returns>Response.</returns>
22             0 references
23             public SaveCategoryResponse(Category category) : this(true, string.Empty, category)
24             {
25
26                 /// <summary>
27                 /// Creates an error response.
28                 /// </summary>
29                 /// <param name="message">Error message.</param>
30                 /// <returns>Response.</returns>
31                 0 references
32             public SaveCategoryResponse(string message) : this(false, message, null)
33             {
34
35         }
36     }
37 }
```

Observe que **SaveCategoryResponse** possui três construtores diferentes:

- Um privado, que vai passar os parâmetros de sucesso e mensagem para a classe base, e também define a propriedade Category;
- Um construtor que recebe apenas a instância de Category como parâmetro. Este criará uma resposta bem-sucedida, chamando o construtor privado para definir as respectivas propriedades;
- Um terceiro construtor que especifica apenas a mensagem. Este será usado para criar uma resposta de falha.
- Como o C# oferece suporte a vários construtores, simplificamos a criação da resposta sem definir um método diferente para lidar com isso, apenas usando diferentes construtores.

Agora podemos alterar nossa **ICategoryService** para adicionar o novo contrato de método de salvamento:

```

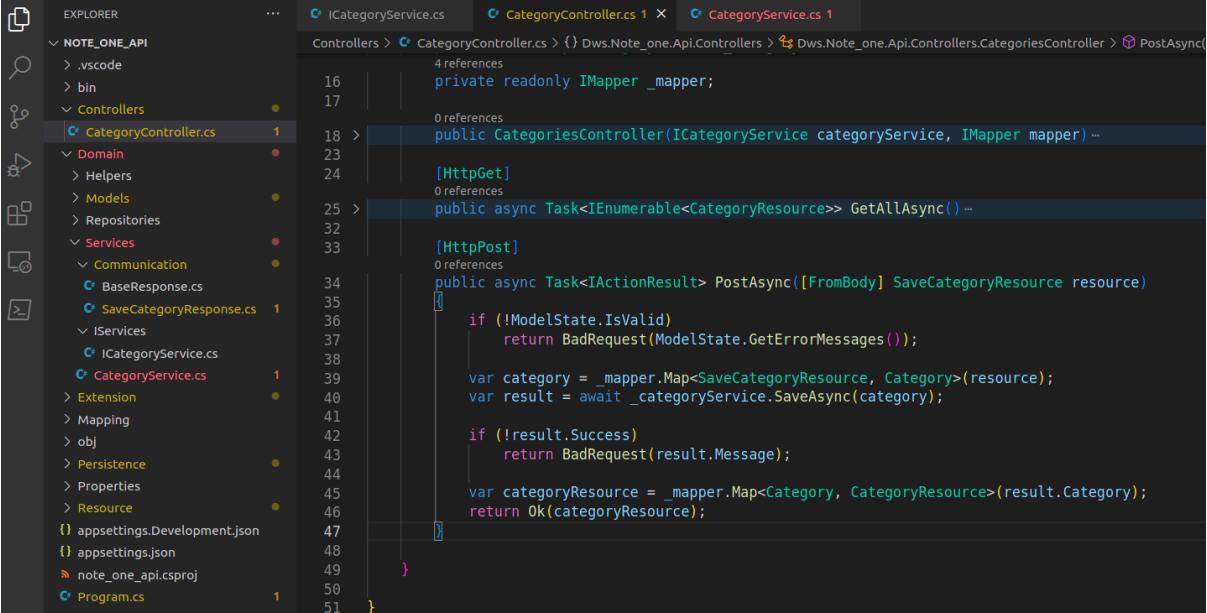
Domain > Services > IServices > ICategoriesService.cs > ...
1  using Dws.Note_one.Api.Domain.Models;
2  using Dws.Note_one.Api.Domain.Services.Communication;
3
4  namespace Dws.Note_one.Api.Domain.Services.IServices
5  {
6      4 references
7      public interface ICategoriesService
8      {
9          1 reference
10         Task<IEnumerable<Category>> ListAsync();
11         0 references
12         Task<SaveCategoryResponse> SaveAsync(Category category);
13     }
14 }

```

Vamos simplesmente passar uma categoria para este método e ele vai lidar com toda a lógica necessária para salvar os dados do modelo, orquestrando repositórios e outros serviços necessários para fazer isso. Observe que não estamos criando uma classe de solicitação específica aqui, pois não precisamos de nenhum outro parâmetro para realizar esta tarefa. Existe um conceito em programação chamado KISS - abreviação de *Keep it Simple, Stupid*. Basicamente, ele diz que você deve manter seu aplicativo o mais simples possível.

Lembre-se disso ao projetar seus aplicativos: aplique apenas o que você precisa para resolver um problema. Não exagere na engenharia de seu aplicativo. Procure usar pelo menos os princípios SOLID na modelagem da aplicação e conceitos de *Domain-Driven Design* como abordagem para guiar o seu trabalho.

Voltando para a implementação, podemos terminar nossa lógica de implementação para rota POST:



```

ICategoriesService.cs CategoryController.cs 1 × CategoryService.cs 1
Controllers > CategoryController.cs > () Dws.Note_one.Api.Controllers > Dws.Note_one.Api.Controllers.CategoriesController > PostAsync()
16    private readonly IMapper _mapper;
17
18    0 references
19    public CategoriesController(ICategoriesService categoryService, IMapper mapper) ...
20
21    [HttpGet]
22    0 references
23    public async Task<IEnumerable<CategoryResource>> GetAllAsync() ...
24
25    [HttpPost]
26    0 references
27    public async Task<IActionResult> PostAsync([FromBody] SaveCategoryResource resource)
28    {
29        if (!ModelState.IsValid)
30            return BadRequest(ModelState.GetErrorMessages());
31
32        var category = _mapper.Map<SaveCategoryResource, Category>(resource);
33        var result = await _categoryService.SaveAsync(category);
34
35        if (!result.Success)
36            return BadRequest(result.Message);
37
38        var categoryResource = _mapper.Map<Category, CategoryResource>(result.Category);
39        return Ok(categoryResource);
40    }
41
42    0 references
43    0 references
44    0 references
45    0 references
46    0 references
47    0 references
48    0 references
49    0 references
50    0 references
51 }

```

Depois de validar os dados da solicitação e mapear o *Resource* para nosso modelo (ou *Model*), passamos para o nosso *Service* para persistir os dados. Se algo falhar, a API retorna uma solicitação inválida. Do contrário, a API mapeia a nova categoria (agora incluindo dados como o novo ID) para o nosso **CategoryResource** criado anteriormente e o envia para a aplicação cliente. A seguir, implementaremos a lógica real para o **CategoryService**.

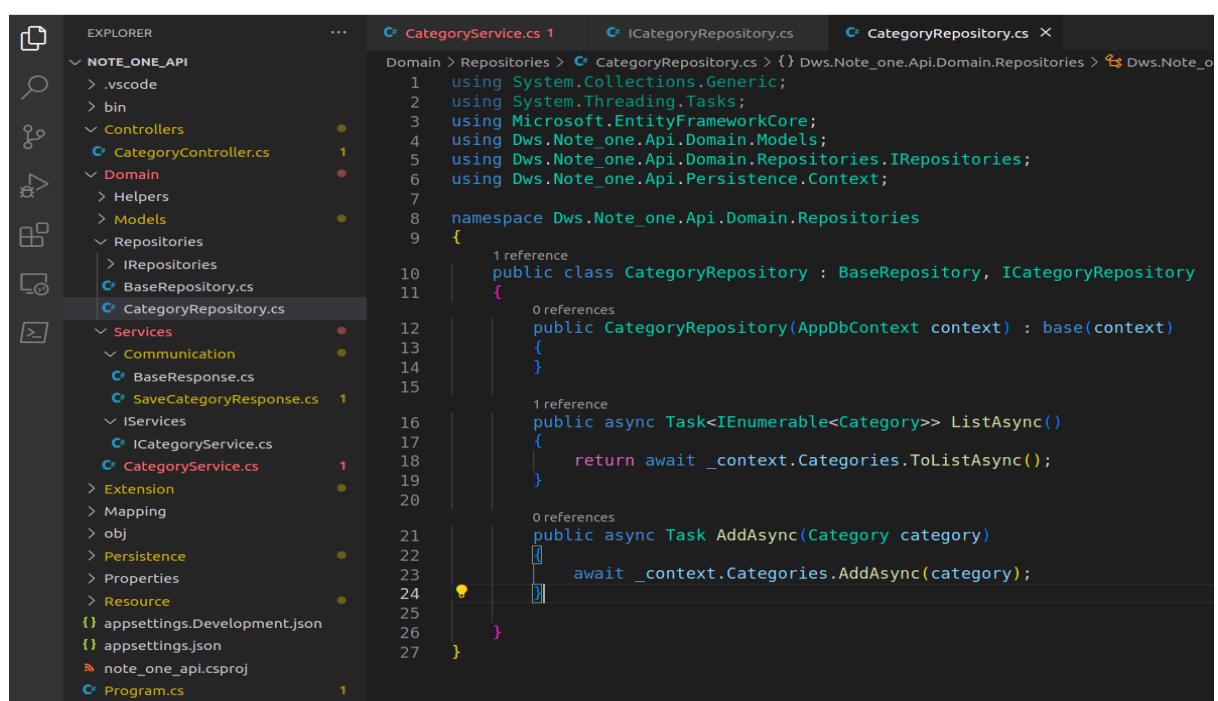
## 4.2.12 Lógica de Banco de Dados e o Padrão de Projeto Unity of Work

Uma vez que vamos persistir os dados no banco de dados, precisamos de um novo método em nosso repositório. Adicione um novo método *AddAsync* à **Interface ICategoryRepository**:

```

1  using System.Collections.Generic;
2  using System.Threading.Tasks;
3  using Dws.Note_one.Api.Domain.Models;
4
5  namespace Dws.Note_one.Api.Domain.Repositories.IRepositories
6  {
7      4 references
8      public interface ICategoryRepository
9      {
10          1 reference
11          Task<IEnumerable<Category>> ListAsync();
12          0 references
13          Task AddAsync(Category category);
14      }
15 }
```

Agora, vamos implementar este método em nossa classe concreta:



```

1  Domain > Repositories > CategoryRepository.cs > {} Dws.Note_one.Api.Domain.Repositories > Dws.Note_one.Api.Domain.Repositories.IRepositories
2  1 using System.Collections.Generic;
3  2 using System.Threading.Tasks;
4  3 using Microsoft.EntityFrameworkCore;
5  4 using Dws.Note_one.Api.Domain.Models;
6  5 using Dws.Note_one.Api.Domain.Repositories.IRepositories;
7  6 using Dws.Note_one.Api.Persistence.Context;
8
9  namespace Dws.Note_one.Api.Domain.Repositories
10 {
11     1 reference
12     public class CategoryRepository : BaseRepository, ICategoryRepository
13     {
14         0 references
15         public CategoryRepository(DbContext context) : base(context)
16         {
17         }
18
19         1 reference
20         public async Task<IEnumerable<Category>> ListAsync()
21         {
22             return await _context.Categories.ToListAsync();
23         }
24
25         0 references
26         public async Task AddAsync(Category category)
27         {
28             await _context.Categories.AddAsync(category);
29         }
30     }
31 }
```

Este novo método implica simplesmente em adicionar uma nova categoria ao nosso conjunto de dados. Quando adicionamos uma classe a um *DBSet* <>, o EF Core começa a rastrear todas as mudanças que acontecem em nosso *Model* e usa esses dados no estado atual para gerar consultas que irão inserir, atualizar ou excluir modelos. A implementação atual simplesmente adiciona o *Model* ao nosso conjunto, mas nossos dados ainda não serão salvos.

Existe um método chamado *SaveChanges* presente na classe de contexto que devemos chamar para realmente executar o comando SQL de inserção no banco de dados. Não chamei *SaveChanges* em *AddAsync* aqui porque uma classe *Repository* não deve persistir dados. Uma classe *Repository* deve apenas tratar uma coleção de objetos na memória.

Este assunto é muito controverso mesmo entre desenvolvedores .NET experientes, mas, eis a motivação para não chamar *SaveChanges* em classes *Repository*: podemos pensar em um *Repository* conceitualmente como qualquer outra coleção presente no framework .NET. Ao lidar com uma coleção em .NET (e muitas outras linguagens de programação, como Javascript e Java), geralmente você pode:

- Adicionar novos itens a coleção;
- Pesquisar e filtrar itens;
- Remover um item da coleção;
- Substituir um determinado item ou atualizá-lo.

Agora pense em uma lista do mundo real. Imagine que você está escrevendo uma lista de compras para comprar coisas em um supermercado (que coincidência, não?).

Na lista, você escreve todas as frutas que precisa comprar. Você pode adicionar frutas a esta lista, remover uma fruta se desistir de comprá-la ou pode substituir o nome de uma fruta. Mas você não pode guardar (ou, na lógica da aplicação, **salvar**) frutas físicas na lista. Não faz sentido dizer uma coisa dessas em linguagem natural.

Se quiser “guardar” as frutas da lista (no caso, comprar todas as frutas), você paga e o supermercado processa os dados do estoque para verificar se tem que comprar mais frutas de um fornecedor ou não.

A mesma lógica pode ser aplicada durante a programação. Um *Repository* não deve salvar, atualizar ou excluir dados. Em vez disso, eles devem delegá-lo a uma classe diferente para lidar com essa lógica. Há outro problema ao salvar dados diretamente em um repositório: você não pode usar **transações**.

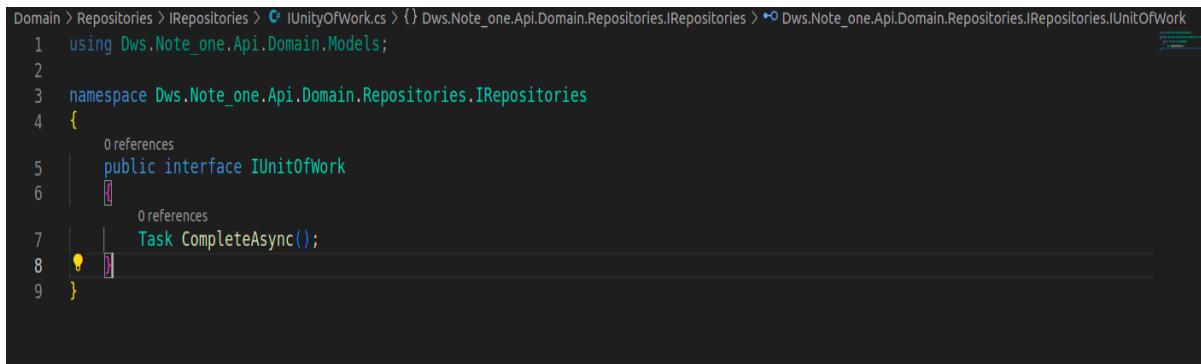
Imagine que nosso aplicativo possui um mecanismo de registro que armazena algum nome de usuário e a ação executada sempre que uma alteração é feita nos dados da API. Agora imagine que, por algum motivo, você tenha uma chamada para um serviço que atualiza o nome de usuário.

Você concorda que, para alterar o nome de usuário em uma tabela de usuários fictícios, primeiro é necessário atualizar todos os logs para informar corretamente quem realizou a operação, certo?

Agora imagine que implementamos o método de atualização para usuários e logs em diferentes classes *Repository*, e ambos chamam *SaveChanges*. O que acontece se um desses métodos falhar no meio do processo de atualização? Você vai acabar com inconsistência de dados.

Devemos salvar nossas alterações no banco de dados somente depois que tudo terminar. Para fazer isso, temos que usar uma **transação**, que é basicamente um recurso que a maioria dos bancos de dados implementa para salvar dados somente após o término de uma operação complexa.

Um Padrão de Projeto comum para lidar com esse problema é o *Unity of Work*. Na nossa API, a aplicação deste Padrão consiste em uma classe que recebe nossa instância **AppDbContext** como uma dependência e expõe métodos para iniciar, completar ou abortar **transações**. Para tal, adicione uma nova **interface** dentro da pasta *IRepositories* da camada de *Domain* chamada *IUnitOfWork*:



```
Domain > Repositories > IRepositories > IUnitOfWork.cs > {} Dws.Note_one.Api.Domain.Repositories > Dws.Note_one.Api.Domain.Repositories.IRepositories.IUnitOfWork
1 using Dws.Note_one.Api.Domain.Models;
2
3 namespace Dws.Note_one.Api.Domain.Repositories.IRepositories
4 {
5     0 references
6     public interface IUnitOfWork
7     {
8         0 references
9             Task CompleteAsync();
10    }
11 }
```

Esta **interface** dispõe apenas de um método que completará de forma assíncrona as operações de gerenciamento de dados. Vamos adicionar a implementação real agora. Adicione uma nova classe chamada *UnitOfWork* na pasta *Repositories* da camada de *Persistence*:

```

Domain > Repositories > UnitOfWork.cs > () Dws.Note_one.Api.Persistence.Repositories > Dws.Note_one.Api.Persistence.Repositories.UnitOfWork
1  using Dws.Note_one.Api.Domain.Repositories.IRepositories;
2  using Dws.Note_one.Api.Persistence.Context;
3
4  namespace Dws.Note_one.Api.Persistence.Repositories
5  {
6      0 references
7      public class UnitOfWork : IUnitOfWork
8      [
9          2 references
10         private readonly AppDbContext _context;
11
12         0 references
13         public UnitOfWork(AppDbContext context)
14         {
15             _context = context;
16         }
17
18         0 references
19         public async Task CompleteAsync()
20         {
21             await _context.SaveChangesAsync();
22         }
23     ]
24 }

```

A classe **UnityOfWork** fornece uma implementação simples e limpa que só salvará todas as alterações no banco de dados depois que você terminar de modificá-lo usando seus repositórios. Se você pesquisar implementações do Padrão *Unity of Work*, encontrará outras mais complexas implementando operações de reversão (ou **rollbacks**). Como o EF Core já implementa o padrão de *Repository* e a *Unity of Work* nas suas próprias operações, não precisamos nos preocupar com um método de reversão.

Agora que você sabe o que é uma *Unity of Work* e por que usá-la com classes *Repository*, vamos implementar a lógica do *Service* concreto:

```

using Dws.Note_one.Api.Domain.Models;
using Dws.Note_one.Api.Domain.Services.IServices;
using Dws.Note_one.Api.Domain.Repositories.IRepositories;
using Dws.Note_one.Api.Domain.Services.Communication;

namespace Dws.Note_one.Api.Domain.Services
{
    public class CategoryService : ICategoryService
    {
        private readonly ICategoryRepository _categoryRepository;
        private readonly IUnitOfWork _unitOfWork;

        public CategoryService(ICategoryRepository categoryRepository,
IUnitOfWork unitOfWork)
        {
            _categoryRepository = categoryRepository;
            _unitOfWork = unitOfWork;
        }
    }
}

```

```

        public async Task<IEnumerable<Category>> ListAsync()
    {
        return await _categoryRepository.ListAsync();
    }

        public async Task<SaveCategoryResponse> SaveAsync(Category
category)
    {
        try
        {
            await _categoryRepository.AddAsync(category);
            await _unitOfWork.CompleteAsync();

            return new SaveCategoryResponse(category);
        }
        catch (Exception ex)
        {
            // Do some logging stuff
            return new SaveCategoryResponse($"An error occurred when
saving the category: {ex.Message}");
        }
    }
}

```

Graças à nossa arquitetura desacoplada, podemos simplesmente passar uma instância de **UnitOfWork** como uma dependência para o nosso *CategoryService*. A lógica de negócios é muito simples. Primeiro, tentamos adicionar a nova categoria ao banco de dados e, em seguida, a API tenta salvá-la, envolvendo tudo dentro de um bloco try-catch. Se algo falhar, a API chama algum serviço de registro fictício e retorna uma resposta indicando falha. Se o processo terminar sem problemas, o aplicativo retorna uma resposta de sucesso, enviando nossos dados de categoria. Simples, certo?

A última etapa antes de testar nossa API é vincular a interface da unidade de trabalho à sua respectiva classe. Adicione esta nova linha ao método builder.*Services* da classe *Program*:

```
builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
```

#### 4.2.13 Testando a rota POST com um software API Client

Inicie nosso aplicativo novamente usando o **dotnet run**. Não podemos usar o browser para testar uma rota POST. Para tal, precisamos de uma aplicação API Client. No mercado

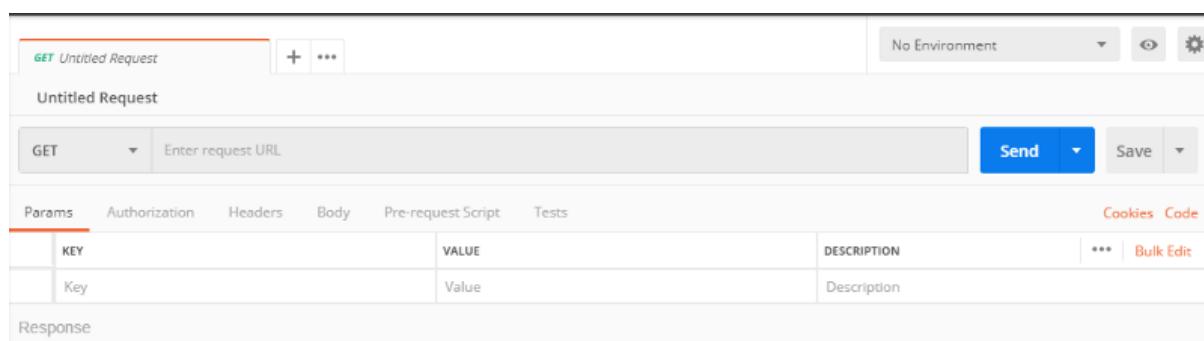
destacam-se: Apache JMeter; Insomnia; Postman. Neste tutorial, utilizamos o Postman. O Postman é uma ferramenta muito útil para testar APIs RESTful. O link para download do Postman é este:

<https://www.postman.com/downloads/>

Neste vídeo, você aprenderá a manusear o Postman:

[https://www.youtube.com/watch?v=u9iBjM-x5Jc&ab\\_channel=PretzelCode](https://www.youtube.com/watch?v=u9iBjM-x5Jc&ab_channel=PretzelCode)

Agora, voltando a nossa implementação, abra o Postman e feche as mensagens de introdução. Você verá uma tela como esta:

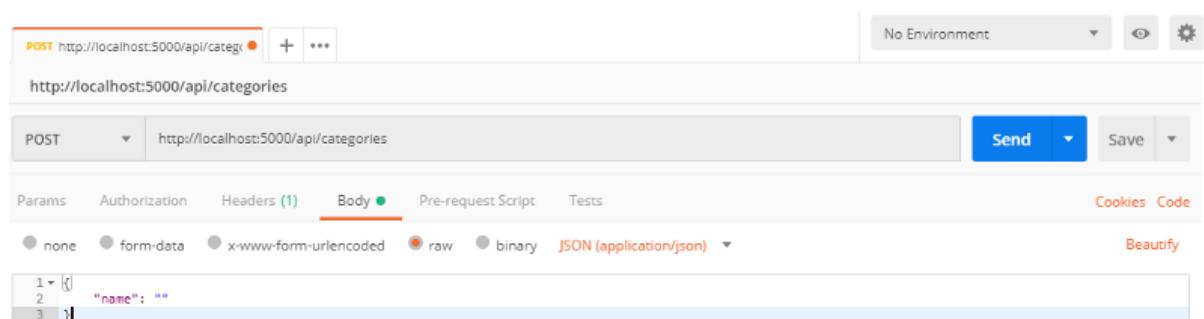


The screenshot shows the Postman application window. At the top, there's a header bar with the title 'GET Untitled Request' and a '+' button. To the right of the title are buttons for 'No Environment', 'Send', 'Save', and settings. Below the header is a main panel titled 'Untitled Request'. It contains a 'Method' dropdown set to 'GET' and a 'Enter request URL' input field which is currently empty. To the right of these fields are 'Send' and 'Save' buttons. Below the URL input is a table with tabs for 'Params', 'Authorization', 'Headers', 'Body', 'Pre-request Script', and 'Tests'. The 'Body' tab is selected and has a single row with columns for 'KEY', 'VALUE', 'DESCRIPTION', and 'Bulk Edit'. Under 'KEY' is 'Key', under 'VALUE' is 'Value', and under 'DESCRIPTION' is 'Description'. At the bottom of the main panel is a 'Response' section.

Altere o GET selecionado por padrão na caixa de seleção para POST. Digite o endereço da API no campo **Enter request URL**. Temos que fornecer os dados do corpo da solicitação para enviar à nossa API. Clique no item de menu *Body* e altere a opção exibida abaixo para *raw*.

O Postman mostrará uma opção de *Text* à direita. Mude para *JSON(application/json)* e cole os seguintes dados JSON abaixo:

```
{  
  "name": ""  
}
```



The screenshot shows the Postman application window again. The title bar now says 'POST http://localhost:5000/api/categories'. The 'Enter request URL' field contains 'http://localhost:5000/api/categories'. The 'Body' tab is selected, showing a JSON editor with the following content:  

```
1 [{}  
2   "name": ""  
3 ]
```

Como você pode ver, enviaremos uma string de nome vazia para nossa rota POST. Clique no botão Enviar. Você receberá uma saída como esta:

The screenshot shows the Postman interface with a request to `http://localhost:5000/api/categories`. The response status is `400 Bad Request`, time `429 ms`, and size `188 B`. The response body is a JSON object with one item: `{"name": "The Name field is required."}`.

Você se lembra da lógica de validação que criamos para o endpoint? Essa saída é a prova de que funciona!

Observe também o código de status 400 exibido à direita. O resultado BadRequest adiciona automaticamente esse código de status à resposta. Agora vamos alterar os dados JSON para um válido para ver a nova resposta:

The screenshot shows the Postman interface with a request to `http://localhost:5000/api/categories`. The method is `POST` and the URL is `http://localhost:5000/api/categories`. The `Body` tab is selected, showing a JSON payload: `{"name": "Bakery"}`. The response status is `200 OK`, time `100 ms`, and size `172 B`. The response body is a JSON object with two items: `{ "id": 1, "name": "Bakery" }`.

**OBS:** nesta Figura e nas demais que ilustrarem as telas do Postman, na minha implementação, uso <http://localhost:5000/api/category> e não 'categories', pois o nome do meu controller é CategoryController e não CategoriesController.

A API criou corretamente nosso novo *Resource*. Até agora, nossa API pode listar e criar categorias. Você aprendeu muitas coisas sobre a linguagem C #, a estrutura ASP.NET Core e também princípios e abordagens de Padrões de Projeto comuns para estruturar suas APIs. Vamos continuar nossa API de categorias criando a rota para atualizar as categorias.

#### 4.2.14 Implementando uma rota PUT de atualização de dados

Para atualizar as instâncias de Category, precisamos de uma rota HTTP PUT. A lógica que devemos codificar é muito semelhante à do POST:

- Primeiro, temos que validar a solicitação de entrada usando o ModelState;

- Se a solicitação for válida, a API deve mapear o recurso de entrada para uma classe de modelo usando AutoMapper;
- Em seguida, devemos ligar para o nosso serviço, informando-o sobre a atualização da categoria, fornecendo o respectivo Id da categoria e os dados atualizados;
- Se não houver categoria com o ID fornecido no banco de dados, retornamos uma solicitação incorreta. Poderíamos usar um resultado *NotFound* em vez disso, mas isso realmente não importa para esse escopo, já que fornecemos uma mensagem de erro aos aplicativos cliente;
- Se a lógica de salvamento for executada corretamente, o serviço deve retornar uma resposta contendo os dados atualizados da categoria. Caso contrário, deve nos dar uma indicação de que o processo falhou e uma mensagem indicando o motivo;
- Finalmente, se houver um erro, a API retornará uma solicitação incorreta. Caso contrário, ele mapeia o modelo de categoria atualizado para um recurso de categoria e retorna uma resposta de sucesso para o aplicativo cliente.

Vamos adicionar o novo método *PutAsync* à classe *CategoryController*:

```
[HttpPut("{id}")]
0 references
public async Task<IActionResult> PutAsync(int id, [FromBody] SaveCategoryResource resource)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState.GetErrorMessages());

    var category = _mapper.Map<SaveCategoryResource, Category>(resource);
    var result = await _categoryService.UpdateAsync(id, category);

    if (!result.Success)
        return BadRequest(result.Message);

    var categoryResource = _mapper.Map<Category, CategoryResource>(result.Category);
    return Ok(categoryResource);
}
```

Se você comparar com a lógica POST, perceberá que temos apenas uma diferença aqui: o atributo *HttpPut* especifica um parâmetro que a rota fornecida deve receber.

Chamaremos esse endpoint especificando o ID da categoria como o último fragmento de URL, como */api/category/1*. O pipeline do ASP.NET Core analisa esse fragmento para o parâmetro de mesmo nome. Agora temos que definir a assinatura do método *UpdateAsync* na interface *ICategoryService*:

```
using Dws.Note_one.Api.Domain.Models;
using Dws.Note_one.Api.Domain.Services.Communication;

namespace Dws.Note_one.Api.Domain.Services.IServices
{
```

```

public interface ICategoryService
{
    Task<IEnumerable<Category>> ListAsync();
    Task<SaveCategoryResponse> SaveAsync(Category category);

    Task<SaveCategoryResponse> UpdateAsync(int id, Category
category);

}
}

```

Para atualizar nossa *Category*, primeiro, precisamos retornar os dados atuais do banco de dados, se houver. Também precisamos atualizá-lo em nosso DBSet <>. Vamos adicionar dois novos contratos de método à nossa interface *ICategoryRepository*:

```

using Dws.Note_one.Api.Domain.Models;

namespace Dws.Note_one.Api.Domain.Repositories.IRepositories
{
    public interface ICategoryRepository
    {
        Task<IEnumerable<Category>> ListAsync();
        Task AddAsync(Category category);
        Task<Category> FindByIdAsync(int id);
        void Update(Category category);
    }
}

```

Definimos o método *FindByIdAsync*, que retornará de forma assíncrona uma categoria do banco de dados, e o método *Update*. Preste atenção que o método *Update* não é assíncrono, pois a API EF Core não requer um método assíncrono para atualizar modelos. Agora vamos implementar a lógica real na classe *CategoryRepository*:

```

public async Task<Category> FindByIdAsync(int id)
{
    return await _context.Categories.FindAsync(id);
}

public void Update(Category category)
{
    _context.Categories.Update(category);
}

```

Finalmente, podemos codificar a lógica do *CategoryService*:

```
public async Task<SaveCategoryResponse> UpdateAsync(int id,  
Category category)  
{  
  
    var existingCategory = await  
_categoryRepository.FindByIdAsync(id);  
  
    if (existingCategory == null)  
        return new SaveCategoryResponse("Category not found.");  
  
    existingCategory.Name = category.Name;  
  
    try  
    {  
        _categoryRepository.Update(existingCategory);  
        await _unitOfWork.CompleteAsync();  
  
        return new SaveCategoryResponse(existingCategory);  
    }  
    catch (Exception ex)  
    {  
        // Do some logging stuff  
        return new SaveCategoryResponse($"An error occurred when  
updating the category: {ex.Message}");  
    }  
}
```

A API tenta obter a categoria do banco de dados. Se o resultado for nulo, retornamos uma resposta informando que a categoria não existe. Se a categoria existe, precisamos definir seu novo nome.

A API, então, tenta salvar as alterações, como quando criamos uma nova categoria. Se o processo for concluído, o serviço retornará uma resposta de sucesso. Caso contrário, a lógica de registro é executada e o terminal recebe uma resposta contendo uma mensagem de erro.

Agora vamos testar. Primeiro, vamos adicionar uma nova categoria para ter um ID válido para usar. Poderíamos usar os identificadores das categorias que propagamos em nosso banco de dados, mas quero fazer isso desta forma para mostrar a você que nossa API vai atualizar o recurso correto.

Execute a API novamente e, usando o Postman, poste uma nova categoria no banco de dados:

The screenshot shows the Postman interface with a POST request to `http://localhost:5000/api/categories`. The Body tab is selected, showing a raw JSON payload: 

```
1 - {  
2   "name": "Bakery"  
3 }
```

. The response tab shows a status of 200 OK with a response body: 

```
1 - {  
2   "id": 1,  
3   "name": "Bakery"  
4 }
```

.

Tendo um ID válido em mãos, altere a opção POST para PUT na caixa de seleção e adicione o valor do ID no final da URL. Altere a propriedade `name` para um nome diferente e envie a solicitação para verificar o resultado:

The screenshot shows the Postman interface with a PUT request to `http://localhost:5000/api/categories/1`. The Body tab is selected, showing a raw JSON payload: 

```
1 - {  
2   "name": "Former Bakery"  
3 }
```

. The response tab shows a status of 200 OK with a response body: 

```
1 - {  
2   "id": 1,  
3   "name": "Former Bakery"  
4 }
```

.

Você pode enviar uma solicitação GET ao endpoint da API para garantir que editou corretamente o nome da categoria:

The screenshot shows the Postman interface with a GET request to `http://localhost:5000/api/categories`. The response tab shows a list of categories in JSON format: 

```
1 - [  
2   {  
3     "id": 100,  
4     "name": "Fruits and Vegetables"  
5   },  
6   {  
7     "id": 101,  
8     "name": "Dairy"  
9   },  
10  {  
11    "id": 1,  
12    "name": "Former Bakery"  
13  }  
14 ]
```

.

A última operação que temos que implementar para categorias é a exclusão de categorias. Vamos fazer isso criando um endpoint HTTP Delete.

#### 4.2.15 Implementando uma rota **DELETE** de remoção de dados

A lógica para excluir categorias é realmente fácil de implementar, pois a maioria dos métodos de que precisamos foi criada anteriormente. Estas são as etapas necessárias para que nossa rota funcione:

- A API precisa chamar nosso serviço, informando para excluir nossa categoria, fornecendo o respectivo Id;
- Se não houver categoria com o ID fornecido no banco de dados, o serviço deve retornar uma mensagem indicando isso;
- Se a lógica de exclusão for executada sem problemas, o serviço deve retornar uma resposta contendo nossos dados de categoria excluídos. Caso contrário, deve nos dar uma indicação de que o processo falhou e uma mensagem de erro potencial;
- Finalmente, se houver um erro, a API retornará uma solicitação incorreta. Caso contrário, a API mapeia a categoria atualizada para um recurso e retorna uma resposta de sucesso ao cliente.

Vamos começar adicionando a nova lógica da rota DELETE:

```
[HttpDelete("{id}")]
0 references
public async Task<IActionResult> DeleteAsync(int id)
{
    var result = await _categoryService.DeleteAsync(id);

    if (!result.Success)
        return BadRequest(result.Message);

    var categoryResource = _mapper.Map<Category, CategoryResource>(result.Category);
    return Ok(categoryResource);
}
```

O atributo `HttpDelete` também define um modelo de id. Antes de adicionar a assinatura `DeleteAsync` à nossa interface `ICategoryService`, precisamos fazer uma pequena refatoração. O novo método de serviço deve retornar uma resposta contendo os dados da categoria, da mesma forma que fizemos para os métodos `PostAsync` e `UpdateAsync`. Poderíamos reutilizar o `SaveCategoryResponse` para essa finalidade, mas não estamos salvando dados neste caso.

Para evitar a criação de uma nova classe com a mesma forma para atender a esse requisito, podemos simplesmente renomear nosso `SaveCategoryResponse` para `CategoryResponse`. Se estiver usando o Visual Studio Code, você pode abrir a classe `SaveCategoryResponse`, colocar o cursor do mouse acima do nome da classe e usar a opção `Change All Occurrences` para renomear a classe:

```

namespace Dws.Note_one.Api.Domain.Services.Communication
{
    16 references
    public class CategoryResponse : BaseResponse
    {
        4 references
        public Category Category { get; private set; }

        2 references
        private CategoryResponse(bool success, string message, Category category) : base(success, message)
        {
            Category = category;
        }

        /// <summary>
        /// Creates a success response.
        /// </summary>
        /// <param name="category">Saved category.</param>
        /// <returns>Response.</returns>
        3 references
        public CategoryResponse(Category category) : this(true, string.Empty, category)
        {}

        /// <summary>
        /// Creates an error response.
        /// </summary>
        /// <param name="message">Error message.</param>
        /// <returns>Response.</returns>
        5 references
        public CategoryResponse(string message) : this(false, message, null)
        {}
    }
}

```

Certifique-se de renomear o nome do arquivo também. Vamos adicionar a assinatura do método `DeleteAsync` à interface `ICategoryService`:

```

public interface ICategoryService
{
    Task<IEnumerable<Category>> ListAsync();
    Task<CategoryResponse> SaveAsync(Category category);

    Task<CategoryResponse> UpdateAsync(int id, Category category);

    Task<CategoryResponse> DeleteAsync(int id);
}

```

Antes de implementar a lógica de exclusão, precisamos de um novo método em nosso repositório. Adicione a assinatura do método `Remove` à interface `ICategoryRepository`:

```

public interface ICategoryRepository
{
    Task<IEnumerable<Category>> ListAsync();
    Task AddAsync(Category category);
    Task<Category> FindByIdAsync(int id);
}

```

```
    void Update(Category category);

    void Remove(Category category);

}
```

E agora adicione a implementação real na classe do repositório:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Dws.Note_one.Api.Domain.Models;
using Dws.Note_one.Api.Domain.Repositories.IRepositories;
using Dws.Note_one.Api.Persistence.Context;

namespace Dws.Note_one.Api.Domain.Repositories
{
    public class CategoryRepository : BaseRepository,
ICategoryRepository
    {

        public CategoryRepository(AppDbContext context) : base(context)
        {
        }

        public async Task<IEnumerable<Category>> ListAsync()
        {
            return await _context.Categories.ToListAsync();
        }

        public async Task AddAsync(Category category)
        {
            await _context.Categories.AddAsync(category);
        }

        public async Task<Category> FindByIdAsync(int id)
        {
            return await _context.Categories.FindAsync(id);
        }

        public void Update(Category category)
        {
            _context.Categories.Update(category);
        }
    }
}
```

```

        }

        public void Remove(Category category)
        {
            _context.Categories.Remove(category);
        }

    }
}

```

*EF Core* requer que a instância do nosso modelo seja passada para o método *Remove* para entender corretamente qual modelo estamos excluindo, em vez de simplesmente passar um *Id*. Finalmente, vamos implementar a lógica na classe *CategoryService*:

```

using Dws.Note_one.Api.Domain.Models;
using Dws.Note_one.Api.Domain.Services.IServices;
using Dws.Note_one.Api.Domain.Repositories.IRepositories;
using Dws.Note_one.Api.Domain.Services.Communication;

namespace Dws.Note_one.Api.Domain.Services
{
    public class CategoryService : ICategoryService
    {
        private readonly ICategoryRepository _categoryRepository;
        private readonly IUnitOfWork _unitOfWork;

        public CategoryService(ICategoryRepository categoryRepository,
IUnitOfWork unitOfWork)
        {
            _categoryRepository = categoryRepository;
            _unitOfWork = unitOfWork;
        }

        public async Task<IEnumerable<Category>> ListAsync()
        {
            return await _categoryRepository.ListAsync();
        }

        public async Task<CategoryResponse> SaveAsync(Category category)
        {
            try

```

```

    {
        await _categoryRepository.AddAsync(category);
        await _unitOfWork.CompleteAsync();

        return new CategoryResponse(category);
    }
    catch (Exception ex)
    {
        // Do some logging stuff
        return new CategoryResponse($"An error occurred when
saving the category: {ex.Message}");
    }
}

public async Task<CategoryResponse> UpdateAsync(int id, Category
category)
{
    var existingCategory = await
_categoryRepository.FindByIdAsync(id);

    if (existingCategory == null)
        return new CategoryResponse("Category not found.");

    existingCategory.Name = category.Name;

    try
    {
        _categoryRepository.Update(existingCategory);
        await _unitOfWork.CompleteAsync();

        return new CategoryResponse(existingCategory);
    }
    catch (Exception ex)
    {
        // Do some logging stuff
        return new CategoryResponse($"An error occurred when
updating the category: {ex.Message}");
    }
}

public async Task<CategoryResponse> DeleteAsync(int id)
{

```

```

        var existingCategory = await
    _categoryRepository.FindByIdAsync(id);

    if (existingCategory == null)
        return new CategoryResponse("Category not found.");

    try
    {
        _categoryRepository.Remove(existingCategory);
        await _unitOfWork.CompleteAsync();

        return new CategoryResponse(existingCategory);
    }
    catch (Exception ex)
    {
        // Do some logging stuff
        return new CategoryResponse($"An error occurred when
deleting the category: {ex.Message}");
    }
}

}

```

Não há nada de novo aqui. O serviço tenta encontrar a categoria por ID e então chama nosso repositório para deletar a categoria. Finalmente, a unidade de trabalho completa a transação executando a operação real no banco de dados.

Agora é hora de testar nosso novo endpoint. Execute o aplicativo novamente e envie uma solicitação DELETE usando o Postman da seguinte maneira:

The screenshot shows the Postman interface with the following details:

- Request URL:** http://localhost:5000/api/categories/101
- Method:** DELETE
- Headers:** Authorization, Headers, Body, Pre-request Script, Tests, Cookies, Code
- Params:** Key: Value, Description
- Body:** JSON (Pretty, Raw, Preview) - Shows the JSON response: {"id": 101, "name": "Dairy"}
- Response Status:** Status: 200 OK, Time: 501 ms, Size: 173 B

Terminamos a API de categorias e finalizamos aqui este módulo.