

DESENVOLVIMENTO .NET

# WEB API

(RESTFUL)

FLAVIO MORENI



5

**LISTA DE FIGURAS**

Figura 5.1 – Usuário utilizando o Internet Banking .....	7
Figura 5.2 – Composição de URL e URN .....	12
Figura 5.3 – Projeto ASP.NET Web .....	14
Figura 5.4 – <i>Template</i> Web API .....	15
Figura 5.5 – Estrutura projeto Asp.NET WebAPI .....	16
Figura 5.6 – Diagrama de classe produto e categoria.....	17
Figura 5.7 – Adicionar <i>Controller</i> .....	22
Figura 5.8 – Selecionar o <i>Scaffold</i> do Controller .....	22
Figura 5.9 – Detalhes de um <i>Controller</i> .....	23
Figura 5.10 – Página com HTTP Error 403 .....	24
Figura 5.11 – Caminho de configuração da rota-padrão .....	25
Figura 5.12 – Endereço completo da Requisição GET .....	25
Figura 5.13 – Erro na Requisição GET .....	26
Figura 5.14 – Erro na Requisição GET .....	27
Figura 5.15 – Erro tratado na requisição GET .....	29
Figura 5.16 – Requisição GET no Postman .....	31
Figura 5.17 – Resposta de dados em formato XML .....	31
Figura 5.18 – Postman com resposta de dados em formato XML.....	34
Figura 5.19 – Requisição DELETE no Postman.....	35
Figura 5.20 – Requisição GET no Postman .....	36
Figura 5.21 – Requisição PUT no Postman .....	37
Figura 5.22 – Consultar os dados alterados.....	38
Figura 5.23 – Requisição GET no <i>Client</i> .....	40
Figura 5.24 – Requisições POST no <i>Client</i> .....	42
Figura 5.25 – Classes de modelo utilizadas para Parse .....	43

## LISTA DE QUADROS

Quadro 5.1 - Exemplos de URI e verbos HTTP .....	13
--	----

EMAP

**LISTA DE CÓDIGOS-FONTE**

Código-fonte 5.1 – Modelo TipoProduto.....	18
Código-fonte 5.2 – Modelo Produto.....	19
Código-fonte 5.3 – Classe TipoProdutoDAL .....	21
Código-fonte 5.4 – <i>Controller</i> Requisição Get .....	24
Código-fonte 5.5 – Classe <i>Controller</i> Requisição GET com Try Catch .....	27
Código-fonte 5.6 – <i>Controller</i> implementar o retorno uniforme.....	28
Código-fonte 5.7 – Método Get para listar todos os dados .....	30
Código-fonte 5.8 – <i>Controller</i> Requisição POST .....	33
Código-fonte 5.9 – XML de dados TipoProduto e Produto.....	34
Código-fonte 5.10 – <i>Controller</i> Requisição DELETE.....	35
Código-fonte 5.11 – <i>Controller</i> Requisição PUT.....	36
Código-fonte 5.12 – XML de alteração do TipoProduto .....	37
Código-fonte 5.13 – Cliente para requisição GET .....	40
Código-fonte 5.14 – Cliente para requisição POST.....	42
Código-fonte 5.15 – Desserialização JSON em C#.....	45
Código-fonte 5.16 – Serialização JSON em C# .....	47

## SUMÁRIO

5 WEB API (RESTFUL).....	6
5.1 Web Services .....	6
5.2 Um pouco sobre API .....	9
5.3 Fundamentos .....	9
5.4 Protocolo HTTP .....	10
5.5 URL .....	10
5.6 URN .....	11
5.7 URI .....	11
5.8 URL, URN, URI .....	11
5.9 Verbos HTTP.....	12
5.10 HTTP Status Code .....	13
5.11 Criar o projeto.....	14
5.12 Modelos.....	16
5.13 Funcionalidades .....	19
5.14 DAL .....	19
5.15 Controllers .....	21
5.16 Requisição GET .....	23
5.17 Melhorar a Requisição GET .....	26
5.18 Try Catch.....	26
5.19 Interface uniforme .....	28
5.20 Get – Listar os dados .....	29
5.21 Usar o Postman.....	30
5.22 Requisição POST .....	32
5.23 Requisição DELETE.....	35
5.24 Requisição PUT .....	36
5.25 Consumir uma API Rest.....	38
5.25.1 Padrão de retorno Json .....	38
5.25.2 Criar a aplicação Cliente .....	38
5.25.3 Cliente de requisição GET com JSON .....	39
5.25.4 Requisição POST com JSON.....	40
5.25.5 Transformação de dados (Parse) .....	42
5.26 Desserialização .....	43
5.27 Serialização .....	45
5.28 Considerações finais .....	47
REFERÊNCIAS .....	49

## 5 WEB API (RESTFUL)

Após passarmos pelas arquiteturas MVC e ORM, do .NET, vamos descobrir a importância dos *web services*, que são responsáveis pela ponte entre o nosso sistema e outras aplicações disponíveis.

O Web Service é a alma dos principais aplicativos dinâmicos. Sabe quando você solicita uma carona pelo Uber ou pede uma comida no iFood? Quem faz esta ligação entre o aplicativo e o sistema do fornecedor é o Web Service!

Você está preparado para integrar as suas soluções com outros sistemas?

### 5.1 Web Services

Expor uma funcionalidade de negócio para ser consumida por um sistema cliente fez parte de todas as soluções para os cenários de negócio que imaginamos. Os Web Services são formas de expormos tais funcionalidades como serviço. O W3C (World Wide Web Consortium) e a OASIS (Organization for the Advancement of Structured Information Standards) são as organizações responsáveis por padronizar os Web Services e empresas como IBM, Microsoft e HP, entre outras participantes do consórcio, apoiaram a criação dos padrões que determinam a tecnologia.

Segundo o W3C, a definição de Web Services é:

“A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network.” (W3C Working Group Note, 11 February 2004)

Ou seja, um sistema de software projetado para suportar a interoperabilidade entre máquinas em uma rede. Em linhas gerais, podemos dizer que Web Services são aplicações desenvolvidas para integrar sistemas que se comunicam com fraco acoplamento, por meio de mensagens que trafegam por meio da Internet ou de uma Intranet, partindo e retornando de ambientes heterogêneos.

Para essa tecnologia, pouco importa se o Web Service foi desenvolvido em uma linguagem de programação e é executado em um sistema operacional ou plataforma de hardware completamente diferente do sistema cliente que consome o

serviço. Para que a interação ocorra, basta que eles se comuniquem utilizando mensagens enviadas sobre o protocolo HTTP, comumente com os dados serializados nos formatos XML ou JSON.

Vamos imaginar o cenário de um cliente consultando o saldo bancário, via Mobile Banking. Neste contexto, o App do banco é caracterizado como o sistema cliente, responsável por consumir o serviço do Internet Banking que retornará o saldo bancário. Novamente, abstraindo inúmeros detalhes de infraestrutura e segurança, vejamos como será a solução utilizando Web Services.

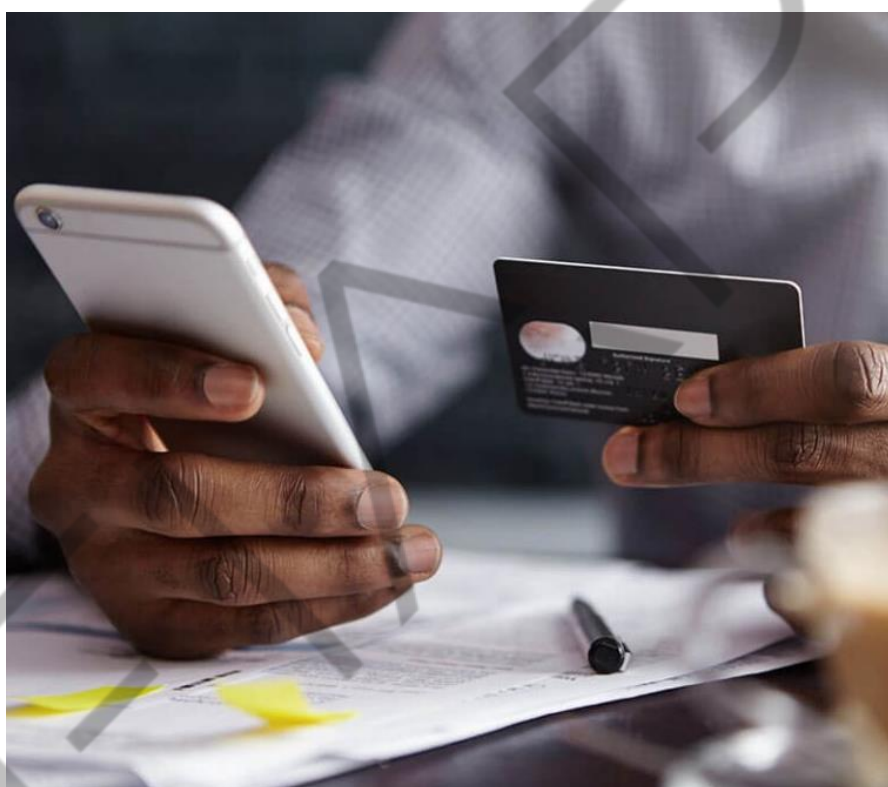


Figura 5.1 – Usuário utilizando o Internet Banking  
Fonte: Banco de imagens Shutterstock (2018)

O cliente do banco deverá selecionar os números, previamente cadastrados da agência e da conta, para, na sequência, digitar a respectiva senha em um formulário do App e disparar a solicitação de consulta.

O aplicativo, então, será responsável por criar uma mensagem serializada no formato XML ou JSON que terá, como parte do conteúdo, os dados fornecidos pelo cliente do banco. Mensagem criada, o App deverá enviá-la pela Internet sobre o protocolo HTTPS para o Web Service responsável pela consulta de saldo do Internet Banking.

Após a autenticação realizada por um componente corporativo, o Web Service do IB será responsável por criar uma mensagem de retorno, igualmente serializada no formato XML ou JSON, que terá, como parte do conteúdo, o saldo da conta. O Web Service retornará a mensagem pela Internet ao sistema cliente (App), também utilizando o protocolo HTTPS. Ao receber a comunicação, o App deverá desserializar a mensagem em um objeto e obter o saldo por meio de um método público do objeto instanciado para, assim, apresentar o valor na tela de consulta do App.

Claro que o processo foi descrito de forma muito sucinta, não imagine que a codificação da consulta ao saldo da conta corrente estará dentro do código do Web Service. Com certeza, o WS utilizará outros componentes de software que são responsáveis pela consulta de forma corporativa em uma base no mainframe do banco. Estes componentes retornarão o saldo para que o Web Service apenas crie a mensagem e atenda o cliente que consumiu o serviço.

Estes mesmos componentes que realizam a consulta de saldo para o Web Service atendem a qualquer outro canal que o cliente do banco esteja usando. Trocando em miúdos, se o cliente consultar o saldo da conta corrente pelo IB, por um ATM ou por um caixa do banco, os mesmos componentes de software deverão ser chamados para realizar a consulta e devolver o resultado ao canal chamador. Lembra do acoplamento fraco que tanto comentamos?

Neste exemplo, utilizamos a tecnologia de Web Services para atender o canal Mobile e empregamos os conceitos da Arquitetura Orientada a Serviços para atender este canal ou a qualquer outro disponibilizado ao cliente que deseja realizar uma consulta de saldo.

Conceitualmente, pelos exemplos, percebemos que um Web Service é um software que possibilita ao provedor de serviços atender a consumidores trocando mensagens pela rede. Os sistemas que se comunicaram pouco sabem sobre as capacidades tecnológicas envolvidas. O App desconhece se o Web Service foi codificado em Java EE, .NET ou PHP e o Web Service não se preocupou se o App foi desenvolvido em Java Android, Swift ou alguma linguagem híbrida.

Isso acontece, pois, tanto na requisição, quanto na resposta, falamos sobre a necessidade de serializar as mensagens em formatos específicos XML ou JSON.



Apenas como histórico, na criação dos padrões para Web Services, as empresas participantes do W3C adotaram o protocolo SOAP baseado em XML. O REST surgiu em 2000, na UC Irvine, em uma tese de doutorado (PHD), e passou a compor a definição de Web Services do W3C em meados de 2004.

## 5.2 Um pouco sobre API

Uma API ou WebAPI é uma evolução de um WebService e ignora os detalhes de implementação e sintaxe do SOAP, deixando o tráfego de informação mais leve e a implementação mais simples. Uma API Web utiliza o padrão arquitetural REST, que tem como foco a diversidade dos recursos ou nome (Ex. recursos para Produtos, recursos para Clientes), diferente dos webservices SOAP, cujo foco são as operações (Ex.: Consultar Clientes, Cadastrar Clientes etc).

A utilização dos serviços no padrão REST passou de uma tendência para uma realidade. Nos últimos anos o crescimento aconteceu de forma exponencial, e um dos grandes colaboradores desse crescimento é a quantidade de serviços usados em aplicativos móveis. Eles precisam ter a complexidade cada vez mais simplificada e também a quantidade dos dados trafegados cada vez mais reduzida.

Assim, o *framework* **ASP.NET Web API** é uma plataforma ideal e indicada para a construção das aplicações no padrão REST.

Em outras palavras, uma API REST não depende de XML para trafegar informações e ignora detalhes de implementação e sintaxe do protocolo. Os formatos mais comuns de um API são JSON, texto e XML, dando ao desenvolvedor o poder de escolha do melhor formato de acordo com sua necessidade. Grandes empresas, como Facebook, Google, Netflix e LinkedIn, passaram a usá-la e disponibilizam APIs a serem usadas por parceiros e usuários dos serviços.

O ASP.NET WEB API utiliza HTTP com REST.

## 5.3 Fundamentos

Como podemos notar, no tópico anterior, falamos sobre WebServices, o surgimento das novas tecnologias e a diferença entre SOAP e REST até chegarmos

no **ASP.NET WEB API**. Mas temos mais fundamentos essenciais para o funcionamento de tudo isso, os mais comuns serão explicados nos tópicos a seguir.

## 5.4 Protocolo HTTP

Não conseguimos abordar os assuntos abaixo sem falar sobre o HTTP (*Hypertext Transfer Protocol*), protocolo da camada de aplicação do modelo OSI para transferência dos dados na rede mundial dos computadores. Em outras palavras, são conjuntos de regras de transmissão dos dados que permitem que máquinas com diferentes configurações possam se comunicar em uma mesma “linguagem/idioma”.

Seu funcionamento é baseado em requisição e resposta *client* e *server*, ou seja, o *client*, ao solicitar um recurso na Internet, envia um pacote dos dados com cabeçalhos (Headers) a um URI (Ou URL) e o destinatário ou servidor vai devolver uma resposta que pode ser um recurso ou outro cabeçalho.

## 5.5 URL

Uniform Resource Locator (Localizador de Recursos Universal), como o próprio nome diz, é um endereço, um Host de um recurso (como um arquivo, uma impressora etc.), que permite o acesso a uma determinada rede disponível; seja Internet ou até mesmo uma rede corporativa como uma empresa (Intranet).

Seu funcionamento é basicamente associar um endereço remoto com um nome de recurso na Internet/Intranet.

Exemplo de URL:

- fiap.com.br
- google.com.br
- facebook.com

## 5.6 URN

Da sigla para *Uniform Resource Name* (Nome de Recursos Universal), é o nome do recurso que será acessado.

Exemplo de URN:

- index.html
- contato.aspx
- home.php

## 5.7 URI

É o acrônimo de *Uniform Resource Identifier* (Identificador de Recursos Universal), podendo ser uma página, uma imagem, um vídeo etc., tendo como principal propósito permitir a interação com o recurso por meio de uma rede, isto é, um identificador único para que não seja confundido.

Exemplo de URI:

- <https://www.facebook.com/zuck>
- <https://www.fiap.com.br/online/graduacao/bacharelado/sistemas-de-informacao/>
- [https://www.google.com.br/search?rlz=1C1HIJA\\_enBR723BR723&ei=fUyPWqrylce5wgTT7pHQA&q=fiap&oq=fiap&gs\\_l=psy-ab.3..35i39k1j0i131k1j0l3j0i67k1j0l4.1769.2212.0.2379.4.4.0.0.0.810.332.4.4.0....0...1.1.64.psy-ab..0.4.329...0i131i67k1.0.G8Vp2Tigdhk](https://www.google.com.br/search?rlz=1C1HIJA_enBR723BR723&ei=fUyPWqrylce5wgTT7pHQA&q=fiap&oq=fiap&gs_l=psy-ab.3..35i39k1j0i131k1j0l3j0i67k1j0l4.1769.2212.0.2379.4.4.0.0.0.810.332.4.4.0....0...1.1.64.psy-ab..0.4.329...0i131i67k1.0.G8Vp2Tigdhk)

## 5.8 URL, URN, URI

Todas essas definições podem ter deixado você confuso, então, vamos explicar de uma maneira mais simples:

A URI é a composição do Protocolo (http:// ou https://), a localização do recurso (URL - **fiap.com.br**) e do nome do recurso (URN - **/online/graduacao/bacharelado/sistemas-de-informacao/**).

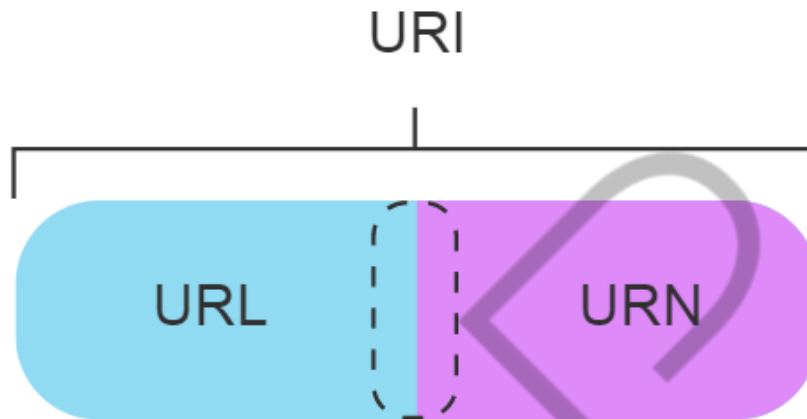


Figura 5.2 – Composição de URL e URN  
Fonte: Google Imagens (2018)

## 5.9 Verbos HTTP

Os verbos HTTP são os métodos de requisição usados para indicar a ação que será executada quando chamamos um recurso de uma API Rest. Segue a lista dos mais conhecidos e utilizados:

- GET
- Responsável por buscar/consultar informações por meio de uma URI. É um método idempotente, isto é, não altera nada, não importa quantas vezes a requisitamos, será o mesmo resultado.
- POST
- Responsável por enviar informações com conteúdo embutido no corpo, podendo ser JSON, XML ou texto por meio de uma URI. É utilizado muitas vezes para gravar uma nova informação no sistema.
- DELETE
- Responsável por remover informações por uma URI.

- PUT
- Responsável por atualizar informações, com conteúdo embutido no corpo, podendo ser XML ou JSON.

Veja alguns exemplos no quadro abaixo:

Endpoint (caminho)	Método	Ação
/api/TipoProduto/{idTipoProduto}	GET	Retorna a lista de TipoProduto.
/api/TipoProduto/	POST	Insere um novo TipoProduto.
/api/TipoProduto/{idTipoProduto}	DELETE	Remove o TipoProduto com o idTipoProduto = {idTipoProduto}.
/api/TipoProduto/{idTipoProduto}	PUT	Altera o TipoProduto com o idTipoProduto = {idTipoProduto}.

Quadro 5.1 - Exemplos de URI e verbos HTTP  
Fonte: Elaborado pelo autor (2018)

## 5.10 HTTP Status Code

O *Status Code* de uma requisição é parte importante de uma WebAPI, pois com ele é possível reconhecer facilmente o que aconteceu com a requisição. O código é um padrão numérico que apresenta o resultado da ação. Seguem alguns exemplos:

- 200 - OK
- A requisição foi bem-sucedida.
- 201 - Created
- O pedido foi cumprido e resultou em um novo recurso que está sendo criado.
- 401 - Unauthorized
- A URI especificada precisa de autenticação.
- 403 - Forbidden
- Indica que o servidor se recusa a atender à solicitação.
- 404 - Not Found
- O recurso requisitado não foi encontrado.
- 500 - Internal Server Error
- Indica um erro do servidor ao processar a solicitação.

## 5.11 Criar o projeto

Para iniciar a criação de um novo serviço ASP.NET WEB API, iremos seguir o mesmo modelo de negócio dos capítulos anteriores, a **Fiap Smart City**, nossa cidade “virtual”, cada vez mais tecnológica, proporcionando à população melhores condições e sustentabilidade.

Vamos colocar a mão na massa?

No Visual Studio 2017, selecione o menu **File > New > Project** (a tecla de atalho **Ctrl + Shift + N**). Selecione também a linguagem **Visual C# > Web** na parte esquerda da janela. No centro, vamos selecionar ainda o tipo de projeto **ASP.NET Web Application** (.NET Framework). Na parte inferior, temos caixas de texto para definir o nome do projeto, o local no sistema dos arquivos e o nome da solução. Para nosso exemplo, vamos usar como nome do projeto e da solução **FiapSmartCityWebAPI**.

A figura apresenta os passos para a seleção da linguagem e o tipo de projeto. Seguem:

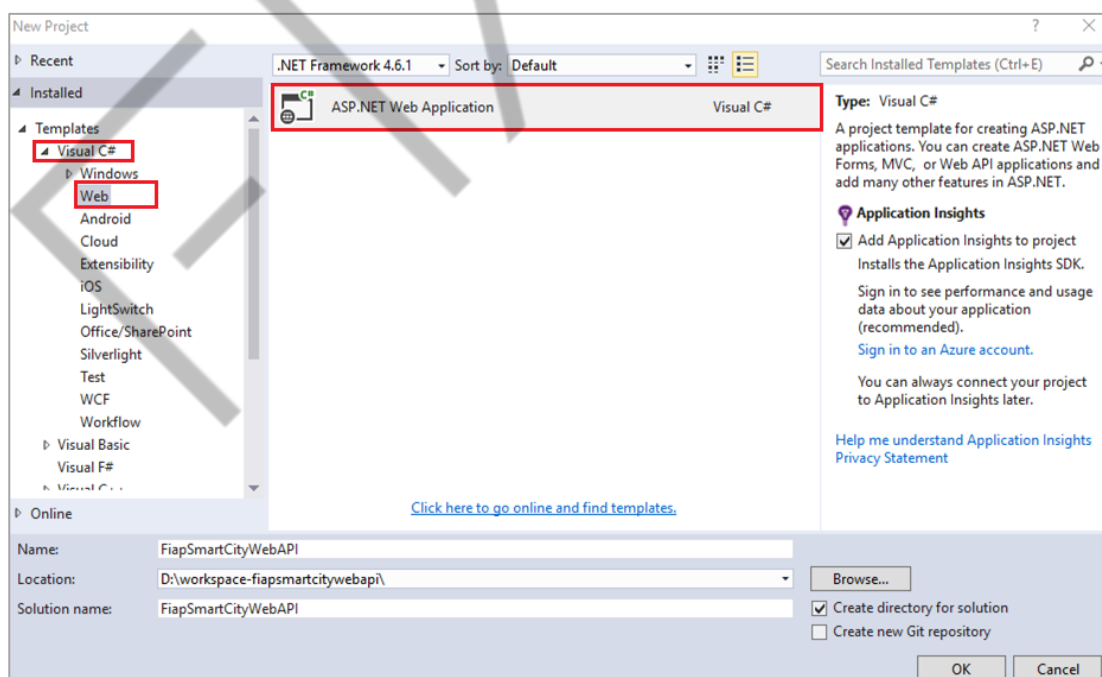


Figura 5.3 – Projeto ASP.NET Web  
Fonte: Elaborado pelo autor (2018)

Na próxima janela, devemos escolher um *template* para a nossa aplicação web. Selecione o *template* **Empty** e, na parte de baixo da janela, marque a caixa de seleção de pasta e referências com a opção **Web API**. Lembrando que, nesse primeiro momento, não criaremos **Testes Unitários** e também não hospedaremos nossa Web API no **Microsoft Azure**. Logo, suas caixas de seleção deverão estar desmarcadas. Após todas essas configurações, clique no botão **OK**.

A figura “Template Web API” apresenta todas as opções que devem ser selecionadas e as que não será preciso selecioná-las para a criação correta do projeto. Veja:

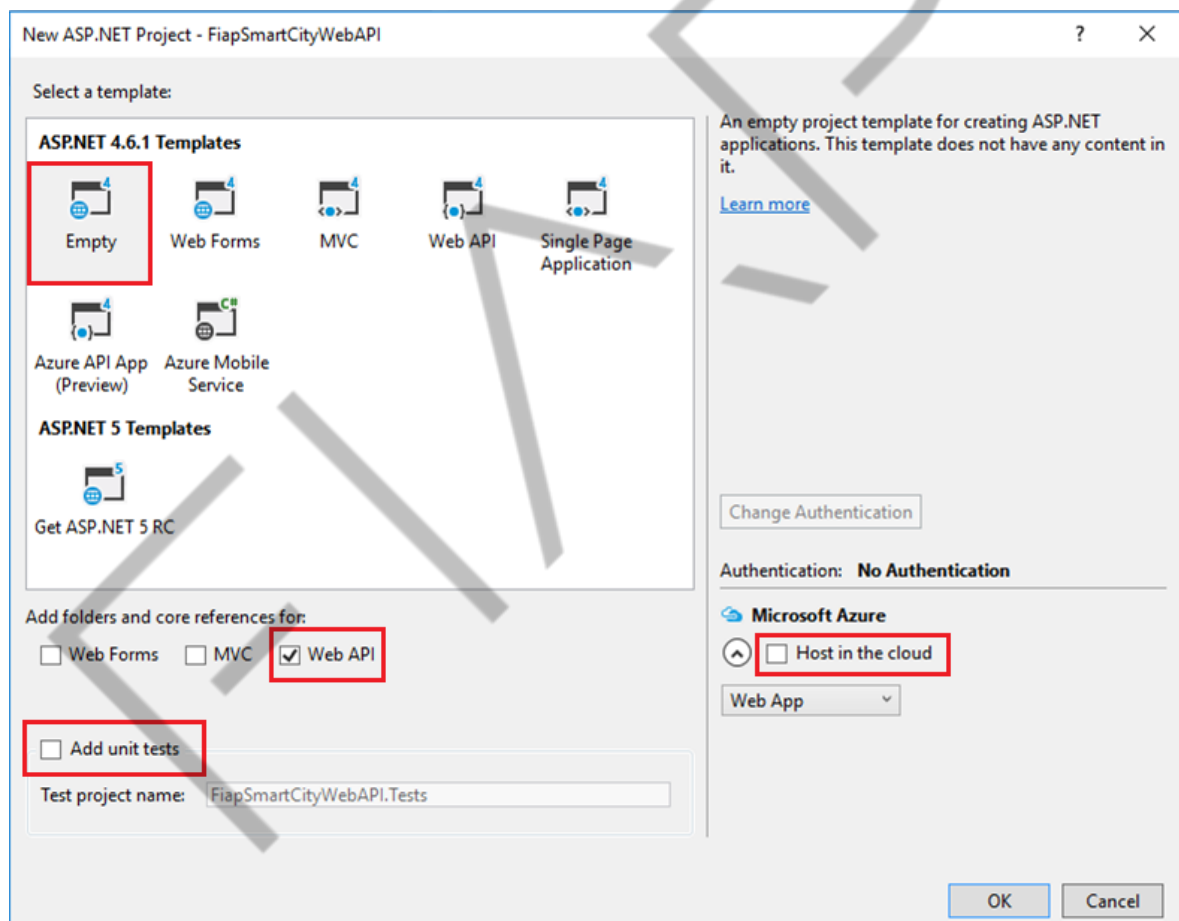


Figura 5.4 – Template Web API  
Fonte: Elaborado pelo autor (2018)

Finalizado a operação de criação, conseguimos verificar a estrutura criada para o nosso projeto. Na janela **Solutions Explorer**, temos nossa solução, nosso projeto da **WebAPI** e as pastas *Controllers*, *Models* e *Views*, que são idênticas ao projeto Asp.NET MVC.

Observe na figura:

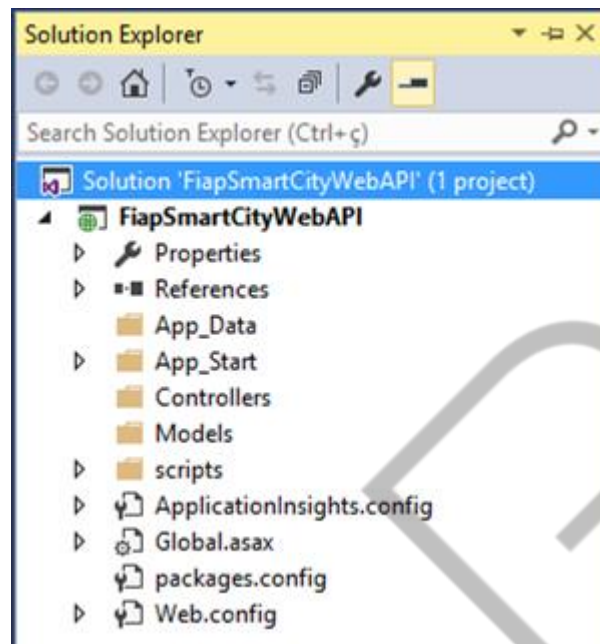


Figura 5.5 – Estrutura projeto Asp.NET WebAPI  
Fonte: Elaborado pelo autor (2018)

## 5.12 Modelos

Com o nosso projeto criado, iremos seguir a mesma estrutura que foi explicada no Capítulo **ASP.NET MVC**.

Nesse primeiro momento, iremos começar pela camada de modelos, onde criaremos a estrutura dos nossos dados. Em seguida, uma camada de acesso a dados e, para finalizar, nossos controladores.

A seguir, segue a representação UML para as nossas classes de modelo:



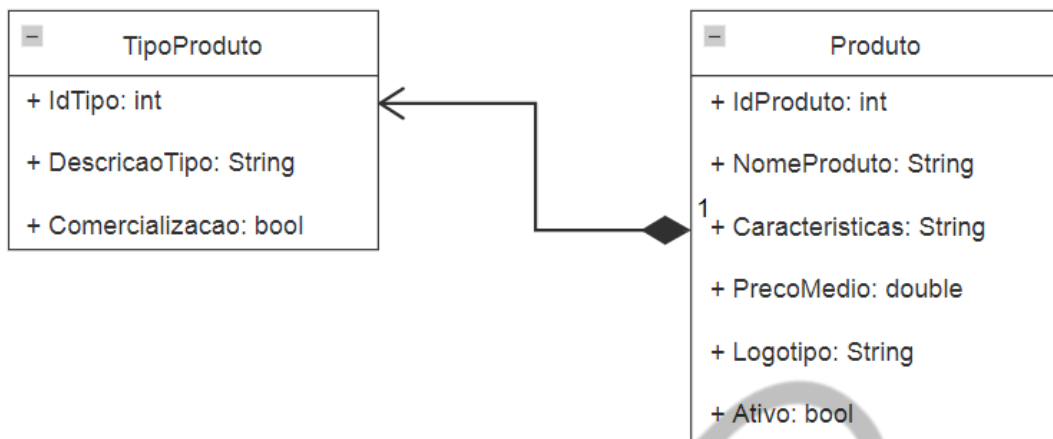


Figura 5.6 – Diagrama de classe produto e categoria  
 Fonte: Elaborado pelo autor (2018)

Os modelos devem ser adicionados no *namespace* Models do projeto. Para criar o modelo **TipoProduto**, clique com o botão direito na pasta **Models** e escolha a opção **Add > Class**. Defina o nome como TipoProduto.cs, utilize o Diagrama de Classe e adicione os atributos `IdTipo`, `DescricaoTipo` e `Comercializado` com seus respectivos tipos. O código-fonte a seguir apresenta a implementação do modelo TipoProduto.

```

using System;
using System.Collections.Generic;
using System.Linq;
namespace FiapSmartCityWebAPI.Models
{
    public class TipoProduto
    {
        public int IdTipo { get; set; }
        public String DescricaoTipo { get; set; }
        public bool Comercializado { get; set; }

        public List<Produto> Produtos { get; set; }

        public TipoProduto()
        {
            this.Produtos = new List<Produto>();
        }

        // MOCK - Método para adicionar um produto ao Tipo
        public void Adiciona(Produto produto)
        {
            this.Produtos.Add(produto);
        }

        // MOCK - Método para remover um produto do tipo
    }
}
  
```

```
        public void Remove(long id)
        {
            Produto produto = Produtos.FirstOrDefault(p =>
p.IdProduto == id);

            Produtos.Remove(produto);
        }

        // MOCK - Método para alterar um produto do tipo
        public void Altera(Produto produto)
        {
            Remove(produto.IdProduto);
            Adiciona(produto);
        }
    }
}
```

Código-fonte 5.1 – Modelo TipoProduto  
Fonte: Elaborado pelo autor (2018)

Seguindo os passos anteriores e o Diagrama de Classe, vamos criar a classe para o modelo de Produto. O diagrama apresenta uma agregação entre Produto e TipoProduto, sendo, assim, na classe de Produto precisamos ter uma propriedade do tipo TipoProduto. Veja o exemplo no código-fonte:

```
using System;

namespace FiapSmartCityWebAPI.Models
{
    public class Produto
    {
        public int IdProduto { get; set; }
        public String NomeProduto { get; set; }
        public String Caracteristicas { get; set; }
        public double PrecoMedio { get; set; }
        public String Logotipo { get; set; }
        public bool Ativo { get; set; }

        // Referência para classe TipoProduto
        public TipoProduto idTipoProduto { get; set; }
    }

    public Produto() { }

    public Produto(int IdProduto, string NomeProduto, string
Caracteristicas, double PrecoMedio, string Logotipo, bool
Ativo, int IdTipoProduto)
    {
        this.IdProduto = IdProduto;
    }
}
```

```
        this.NomeProduto = NomeProduto;
        this.Caracteristicas = Caracteristicas;
        this.PrecoMedio = PrecoMedio;
        this.Logotipo = Logotipo;
        this.Ativo = Ativo;
        this.IdTipoProduto = IdTipoProduto;
    }
}
```

Código-fonte 5.2 – Modelo Produto  
Fonte: Elaborado pelo autor (2018)

### 5.13 Funcionalidades

Temos dois modelos definidos e criados em nosso projeto, precisamos agora desenvolver os mecanismos para alimentar com dados. Vamos iniciar criando nossa Camada DAL e depois nossos controladores, que serão os responsáveis pelas requisições à API e também pela validação do fluxo dos nossos serviços.

### 5.14 DAL

*Data Access Layer* (Objeto de Acesso a Dados) é um padrão para persistência ou consulta de dados, separando as regras de negócio das regras de acesso a banco de dados.

Nesse exemplo de ASP.NET WebAPI, trabalharemos com **TipoProduto** e **Produtos** relacionado ao mesmo. Segue o exemplo do código da classe DAL.cs, que irá simular nosso banco de dados:

```
using FiapSmartCityWebAPI.Models;
using System.Collections.Generic;

namespace FiapSmartCityWebAPI.DAL
{
    public class TipoProdutoDAL
    {
        // Lista criada para armazenar uma lista de Tipo de
        // produto simulando o banco de dados
        private static Dictionary<long, TipoProduto>
        bancoTipoProduto = new Dictionary<long, TipoProduto>();
        private static int contadorBanco = 2;

        // Construtor estático serve para criar objetos do Tipo
        // de Produto e Produto
    }
}
```

```

// Simulando o banco de dados
static TipoProdutoDAL()
{
    TipoProduto EnergiaSolar = new TipoProduto();
    EnergiaSolar.IdTipo = 1;
    EnergiaSolar.DescricaoTipo = "Energia Solar";
    EnergiaSolar.Comercializado = true;

    Produto FotoVoltatica = new Produto();
    FotoVoltatica.IdProduto = 800;
    FotoVoltatica.NomeProduto = "Energia Solar
Fotovoltatica";
    FotoVoltatica.Caracteristicas = @"A tecnologia
fotovoltaica (FV)
converte
diretamente os raios
solares em
eletricidade";
    FotoVoltatica.PrecoMedio = 4000.00;
    FotoVoltatica.Logotipo =
@"data:image/jpeg;base64";
    FotoVoltatica.Ativo = true;
    FotoVoltatica.IdTipoProduto = EnergiaSolar.IdTipo
= 1;

    //Referência do Novo Produto
    EnergiaSolar.Adiciona(FotoVoltatica);

    TipoProduto tinta = new TipoProduto();
    tinta.IdTipo = 2;
    tinta.DescricaoTipo = "Tinta";
    tinta.Comercializado = true;

    //Inserir Registro no Banco
    bancoTipoProduto.Add(1, EnergiaSolar);
    bancoTipoProduto.Add(2, tinta);
}

public void Inserir(TipoProduto TipoProduto)
{
    contadorBanco++;
    TipoProduto.IdTipo = contadorBanco;
    bancoTipoProduto.Add(contadorBanco, TipoProduto);
}

public TipoProduto Consultar(int IdTipo)
{
    return bancoTipoProduto[IdTipo];
}

```

```
public IList<TipoProduto> Listar()  
{  
    return  
List<TipoProduto>(bancoTipoProduto.Values);  
}  
  
public void Excluir(int IdTipo)  
{  
    bancoTipoProduto.Remove(IdTipo);  
}  
  
public void Alterar(TipoProduto tipoProduto)  
{  
    bancoTipoProduto[tipoProduto.IdTipo]  
tipoProduto;  
}  
}
```

Código-fonte 5.3 – Classe TipoProdutoDAL  
Fonte: Elaborado pelo autor (2018)

**DICA:** Os componentes DAL podem ser usados do projeto FiapSmartCity (**MVC e Entity Framework**), basta baixar as bibliotecas do EF e configurar o acesso ao banco de dados no projeto de WebAPI.

## 5.15 Controllers

Em um projeto **ASP.NET WebAPI**, toda a requisição será recebida e gerenciada por um *Controller*, que é responsável por receber o pedido, acionar os componentes necessários e gerar a resposta para o navegador.

Chegou a hora de criarmos nossa *Controller*.

Com um clique no botão direito na pasta *Controllers* do projeto, selecione a opção **Add > Controller** como na Figura “Adicionar Controller”. O Visual Studio apresentará a janela **Add Scaffold**, selecione, então, a opção “**Web API 2 Controller - Empty**” conforme Figura “Selecionar o Scaffold do Controller”.

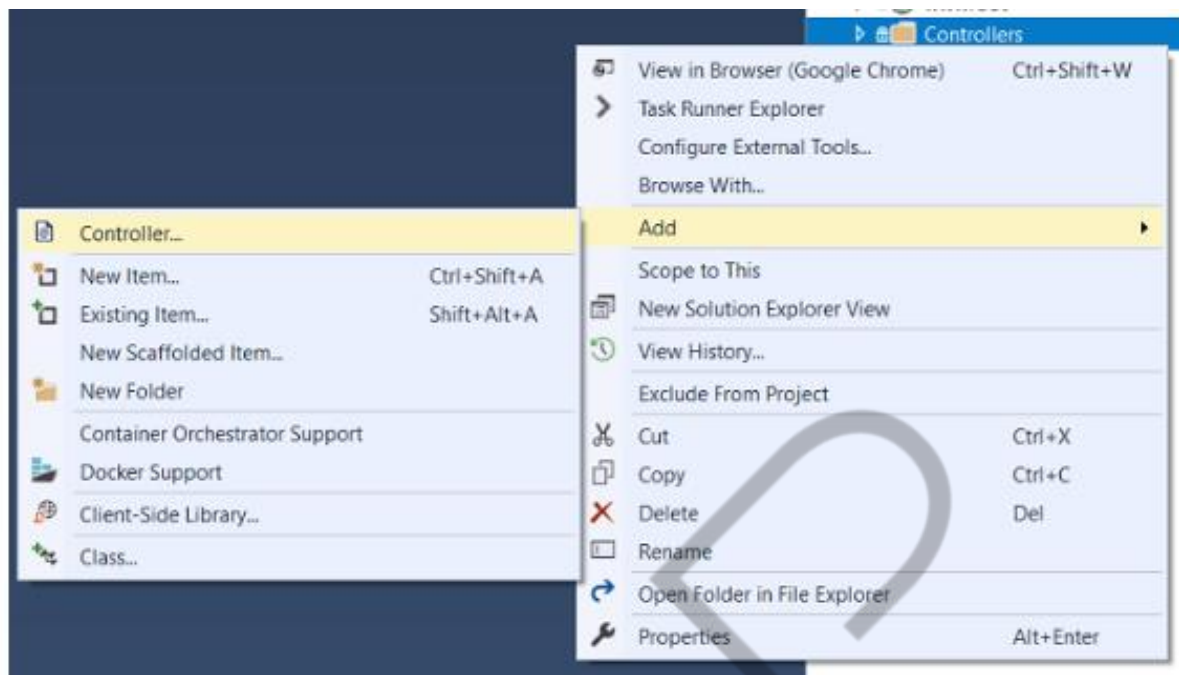


Figura 5.7 – Adicionar *Controller*  
Fonte: Elaborado pelo autor (2018)

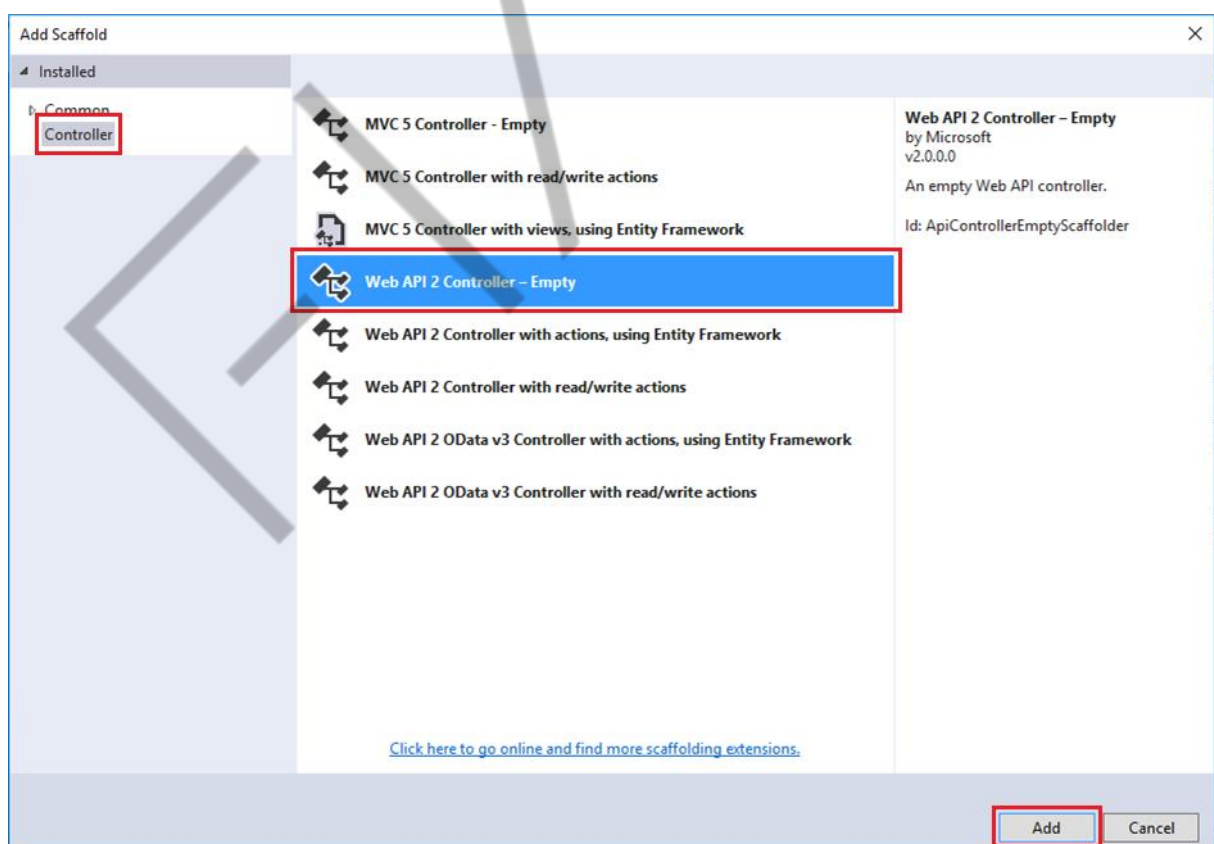


Figura 5.8 – Selecionar o *Scaffold* do Controller  
Fonte: Elaborado pelo autor (2018)

O próximo passo é definir o nome do controlador, que será **TipoProdutoController** em nosso projeto. Clique no botão Add e aguarde a criação. Lembre-se, todo *controller* deverá ter o sufixo **Controller** em seu nome.

Pronto! Primeiro controlador criado no projeto. Agora podemos observar a classe criada no *namespace Controllers*. No código da classe Controller, é possível ver a importação do *namespace System.Web.Http* e a extensão da classe **System.Web.Http.ApiController**.

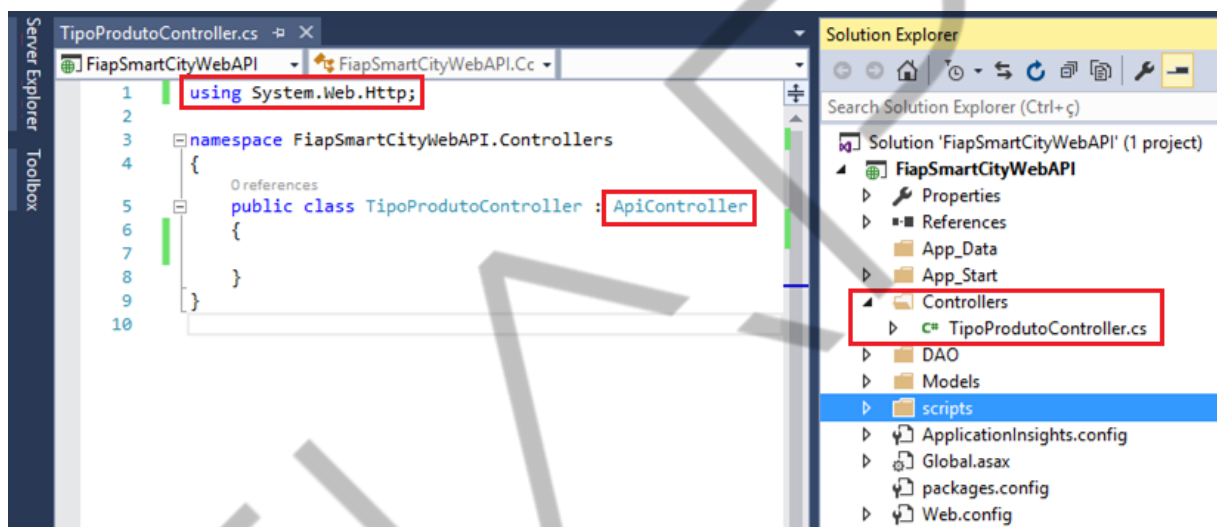


Figura 5.9 – Detalhes de um Controller  
Fonte: Elaborado pelo autor (2018)

## 5.16 Requisição GET

Com nosso *controller* criado, agora criaremos nossa primeira requisição, baseada nos **Verbos HTTP**, a requisição **GET**.

Nosso método GET será implementado capturando o Id para o tipo de produto, consultando nossa camada dos dados com o Id capturado e retornando um objeto TipoProduto. Por convenção, o nome do nosso método será Get(). Segue código abaixo:

```
using FiapSmartCityWebAPI.DAL;
using FiapSmartCityWebAPI.Models;
using System.Web.Http;

namespace FiapSmartCityWebAPI.Controllers
{
```

```

public class TipoProdutoController : ApiController
{
    public IHttpActionResult Get(int id)
    {
        try
        {
            TipoProdutoDAL dal = new TipoProdutoDAL();
            TipoProduto TipoProduto = dal.Consultar(id);
            return Ok(TipoProduto);
        }
        catch (KeyNotFoundException)
        {
            return NotFound();
        }
    }
}

```

Código-fonte 5.4 – *Controller* Requisição Get  
Fonte: Elaborado pelo autor (2018)

*Controller* criado, requisição **GET** criada, podemos fazer o primeiro teste. Pressione a tecla **F5** e aguarde o navegador-padrão do seu computador ser aberto, será exibido o resultado da figura abaixo:

#### HTTP Error 403.14 - Forbidden

O servidor Web está configurado para não listar o conteúdo deste diretório.

##### Causas mais prováveis:

- Um documento padrão não está configurado para a URL solicitada, e a pesquisa no diretório não está habilitada no servidor.

##### Ações que você pode tentar:

- Se não deseja habilitar a pesquisa no diretório, verifique se um documento padrão está configurado e se ele existe.
- Habilite a pesquisa no diretório.
  1. Vá para o diretório de instalação do IIS Express.
  2. Execute `appcmd set config /section:system.webServer/directoryBrowse /enabled:true` para habilitar a pesquisa de diretório no nível do servidor.
  3. Execute `appcmd set config ["%IISExpress%\wwwroot"] /section:system.webServer/directoryBrowse /enabled:true` para habilitar a pesquisa de diretório no nível do site.
- Verifique se o atributo `configuration/system.webServer/directoryBrowse.enabled` está definido como verdadeiro no arquivo de configuração do site ou aplicativo.

##### Informações detalhadas do erro:

<b>Módulo</b>	DirectoryListingModule	<b>URL solicitada</b>	http://localhost:58576/
<b>Notificação</b>	ExecuteRequestHandler	<b>Caminho físico</b>	C:\workspace\flap\SmartCityWebAPI\FlapSmartCityWebAPI
<b>Manipulador</b>	StaticFile	<b>Método de logon</b>	Anônimo
<b>Código do erro</b>	0x00000000	<b>Usuário de logon</b>	Anônimo
		<b>Diretório de Rastreamento de Solicitação</b>	C:\Users\pedroivco\Documents\IISExpress\TraceLogFiles\IISExpress\SMARTCITYWEBAPI

##### Mais informações:

Este erro ocorre quando um documento não está especificado na URL, nenhum documento padrão está especificado para o site ou aplicativo, e a listagem de diretório não está habilitada para o site ou aplicativo. Essa configuração pode ser desabilitada para proteger o conteúdo do servidor.

[Exibir mais informações >](#)

Figura 5.10 – Página com HTTP Error 403  
Fonte: Elaborado pelo autor (2018)

O navegador irá exibir uma tela de erro informando que “O servidor Web está configurado para não listar o conteúdo deste diretório” e apesar de apresentar uma mensagem de erro, isso não significa que nosso teste não foi bem-sucedido.



Assim como nas aplicações ASP.NET MVC, precisamos colocar o endereço completo do *controller*, conforme sua rota:

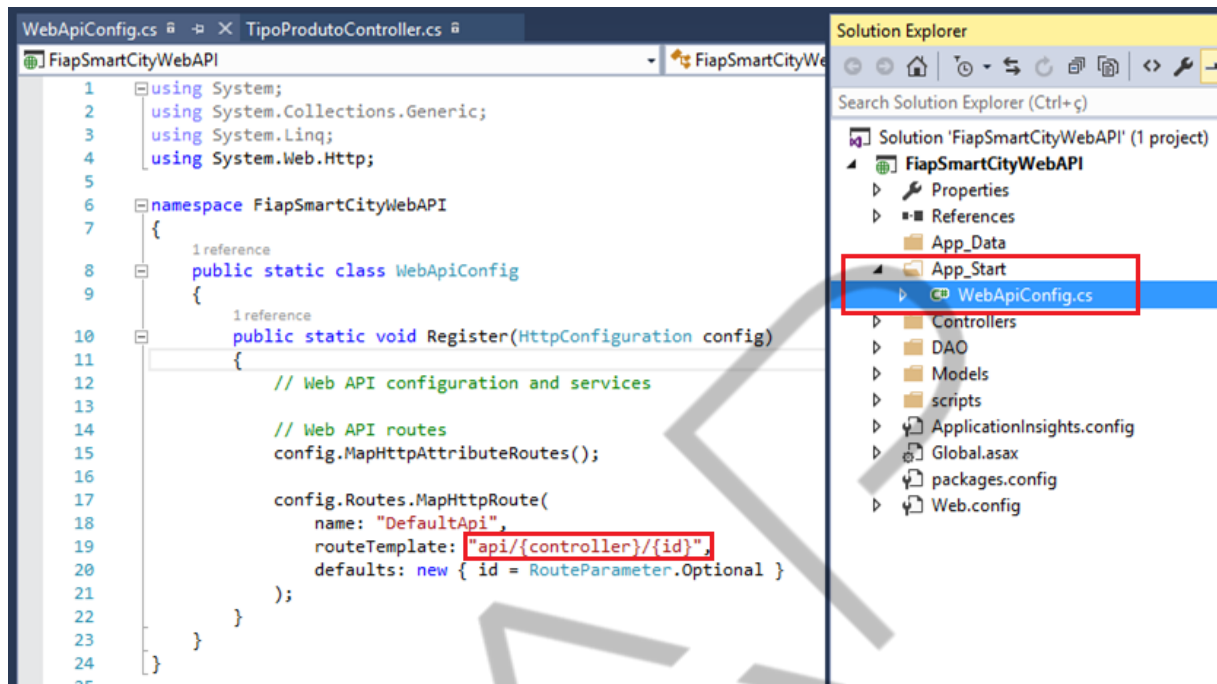


Figura 5.11 – Caminho de configuração da rota-padrão  
Fonte: Elaborado pelo autor (2018)

A figura seguinte apresenta o endereço completo para a execução do *controller*, e o resultado esperado, o nosso **TipoProduto** e **Produtos** cadastrado:

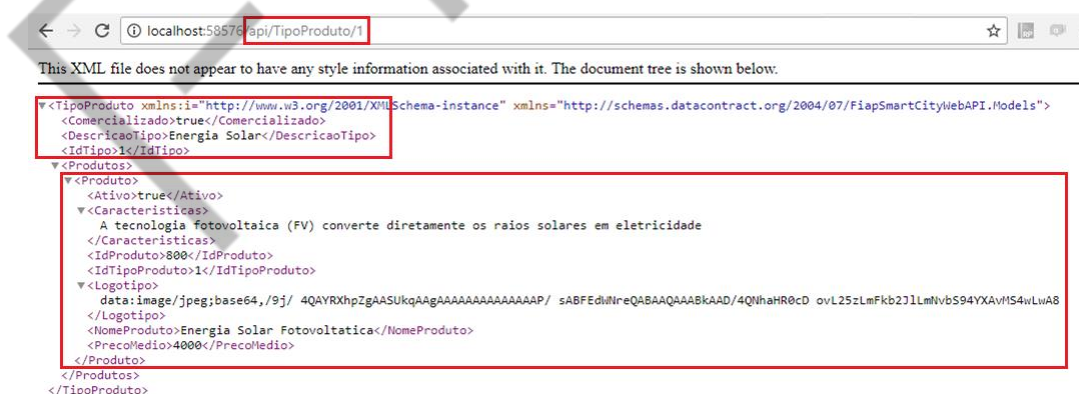


Figura 5.12 – Endereço completo da Requisição GET  
Fonte: Elaborado pelo autor (2018)

**IMPORTANTE:** No endereço `http://localhost:5876/api/TipoProduto/1`, o número final (1) indica qual é o código do tipo de produto que vamos consultar.

## 5.17 Melhorar a Requisição GET

E se executarmos uma requisição GET no nosso projeto, passando um **idTipoProduto** que não está cadastrado no nosso banco de dados, o que acontece? Vamos ver:



Figura 5.13 – Erro na Requisição GET  
Fonte: Elaborado pelo autor (2018)

Como podemos notar na execução anterior, temos um erro do tipo **<ExceptionType>System.Collections.Generic.KeyNotFoundException</ExceptionType>** que está sendo exibido diretamente para o requisitante nossa ASP.NET Web API. Dado o cenário, vamos aprender a tratar o erro e retornar uma resposta-padrão, isto é, criar uma interface uniforme.

## 5.18 Try Catch

Um bloco *“try”* é chamado de bloco “protegido” porque, caso ocorra algum problema com os comandos dentro do bloco, a execução desviará para os blocos *“catch”* correspondentes.

Segue o código abaixo:

```

namespace FiapSmartCityWebAPI.Controllers
{
    public class TipoProdutoController : ApiController
    {
        public TipoProduto Get(int id)
        {
            try
            {
                TipoProdutoDAL dal = new TipoProdutoDAL();
                TipoProduto TipoProduto = dal.Busca(id);
                return TipoProduto;
            }
            // Capturando um exceção de Chave não encontrada
            catch (KeyNotFoundException)
            {
                throw;
            }
        }
    }
}

```

Código-fonte 5.5 – Classe *Controller* Requisição GET com Try Catch  
Fonte: Elaborado pelo autor (2018)

Implementado o *Try Catch*, vamos realizar novamente o teste:



Figura 5.14 – Erro na Requisição GET  
Fonte: Elaborado pelo autor (2018)

Mas por quê? Se implementamos o *Try Catch*? Como citado anteriormente, o *Try* executa o bloco de comando, caso ocorra algum erro, o fluxo é direcionado para

o Catch, já que nenhum dos dois trata respostas de sucesso ou falha. Porém, nada está perdido, pois veremos sobre **interface uniforme** no próximo tópico.

## 5.19 Interface uniforme

O que é? De onde vem? Como se alimenta?

Brincadeiras à parte, interface uniforme é nada mais que um retorno unificado, diferente de um modelo único das respostas, é um padrão que toda a *web* entende, assim, qualquer serviço/aplicação/usuário que usar nossa WebAPI poderá ler o HTTP *Status Code* e entenderá a resposta.

Colocando prática, usaremos como retorno dos métodos o tipo **IHttpActionResult**, que nada mais é do que uma *factory* de respostas **HTTP** usando **HttpResponseMessage**.

Segue o código abaixo:

```
using FiapSmartCityWebAPI.DAL;
using FiapSmartCityWebAPI.Models;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace FiapSmartCityWebAPI.Controllers
{
    public class TipoProdutoController : ApiController
    {
        public IHttpActionResult Get(int id)
        {
            try
            {
                TipoProdutoDAL dal = new TipoProdutoDAL();
                TipoProduto TipoProduto = dal.Busca(id);
                return Ok(TipoProduto) ;
            }
            catch (KeyNotFoundException)
            {
                return NotFound() ;
            }
        }
    }
}
```

Código-fonte 5.6 – *Controller* implementar o retorno uniforme

Fonte: Elaborado pelo autor (2018)

Como podemos ver, implementamos a interface uniforme para o cenário de sucesso e falha. O método GET possui como resultado um objeto do tipo `IHttpActionResult`, permitindo, assim, o uso dos métodos **Ok()** e **NotFound()** para padronizar a interface de retorno.

Caso o fluxo para a consulta de um `TipoProduto` seja executado com sucesso, o método `Ok ()` devolverá o objeto `TipoProduto` encontrado e o `StatusCode 200`. Para o fluxo de insucesso, o método `NotFound()` será disparado e devolverá o `StatusCode 404` para o solicitante.

Vamos ver o resultado da execução?

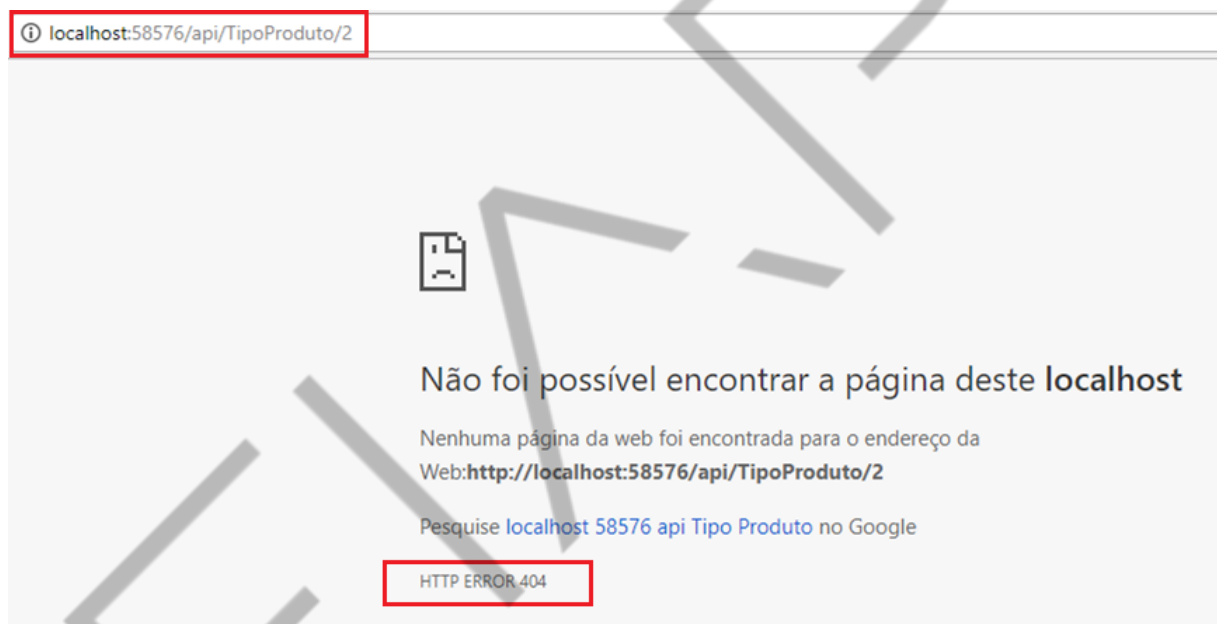


Figura 5.15 – Erro tratado na requisição GET  
Fonte: Elaborado pelo autor (2018)

A requisição retornou o HTTP Status Code 404 esperado, conforme podemos ver na **figura anterior**.

## 5.20 Get – Listar os dados

O nosso *controller* `TipoProdutos` possui um método `Get ()` que recebe o `Id` como parâmetro e consulta a informação de um determinado tipo. Podemos ter mais um método `Get ()`, porém é obrigatório que a assinatura seja diferente. Vamos

implementar um método Get () sem nenhum parâmetro e seu objeto será o retorno de uma lista de tipo, veja o código-fonte abaixo:

```
using FiapSmartCityWebAPI.DAL;
using FiapSmartCityWebAPI.Models;
using System;
using System.Collections.Generic;
using System.Web.Http;

namespace FiapSmartCityWebAPI.Controllers
{
    public class TipoProdutoController : ApiController
    {
        // Método responsável por listar os Tipos de Produtos
        public IHttpActionResult Get()
        {
            return Ok( new TipoProdutoDAL().Listar());
        }

        // Método responsável por consultar o detalhe de um
        Tipo
        public IHttpActionResult Get(int id)
        {
            try
            {
                TipoProdutoDAL dal = new TipoProdutoDAL();
                TipoProduto TipoProduto = dal.Consultar(id);
                return Ok(TipoProduto);
            }
            catch (KeyNotFoundException)
            {
                return NotFound();
            }
        }
    }
}
```

Código-fonte 5.7 – Método Get para listar todos os dados  
Fonte: Elaborado pelo autor (2018)

## 5.21 Usar o Postman

A figura “Requisição GET no Postman” mostra como fazer uma requisição no aplicativo Postman. Selecione o tipo de requisição desejada, que, no nosso caso, é **GET**. Em seguida, informe a **URI** da WebAPI e clique no botão **Send**. Nesse primeiro momento, iremos simular uma requisição de sucesso e podemos ver, na aba **Body**, o resultado no corpo da requisição, e o **Status: 200 OK**, conforme nossa configuração.

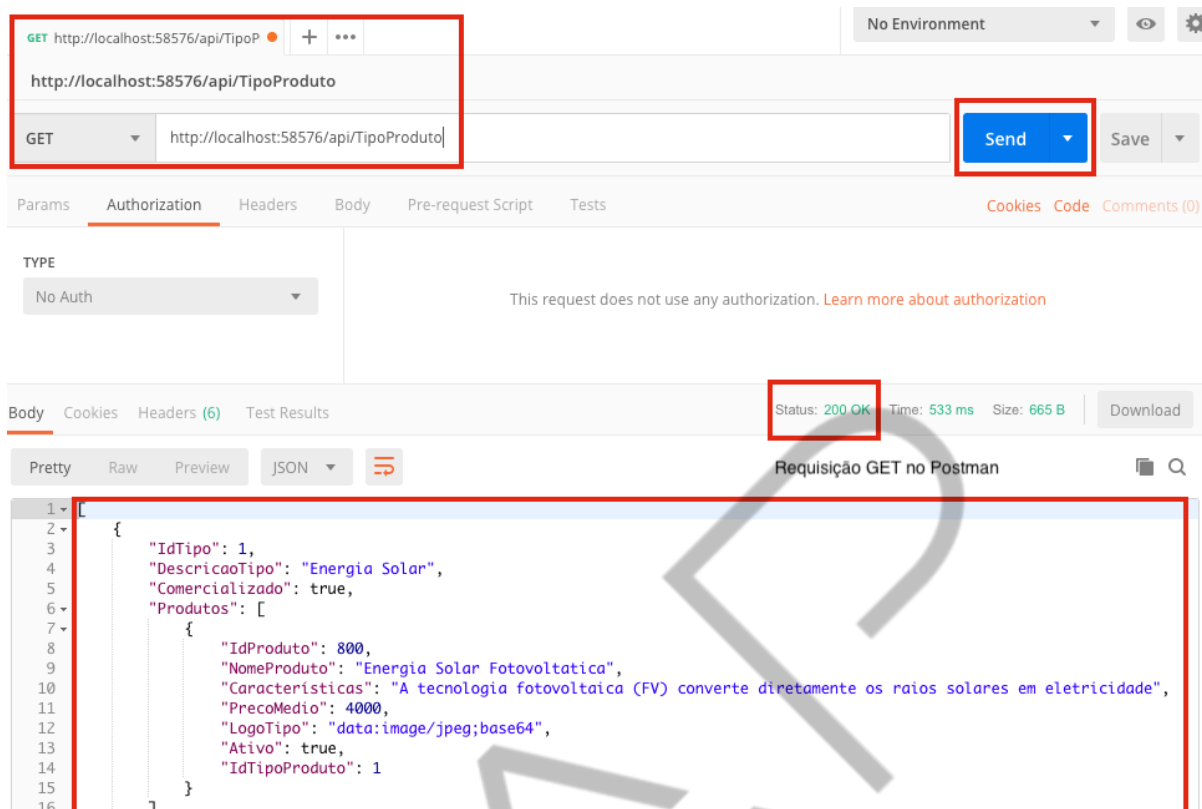


Figura 5.16 – Requisição GET no Postman  
Fonte: Elaborado pelo autor (2018)

Por padrão, o resultado da requisição está em JSON, mas podemos mudar isso, incluindo, no Headers, a opção Accept: **application/Formato** do arquivo (XML, JSON, Javascript, ECMASCRIPT, etc.).

Aproveitando, vamos testar o cenário de falha. Segue exemplo na figura abaixo:

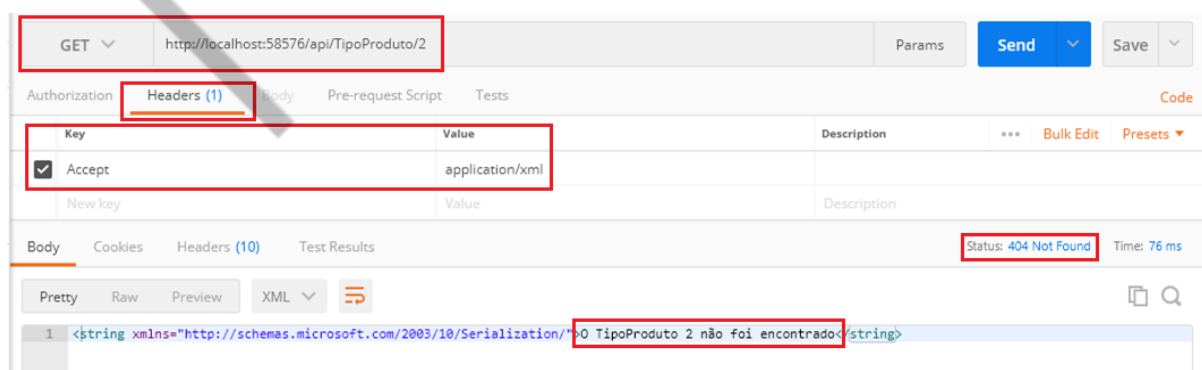


Figura 5.17 – Resposta de dados em formato XML  
Fonte: Elaborado pelo autor (2018)

Agora conseguimos ver todas as informações que nossa ASP.NET Web API está nos fornecendo, mensagem de erro personalizada, **HTTP Status Code: 404 Not Found**, conforme configuramos, e formato da resposta em **XML**, devido à configuração do Accept: application/xml.

Com a requisição **GET** pronta, vamos falar sobre a requisição **POST**.

## 5.22 Requisição POST

O método de requisição POST recebe o conteúdo para ser inserido no corpo da requisição, isso explica a anotação **[FromBody]** como parâmetro no método.

O tipo de retorno na assinatura do método POST continua sendo do tipo **IHttpActionResult**, mas os métodos usados para o retorno em caso de sucesso e falha são outros. Assim que um objeto TipoProduto for adicionado no sistema, o *controller* vai usar o método **Created()** para retornar o **StatusCode 201**, indicando que a informação foi inserida com sucesso.

Para uma falha na inclusão de dados, é utilizado o método **BadRequest()**, retornando ao solicitante o **StatusCode 400**, que indica que as informações estão incompletas ou erradas.

Segue a implementação do método POST, note que o corpo do método consiste em apenas efetuar uma chamada ao DAL e executar o método **Inserir()**, Segue abaixo:

```
public IHttpActionResult Post([FromBody] TipoProduto
TipoProduto)
{
    try
    {
        // Cria o objeto DAL
        TipoProdutoDAL dal = new TipoProdutoDAL();
        // Insere a informação do banco de dados
        dal.Inserir(TipoProduto);

        // Cria uma propriedade para efetuar a consulta
        da informação cadastrada
        string location =
            Url.Link("DefaultApi",
                new { controller = "tipoproduto", id =
TipoProduto.IdTipo });
```



```

        return Created(new Uri(location),
TipoProduto);
    }
    catch (Exception)
    {
        return BadRequest();
    }
}

```

Código-fonte 5.8 – *Controller* Requisição POST  
Fonte: Elaborado pelo autor (2018)

No Postman, temos que passar no corpo (**Body**) o **XML** com os novos dados. Neste caso, iremos cadastrar um novo **TipoProduto** e dois **Produtos** que estarão relacionados ao mesmo.

Segue o exemplo do **XML**:

```

<TipoProduto xmlns:i="http://www.w3.org/2001/XMLSchema-
instance"
xmlns="http://schemas.datacontract.org/2004/07/FiapSmartCityW
ebAPI.Models">
  <Comercializado>true</Comercializado>
  <DescricaoTipo>Filtro de Agua</DescricaoTipo>
  <IdTipo>2</IdTipo>
  <Produtos>
    <Produto>
      <Ativo>true</Ativo>
      <Caracteristicas>
        Possui o sistema de filtragem mais básico, com minerais
      </Caracteristicas>
      <IdProduto>801</IdProduto>
      <IdTipoProduto>1</IdTipoProduto>
      <Logotipo>
        data:image/jpeg;base64
      </Logotipo>
      <NomeProduto>Filtro de barro</NomeProduto>
      <PrecoMedio>200</PrecoMedio>
    </Produto>
    <Produto>
      <Ativo>true</Ativo>
      <Caracteristicas>
        O sistema de entrega de galões já esteve muito em alta
      </Caracteristicas>
      <IdProduto>802</IdProduto>
      <IdTipoProduto>1</IdTipoProduto>
      <Logotipo>
        data:image/jpeg;base64

```

```

</Logotipo>
    <NomeProduto>Galão</NomeProduto>
    <PrecoMedio>20</PrecoMedio>
  </Produto>
</Produtos>
</TipoProduto>

```

Código-fonte 5.9 – XML de dados TipoProduto e Produto  
Fonte: Elaborado pelo autor (2018)

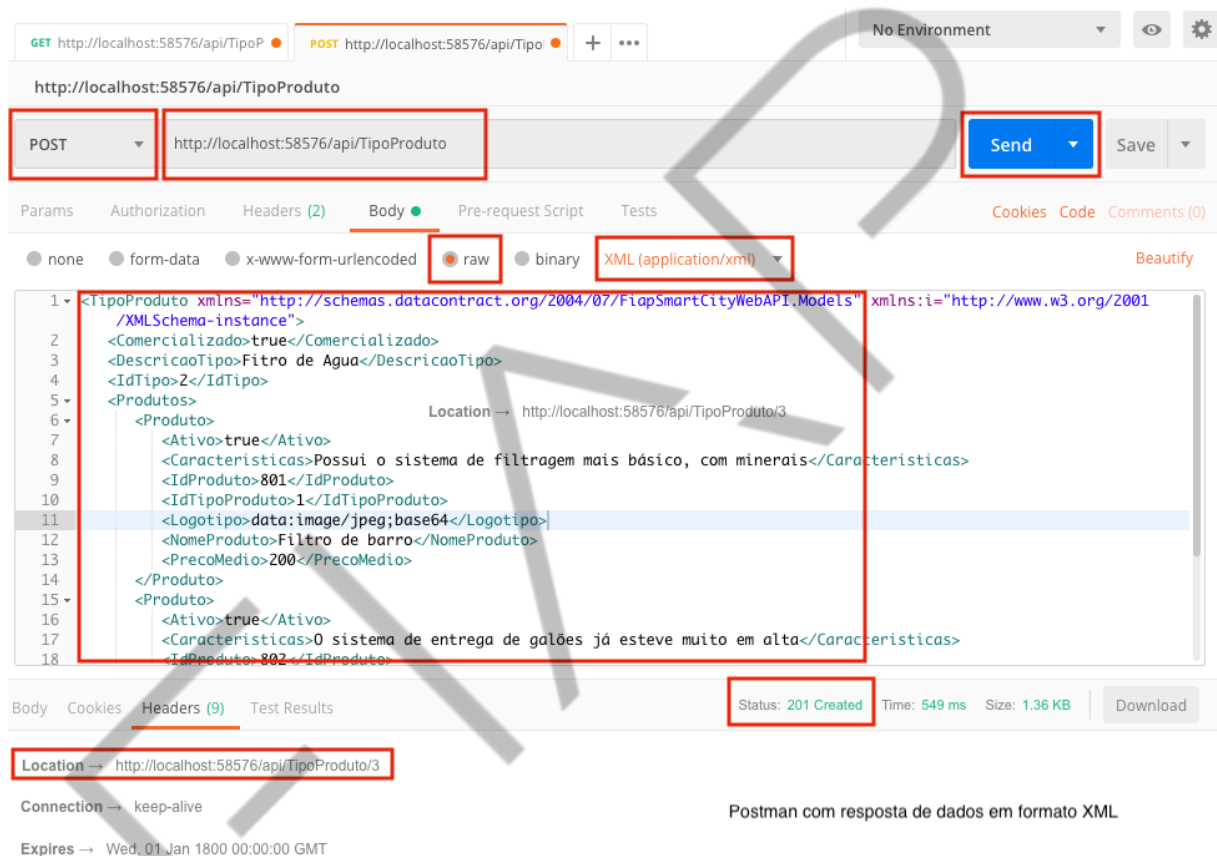


Figura 5.18 – Postman com resposta de dados em formato XML  
Fonte: Elaborado pelo autor (2018)

Como podemos ver, a requisição **POST** se comportou conforme o esperado, retornando o **HTTP Status Code: 201 Created**, a propriedade **location** do **Headers** com o caminho de consulta do recurso, através da requisição **GET** (<http://localhost:58576/api/tipoproduto/3>).

**IMPORTANTE:** A propriedade **Location** é apenas uma forma de consultar rapidamente os dados inseridos. É padrão comum que os desenvolvedores adicionam ao retorno de um método Post de uma API.

### 5.23 Requisição DELETE

A requisição **DELETE**, como o próprio nome diz, irá deletar algum recurso por meio de sua chave ou **id**.

Segue exemplo abaixo, lembrando que a convenção exige que o nome do método seja Delete (), os métodos de retorno são Ok () para o fluxo de sucesso e BadRequest () caso algum erro seja capturado.

Veja:

```
public IHttpActionResult Delete(int id)
{
    try
    {
        TipoProdutoDAL dal = new TipoProdutoDAL();
        dal.Excluir(id);
        return Ok();
    }
    catch (Exception)
    {
        return BadRequest();
    }
}
```

Código-fonte 5.10 – Controller Requisição DELETE  
Fonte: Elaborado pelo autor (2018)

Neste exemplo, excluiremos o **TipoProduto** de **id = 2**, ficando, assim, no Postman:

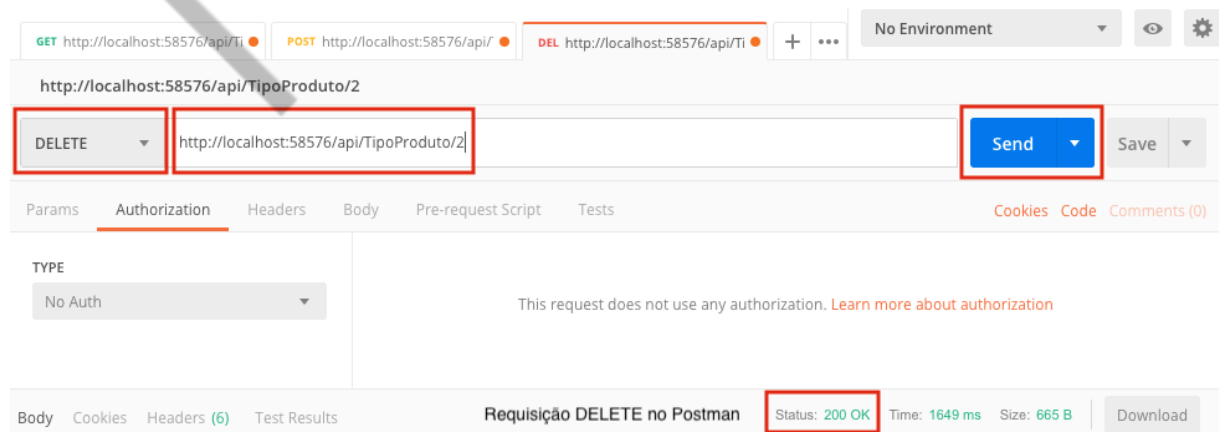


Figura 5.19 – Requisição DELETE no Postman  
Fonte: Elaborado pelo autor (2018)

Seguindo o **HTTP Status Code**, a requisição **DELETE** foi executada com sucesso, para tirarmos a prova, vamos consultar com a requisição **GET** o **TipoProduto** de **idTipoProduto = 2**. Segue abaixo a figura:

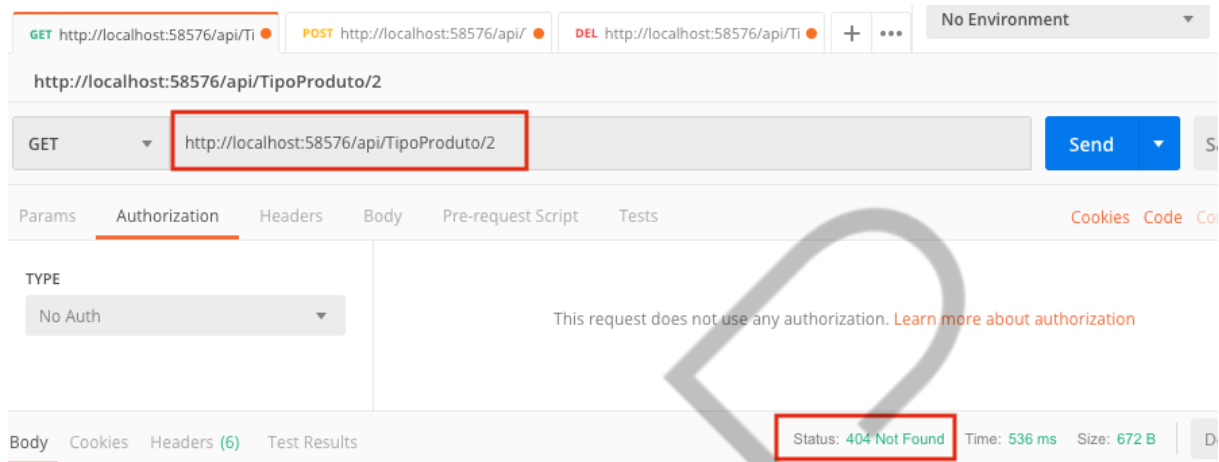


Figura 5.20 – Requisição GET no Postman  
Fonte: Elaborado pelo autor (2018)

## 5.24 Requisição PUT

A requisição **PUT** é a última requisição que iremos abordar neste capítulo. Tendo a função de atualizar dados de um recurso.

Segue código abaixo:

```
public IActionResult Put([FromBody] TipoProduto
tipoProduto)
{
    try
    {
        TipoProdutoDAL dal = new TipoProdutoDAL();
        dal.Alterar(tipoProduto);
        return Ok();
    }
    catch (Exception)
    {
        return BadRequest();
    }
}
```

Código-fonte 5.11 – Controller Requisição PUT  
Fonte: Elaborado pelo autor (2018)

A requisição PUT é muito similar à requisição POST, pois o seu conteúdo precisa ser enviado no corpo da requisição. O único ponto de atenção é que, no conteúdo dos dados enviados no método PUT, é preciso ter o identificador ou Chave Primária que será usado para atualizar o registro correto. O exemplo a seguir irá alterar o TipoProduto com o Id 2, mudando a descrição e a propriedade comercializadas. Seja o XML com os novos dados:

Segue exemplo do XML:

```
<TipoProduto xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/FiapSmartCityWebAPI.Models">
  <Comercializado>false</Comercializado>
  <DescricaoTipo>Filtro d'agua alterado</DescricaoTipo>
  <IdTipo>2</IdTipo>
</TipoProduto>
```

Código-fonte 5.12 – XML de alteração do TipoProduto  
Fonte: Elaborado pelo autor (2018)

Veja o resultado da execução do método PUT na ferramenta Postman:

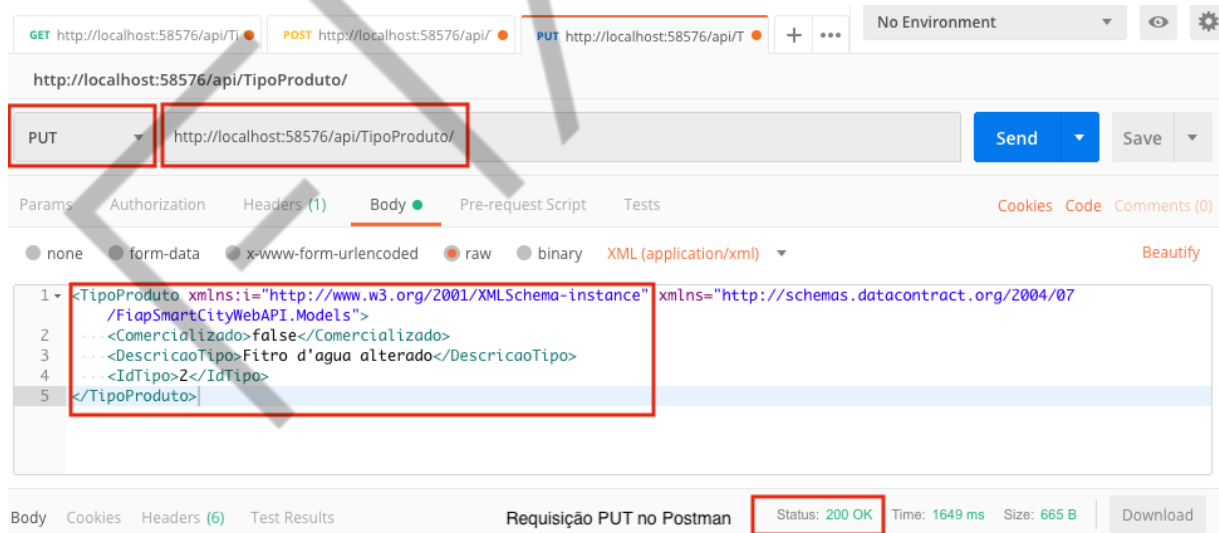


Figura 5.21 – Requisição PUT no Postman  
Fonte: Elaborado pelo autor (2018)

Como sabemos, segundo o **HTTP Status Code 200 OK**, nossa requisição foi executada com sucesso, mas vamos validar executando o método GET. Veja na imagem abaixo:

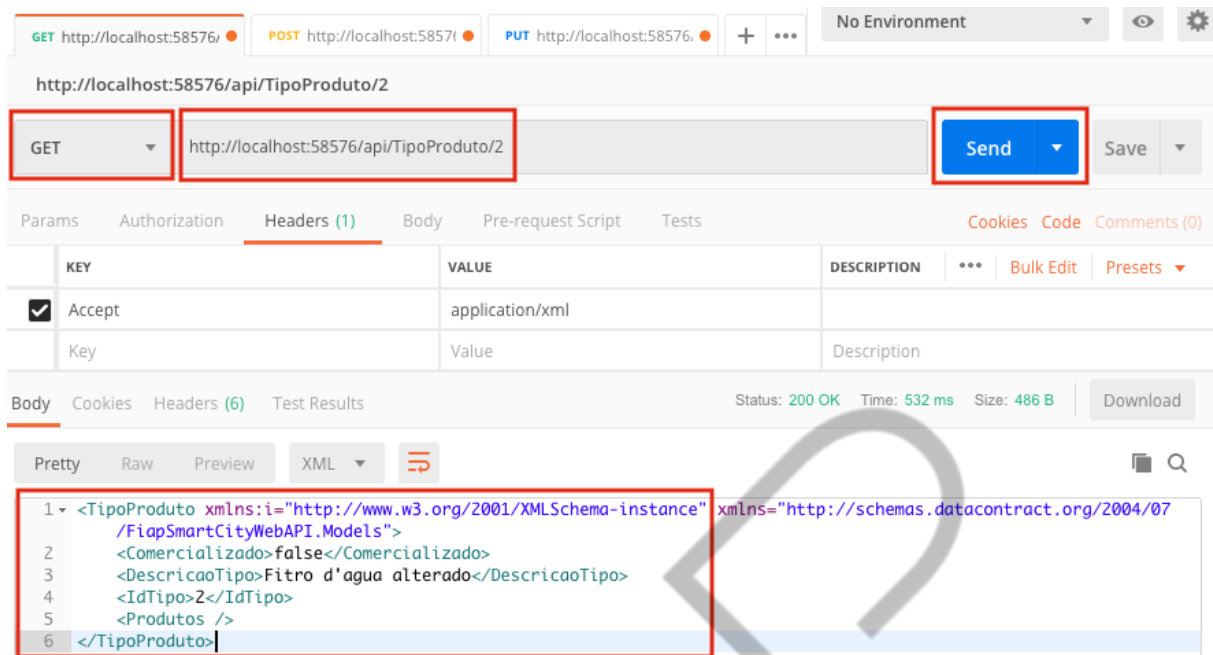


Figura 5.22 – Consultar os dados alterados  
Fonte: Elaborado pelo autor (2018)

É possível notar que os dados foram atualizados com sucesso.

## 5.25 Consumir uma API Rest

### 5.25.1 Padrão de retorno Json

Por padrão, uma **ASP.NET WebAPI** retorna suas respostas com a formatação dos dados **JSON**, assim como muitas das API disponíveis para uso no mercado. O padrão JSON se comporta melhor em alguns cenários, por exemplo, nas requisições **POST** e **PUT**, onde temos que inserir os dados que são cadastrados ou alterados no corpo da requisição, em muitos casos, esses dados apresentam entidades com muitos atributos.

Uma das vantagens de uma **ASP.NET WebAPI** é que, independentemente do formato da entrada dos dados, ela funcionará perfeitamente.

### 5.25.2 Criar a aplicação Cliente

Para esse exemplo, vamos precisar de uma aplicação do tipo Console C#. O nome da nossa aplicação será **FiapSmartCityClient** e terá como objetivo a inserção

dos novos tipos de produtos e a consulta dos tipos cadastrados. Assim, vamos fazer a execução das APIs no método POST e GET, e para as duas execuções vamos trabalhar apenas com dados no formato JSON.

**IMPORTANTE:** Nos exemplos da aplicação *Client*, vamos precisar abrir duas janelas do Visual Studio. A primeira será para abrir e executar o projeto de API, a segunda será usada para codificação e execução do *Client*.

### 5.25.3 Cliente de requisição GET com JSON

A forma mais simples de executar requisições a APIs com C# é usando as classes **System.Net.Http.HttpClient** e **System.Net.Http.HttpResponseMessage**. A classe **HttpClient** é a responsável em criar a conexão com o recurso e executar o método solicitado. A classe **HttpResponseMessage** fica com a responsabilidade de coletar e deixar o conteúdo da resposta disponível para o uso e manipulação.

O exemplo seguinte executará o método GET da nossa API de Tipo de Produto, e com sucesso na execução será exibido o conteúdo JSON com todos os registros cadastrados.

Segue abaixo o código:

```
using System;

namespace FiapSmartCityClient
{
    class Program
    {
        static void Main(string[] args)
        {
            get();
            Console.Read();
        }

        static void get()
        {
            // Criando um objeto Cliente para conectar com o
recurso.
            System.Net.Http.HttpClient client = new
            System.Net.Http.HttpClient();

            // Execute o método Get passando a url da API e
salvando o resultado.
            // em um objeto do tipo HttpResponseMessage

```

```

        System.Net.Http.HttpResponseMessage resposta =
            client.GetAsync("http://localhost:58576/api/TipoProduto").Result;

        // Verifica se o Status Code é 200.
        if (resposta.IsSuccessStatusCode)
        {
            // Recupera o conteúdo JSON retornado pela API
            string conteudo =
                resposta.Content.ReadAsStringAsync().Result;

            // Imprime o conteúdo na janela Console.
            Console.WriteLine(conteudo.ToString());
        }
    }
}

```

Código-fonte 5.13 – Cliente para requisição GET  
Fonte: Elaborado pelo autor (2018)

Segue a janela do aplicativo **client** em execução e a exibição do conteúdo JSON retornado pela API, veja:

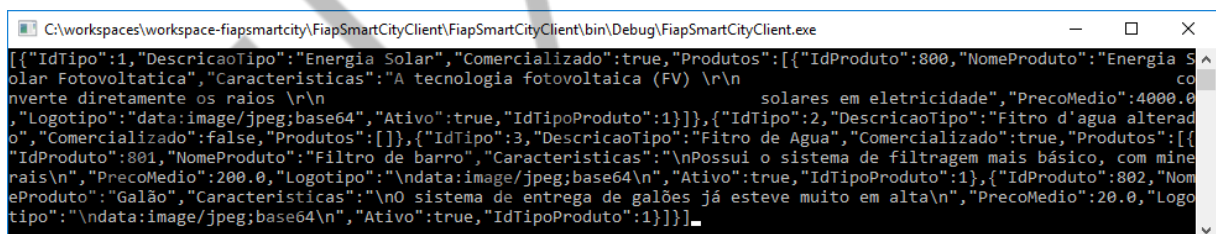


Figura 5.23 – Requisição GET no *Client*  
Fonte: Elaborado pelo autor (2018)

#### 5.25.4 Requisição POST com JSON

Seguindo a mesma linha da requisição GET, vamos usar a classe `HttpClient` para exemplificar uma execução do método POST. Porém, como é sabido, em uma requisição do tipo POST é necessário enviar o conteúdo (JSON) no corpo da mensagem, assim, vamos fazer o uso da classe **`System.Web.Net.StringContent`** para transformar no texto em um conteúdo JSON que será entendido pelo *Controller* WebAPI.



O código abaixo apresenta o exemplo de conversão de um texto em conteúdo JSON e a execução do método POST da **WebAPI** TipoProduto, segue:

```
using System;
using System.Net.Http;
using System.Text;

namespace FiapSmartCityClient
{
    class Program
    {
        static void Main(string[] args)
        {
            post();
            Console.Read();
        }

        static void post()
        {
            // Criando um objeto Cliente para conectar com o
recurso.
            System.Net.Http.HttpClient client = new
            System.Net.Http.HttpClient();

            // Conteúdo do tipo de produto em JSON.
            String json = "{ 'IdTipo':
100, 'DescricaoTipo': 'Robo de
Limpeza', 'Comercializado': true }";

            // Convertendo texto para JSON StringContent.
            StringContent conteudo = new
            StringContent(json.ToString(), Encoding.UTF8,
            "application/json");

            // Execute o método POST passando a url da API
// e envia o conteúdo do tipo StringContent.
            System.Net.Http.HttpResponseMessage resposta =
            client.PostAsync("http://localhost:58576/api/
TipoProduto", conteudo).Result;

            // Verifica que o POST foi executado com sucesso
            if (resposta.IsSuccessStatusCode)
            {
                Console.WriteLine("Tipo do produto criado com
sucesso");
                Console.WriteLine("Link para consulta: " +
resposta.Headers.Location);
            }
        }
    }
}
```

```
}  
}
```

Código-fonte 5.14 – Cliente para requisição POST  
Fonte: Elaborado pelo autor (2018)

Segue também a imagem com o resultado da execução:

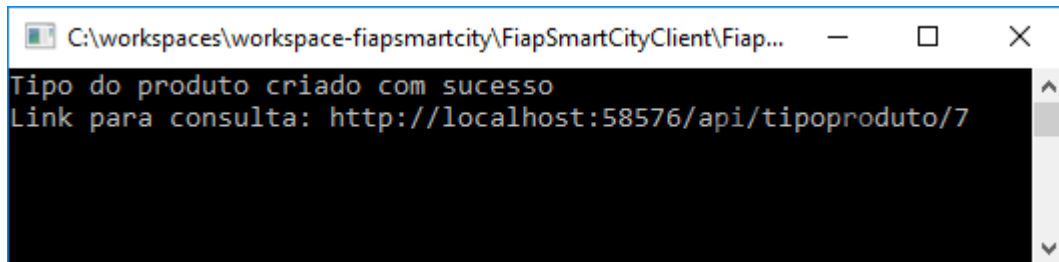


Figura 5.24 – Requisições POST no *Client*  
Fonte: Elaborado pelo autor (2018)

Como podemos ver, o **HTTP Status Code** retornou sucesso, e na segunda linha impressa no resultado temos no valor a propriedade **Location**, isto é, o caminho para consultar os dados do novo **TipoProduto** com uma requisição **GET**.

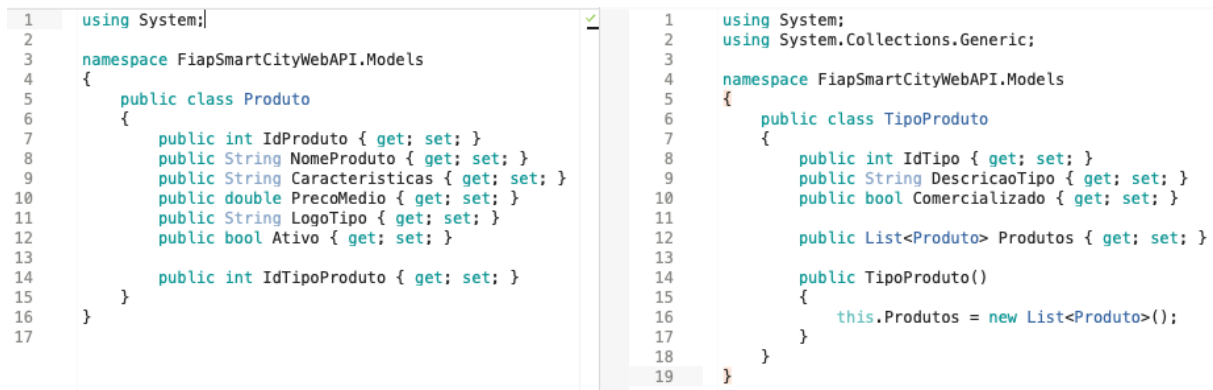
#### 5.25.5 Transformação de dados (Parse)

O conceito de Parse significa transformar dados de diferentes tipos na orientação a objeto, é possível transformar dados string em número, dados numéricos do tipo double ou float em números inteiros e assim uma infinidade de transformações.

Mas qual é a relação entre Parse e WebAPI? Seguindo o texto, será fácil de entender a relação e também a simplicidade que irá proporcionar algumas transformações em nosso client de API.

Antes de apresentar as transformações, vamos criar em nosso projeto **FiapSmartCityClient** o namespace **Models** e adicionar duas classes, **TipoProduto** e **Produto**. Essas classes devem ter os atributos do mesmo tipo e nomes usados nos projetos **FiapSmartCity** (MVC) e **FiapSmartCityWebApi**.

A figura a seguir apresenta o conteúdo das duas classes de modelo:



```

1 using System;
2
3 namespace FiapSmartCityWebAPI.Models
4 {
5     public class Produto
6     {
7         public int IdProduto { get; set; }
8         public String NomeProduto { get; set; }
9         public String Caracteristicas { get; set; }
10        public double PrecoMedio { get; set; }
11        public String LogoTipo { get; set; }
12        public bool Ativo { get; set; }
13
14        public int IdTipoProduto { get; set; }
15    }
16 }
17
1 using System;
2 using System.Collections.Generic;
3
4 namespace FiapSmartCityWebAPI.Models
5 {
6     public class TipoProduto
7     {
8         public int IdTipo { get; set; }
9         public String DescricaoTipo { get; set; }
10        public bool Comercializado { get; set; }
11
12        public List<Produto> Produtos { get; set; }
13
14        public TipoProduto()
15        {
16            this.Produtos = new List<Produto>();
17        }
18    }
19 }

```

Figura 5.25 – Classes de modelo utilizadas para Parse  
Fonte: Elaborado pelo autor (2018)

## 5.26 Desserialização

A palavra **Desserialização** é um pouco estranha, certo? Mas o objetivo desse conceito é bem simples. Desserialização significa transformar conteúdo texto ou JSON em objetos C#.

**IMPORTANTE:** A transformação de texto JSON em objeto pode ser aplicada em outras linguagens, como: Java, C++, Javascript, VB.Net. Assim, esse conceito não é apenas da linguagem C#.

Podemos usar como exemplo a requisição GET da API TipoProduto. Se a requisição for efetuada com sucesso, a API retorna uma lista dos objetos TipoProduto com suas propriedades e valores em formato **texto**, que é composto por um conteúdo JSON. Capturar esse texto e manipular essa informação textual não é um processo muito simples e viável.

Imagine transformar o conteúdo texto em objetos C#. Imaginou?

Pois bem, é exatamente isso que o conceito de desserialização faz para o desenvolvedor. Com ele, é possível transformar texto com conteúdo JSON em objetos C#, por meio da biblioteca **Newtonsoft.Json** e da classe **JsonConvert**.

Antes de iniciar o código, é necessário importar a biblioteca **Newtonsoft.Json** no projeto usando o **Nuget Package Manager**.

Agora, com uma linha de código, é possível transformar o resultado de um método GET em um objeto C#. O exemplo abaixo apresenta o código para recuperar

os dados da WebAPI, desserializar para uma lista de TipoProduto e, por fim, interagir sobre a lista e imprimir os resultados. Segue:

```
using FiapSmartCityClient.Models;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;

namespace FiapSmartCityClient
{
    class Program
    {
        static void Main(string[] args)
        {
            get();
            Console.Read();
        }

        static void get()
        {
            // Criando um objeto Cliente para conectar com o
recurso
            System.Net.Http.HttpClient client = new
            System.Net.Http.HttpClient();

            // Execute o método Get passando a url da API e
salvando o resultado
            // em um objeto do tipo HttpResponseMessage
            System.Net.Http.HttpResponseMessage resposta =
                client.GetAsync("http://localhost:58576/api/T
            ipoProduto").Result;

            // Verifica se o Status Code é 200
            if (resposta.IsSuccessStatusCode)
            {
                // Recupera o conteúdo JSON retornado pela API
                string conteudo =
                resposta.Content.ReadAsStringAsync().Result;

                // Convertendo o conteúdo em uma lista de
TipoProduto
                List<TipoProduto> lista =
                    JsonConvert.DeserializeObject<List<TipoPr
                oduto>>(conteudo);

                // Imprime o conteúdo na janela Console
                foreach(var item in lista)
                {
                    Console.WriteLine("Descrição:" +
                    item.DescricaoTipo);
                }
            }
        }
    }
}
```

```
        Console.WriteLine("Comercializado:" +  
item.Comercializado);  
        Console.WriteLine(" ===== ");  
        Console.WriteLine("");  
    }  
  
    }  
  
    }  
  
}
```

Código-fonte 5.15 – Desserialização JSON em C#  
Fonte: Elaborado pelo autor (2018)

## 5.27 Serialização

Chegou o ponto de executar o processo contrário da desserialização, ou seja, vamos transformar um objeto C# em texto JSON.

O processo de serialização é extremamente útil para métodos POST e PUT, pois agiliza a montagem da string JSON que deverá ser enviada no corpo da requisição. Semelhante ao exemplo de desserialização, vamos usar os mesmos componentes da biblioteca **Newtonsoft.Json**, alterando apenas a chamada para o método de serialização.

Veja o exemplo no código-fonte abaixo:

```
using FiapSmartCityClient.Models;  
using Newtonsoft.Json;  
using System;  
using System.Collections.Generic;  
using System.Net.Http;  
using System.Text;  
  
namespace FiapSmartCityClient  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            //get();  
            post();  
            Console.Read();  
        }  
  
        static void post()  
        {
```

```
// Criando um objeto Cliente para conectar com o
recurso
System.Net.Http.HttpClient client = new
System.Net.Http.HttpClient();

// Conteudo do tipo de produto em JSON
TipoProduto tipo = new TipoProduto();
tipo.IdTipo = 101;
tipo.DescricaoTipo = "Grid de Energia Solar";
tipo.Comercializado = true;

var json = JsonConvert.SerializeObject(tipo);

// Convertendo texto para JSON StringContent
StringContent conteudo = new
StringContent(json.ToString(), Encoding.UTF8,
"application/json");

// Execute o método POST passando a url da API
// e envia o conteudo do tipo StringContent
System.Net.Http.HttpResponseMessage resposta =
client.PostAsync("http://localhost:58576/api/
TipoProduto", conteudo).Result;

// Verifica que o POST foi executado com sucesso
if (resposta.IsSuccessStatusCode)
{
    Console.WriteLine("Tipo do produto criado com
sucesso");
    Console.WriteLine("Link para consulta: " +
resposta.Headers.Location);
}
}

static void get()
{
    // Criando um objeto Cliente para conectar com o
recurso
    System.Net.Http.HttpClient client = new
    System.Net.Http.HttpClient();

    // Execute o método Get passando a url da API e
    salvando o resultado
    // em um objeto do tipo HttpResponseMessage
    System.Net.Http.HttpResponseMessage resposta =
        client.GetAsync("http://localhost:58576/api/T
        ipoProduto").Result;

    // Verifica se o Status Code é 200
    if (resposta.IsSuccessStatusCode)
    {
```

```
// Recupera o conteúdo JSON retornado pela API
string conteudo =
resposta.Content.ReadAsStringAsync().Result;

// Convertendo o conteúdo em uma lista de
TipoProduto
List<TipoProduto> lista =
    JsonConvert.DeserializeObject<List<TipoPr
oduto>>(conteudo);

// Imprime o conteúdo na janela Console
foreach(var item in lista)
{
    Console.WriteLine("Descrição:" +
item.DescricaoTipo);
    Console.WriteLine("Comercializado:" +
item.Comercializado);
    Console.WriteLine(" ===== ");
    Console.WriteLine("");
}
}
}
}
```

Código-fonte 5.16 – Serialização JSON em C#  
Fonte: Elaborado pelo autor (2018)

Note no código-fonte que não temos mais uma variável do tipo string com o conteúdo JSON a ser enviado, agora temos uma instância da classe TipoProduto que será convertida e postada no corpo da requisição POST.

## 5.28 Considerações finais

O capítulo deu início à apresentação dos conceitos de APIs/WebAPIs desenvolvidas com o *framework* ASP.NET WebAPI. Mostrando os conceitos das requisições HTTP e as similaridades com os componentes de aplicativos ASP.NET MVC (Ex.: *Controllers* e *Models*).

Finalizamos com uma aplicação, mostrando seu funcionamento na prática, focando nas operações fundamentais de uma aplicação de Consulta, Criação, Exclusão e Atualização. Também foi apresentada a forma de uso de WebAPI em C#.

para isso, um projeto do tipo Console exemplificou as chamadas de API, as facilidades de trabalho com o formato JSON e as formas de transformações dos dados (Parse).

EXEMPLO



## REFERÊNCIAS

ARAÚJO, E. C. **Orientação a Objetos em C# - Conceitos e implementações em .NET**. São Paulo: Casa do Código, 2010.

\_\_\_\_\_. **ASP.NET MVC5 Crie aplicações web na plataforma Microsoft**. São Paulo: Casa do Código, 2010.

MICROSOFT. **API da Web do ASP.NET** Disponível em: <[https://msdn.microsoft.com/pt-br/library/hh833994\(v=vs.108\).aspx](https://msdn.microsoft.com/pt-br/library/hh833994(v=vs.108).aspx)>. Acesso em: 10 fev. 2018.

\_\_\_\_\_. **HttpClient**. Disponível em: <[https://msdn.microsoft.com/en-us/library/system.net.http.httpclient\(v=vs.118\).aspx](https://msdn.microsoft.com/en-us/library/system.net.http.httpclient(v=vs.118).aspx)>. Acesso em: 23 fev. 2018.

SIÉCOLA, P. **Web Services REST com ASP.NET Web API e Windows Azure**. São Paulo: Casa do Código, 2010.

WASSON, M. **Getting started with ASP.NET Web API 2 (C#)**. Disponível em: <<https://docs.microsoft.com/en-us/aspnet/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api>>. Acesso em: 10 fev. 2018.