

FRAMEWORKS JAVA, .NET &
WEBSERVICES

WEBSERVICES NO **UNIVERSO** **MICROSOFT**



6B

LISTA DE FIGURAS

Figura 1 – Usuário utilizando o Internet Banking	7
Figura 2 – Composição de URL e URN	12
Figura 3 – Projeto ASP.NET Web	14
Figura 4 – Nome do projeto	15
Figura 5 – <i>Target Framework</i>	15
Figura 6 – Estrutura de projeto ASP.NET WebAPI	16
Figura 7 – Diagrama de classe Cliente e Representante	17
Figura 8 – Instalando o Oracle.EntityFrameworkCore	20
Figura 9 – Classe DataBaseContext	21
Figura 10 – Adicionar <i>Controller</i>	27
Figura 11 – Selecionar o <i>Scaffold</i> do <i>Controller</i>	27
Figura 12 – Detalhes de um <i>Controller</i>	28
Figura 13 – Endereço e resultado da execução	30
Figura 14 – Caminho de configuração da rota para o controller	30
Figura 16 – Erro tratado na requisição GET	33
Figura 17 – Requisição GET no Postman	35
Figura 18 – Postman com resposta de dados em formato JSON	37
Figura 19 – Requisição DELETE no Postman	39
Figura 20 – Requisição PUT no Postman	41
Figura 21 – Requisição GET no <i>Client</i>	45
Figura 22 – Requisições POST no <i>Client</i>	46
Figura 23 – Classes de modelo utilizadas para Parse	47

LISTA DE QUADROS

Quadro 1 – Exemplos de URI e verbos HTTP.....	13
---	----

EXEMPLO

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Modelo Representante.....	18
Código-fonte 2 – Modelo Cliente	19
Código-fonte 3 – appsettings.json criando String de conexão Oracle	21
Código-fonte 4 – Implementação DataBaseContext	22
Código-fonte 5 – Configurando e instanciando o DataBaseContext	22
Código-fonte 6 – Classe RepresentanteRepository	24
Código-fonte 7 – Classe ClienteRepository.....	26
Código-fonte 8 – <i>Controller</i> Requisição Get.....	29
Código-fonte 9 – <i>Controller</i> implementar o retorno uniforme	32
Código-fonte 10 – Método Get para listar todos os dados	35
Código-fonte 11 – <i>Controller</i> Requisição POST	36
Código-fonte 12 – JSON de dados Representante	37
Código-fonte 13 – <i>Controller</i> Requisição DELETE.....	38
Código-fonte 14 – <i>Controller</i> Requisição PUT.....	40
Código-fonte 15 – JSON de alteração do Representante	40
Código-fonte 16 – Cliente para requisição GET	44
Código-fonte 17 – Cliente para requisição POST.....	46
Código-fonte 18 – Desserialização JSON em C#.....	49
Código-fonte 19 – Serialização JSON em C#	50

SUMÁRIO

1 WEBSERVICES NO UNIVERSO MICROSOFT.....	6
1.1 Web Services	6
1.2 Um pouco sobre API	9
1.3 Fundamentos	9
1.4 Protocolo HTTP	10
1.5 URL	10
1.6 URN	10
1.7 URI	11
2 URL, URN, URI	12
2.1 Verbos HTTP.....	12
2.2 HTTP Status Code	13
2.3 Criar o projeto.....	14
2.4 Modelos.....	16
2.5 Funcionalidades	19
2.5.1 Repository ou DAL	20
2.6 Controllers.....	26
2.7 Requisição GET	28
3 MELHORAR A REQUISIÇÃO GET	31
3.2 Interface uniforme	31
3.3 Get – Listar os dados	33
3.4 Usar o Postman.....	35
3.5 Requisição POST.....	36
3.6 Requisição DELETE.....	38
3.7 Requisição PUT	39
4 CONSUMIR UMA API REST.....	43
4.1 Padrão de retorno JSON	43
4.2 Criar a aplicação Cliente	43
4.3 Cliente de requisição GET com JSON	43
4.3.1 Requisição POST com JSON.....	45
4.3.2 Transformação de dados (Parse).....	47
4.3.3 Desserialização	47
4.4 Serialização.....	49
CONCLUSÃO.....	52
REFERÊNCIAS.....	53

1 WEBSERVICES NO UNIVERSO MICROSOFT

Após passarmos pelas arquiteturas MVC e ORM do .NET, descobriremos a importância dos *web services*, que são os 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 corrida pelo Uber ou pede comida no iFood? Quem faz essa ligação entre o aplicativo e o sistema do fornecedor é o Web Service!

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

1.1 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. Empresas como IBM, Microsoft e HP e entre outras participantes do consórcio apoiaram a criação dos padrões que são usados na 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 de fevereiro de 2004)

Ou seja, um sistema de software projetado para dar suporte à interoperabilidade entre máquinas em uma rede. Em linhas gerais, podemos dizer que Web Services são aplicações desenvolvidas para integrar sistemas que se comunicam com pouco 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. Nesse contexto, o aplicativo 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 1 – Usuário utilizando o Internet Banking
Fonte: 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 aplicativo 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 aplicativo 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 Internet Banking 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 (aplicativo), também utilizando o protocolo HTTPS. Ao receber a comunicação, o aplicativo 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 aplicativo.

É 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. Esses componentes retornarão o saldo para que o Web Service apenas crie a mensagem e atenda ao cliente que consumiu o serviço.

Esses 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 caixa eletrônico 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. Você se lembra do acoplamento fraco sobre o qual tanto comentamos?

Nesse exemplo, utilizamos a tecnologia de Web Services para atender ao canal Mobile e empregamos os conceitos da Arquitetura Orientada a Serviços para atender a esse 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 aplicativo desconhece se o Web Service foi codificado em Java EE, .NET ou PHP, e o Web Service não se preocupa se o aplicativo foi desenvolvido em Java Android, Swift ou alguma linguagem híbrida.

Isso acontece porque, tanto na requisição como 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, e passou a compor a definição de Web Services do W3C em meados de 2004.

1.2 Um pouco sobre API

Uma API ou WebAPI é uma evolução de um WebService e ignora os detalhes de implementação e a 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 (por exemplo: recursos para Produtos, recursos para Clientes), diferentemente dos webservices SOAP, cujo foco são as operações (por exemplo: 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 é a quantidade de serviços usados em aplicativos móveis. Eles precisam ter a complexidade cada vez mais simplificada e 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 utilizadas por parceiros e usuários dos serviços.

O ASP.NET WEB API utiliza HTTP com REST.

1.3 Fundamentos

Como podemos notar, no tópico anterior, falamos sobre Web Services, o surgimento das novas tecnologias e a diferença entre SOAP e REST, até chegarmos ao **ASP.NET WEB API**. Mas temos mais fundamentos que são essenciais para o funcionamento de tudo isso, os mais comuns serão explicados nos tópicos a seguir.

1.4 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 máquinas com diferentes configurações comunicarem-se 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.

1.5 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: internet ou rede corporativa de 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

1.6 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

1.7 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..35i39k1j0i131k1j0l3j0i67k1j0l4.1769.2212.0.2379.4.4.0.0.0.86.332.4.4.0....0...1.1.64.psy-ab..0.4.329...0i131i67k1.0.G8Vp2Tigdhk

2 URL, URN, URI

Todas essas definições podem ter deixado você confuso, então, explicaremos 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/**).

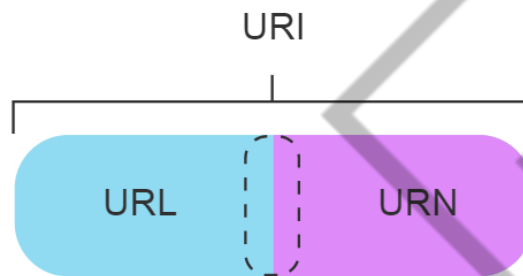


Figura 2 – Composição de URL e URN
Fonte: Google Imagens (2018)

2.1 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 façamos a requisição, o resultado será o mesmo.
- 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.
- PATCH - O método PATCH é utilizado para aplicar modificações parciais em um recurso.

- HEAD - O método HEAD solicita uma resposta de forma idêntica ao método GET, porém, sem conter o corpo da resposta.

Veja alguns exemplos no Quadro “Exemplos de URI e verbos HTTP”:

Endpoint (caminho)	Método	Ação
/api/cliente/{ClientId}	GET	Retorna a lista de clientes cadastrados.
/api/cliente/	POST	Usado para inserir um novo recurso de cliente no sistema.
/api/cliente/{ClientId}	PUT	Alterar um recurso de cliente já existente.
/api/cliente/{ClientId}	DELETE	Remover um cliente existente.

Quadro 1 – Exemplos de URI e verbos HTTP
Fonte: Elaborado pelo autor (2022)

2.2 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.

2.3 Criar o projeto

Para iniciar a criação de um novo serviço ASP.NET WEB API, seguiremos o mesmo modelo de negócio dos capítulos anteriores: a **Fiap.Web.AspNet**.

Vamos colocar a mão na massa?

No Visual Studio 2022, selecione o menu **File > New > Project** (a tecla de atalho **Ctrl + Shift + N**). O template de projeto será o **ASP.NET Core Web API**. Para nosso exemplo, usaremos **Fiap.Api.AspNet** como nome do projeto e da solução.

A Figura “Projeto ASP.NET Core Web API” apresenta os passos para a seleção da linguagem e o tipo de projeto.

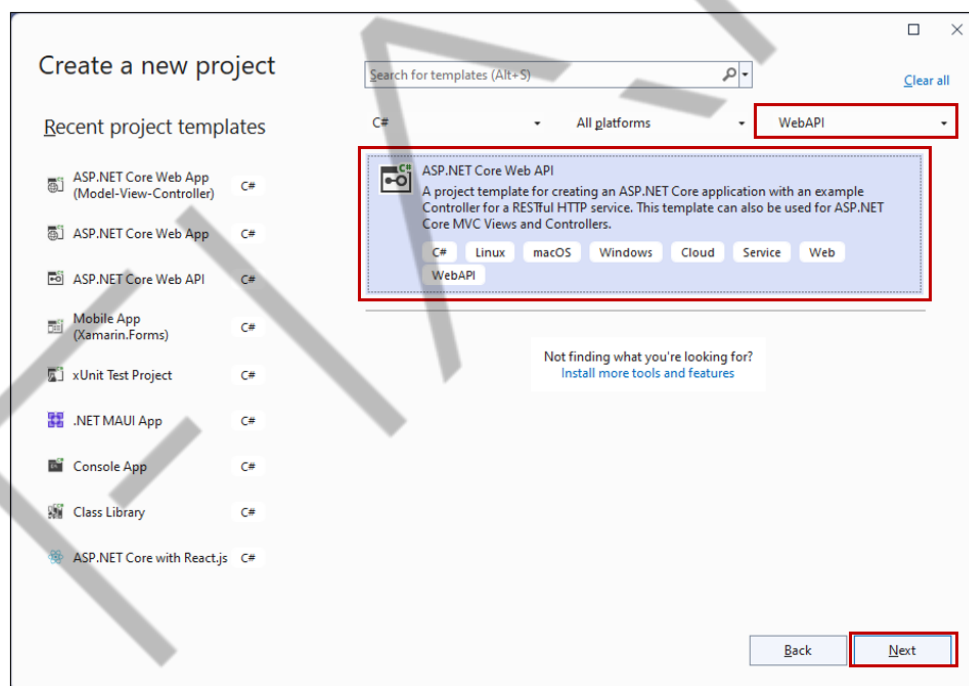
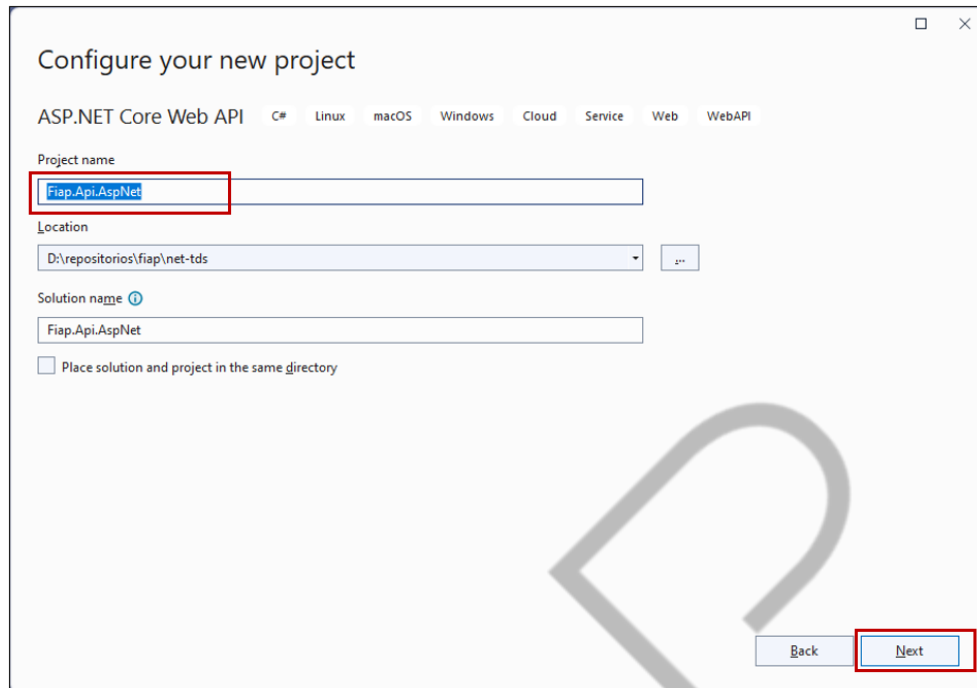


Figura 3 – Projeto ASP.NET Web
Fonte: Elaborado pelo autor (2022)

Na janela seguinte, devemos inserir o nome do projeto “**Fiap.Api.AspNet**” e a pasta do seu computador para salvar o projeto.



Configure your new project

ASP.NET Core Web API C# Linux macOS Windows Cloud Service Web WebAPI

Project name
Fiap.Api.AspNet

Location
D:\repositorios\fiap\net-tds

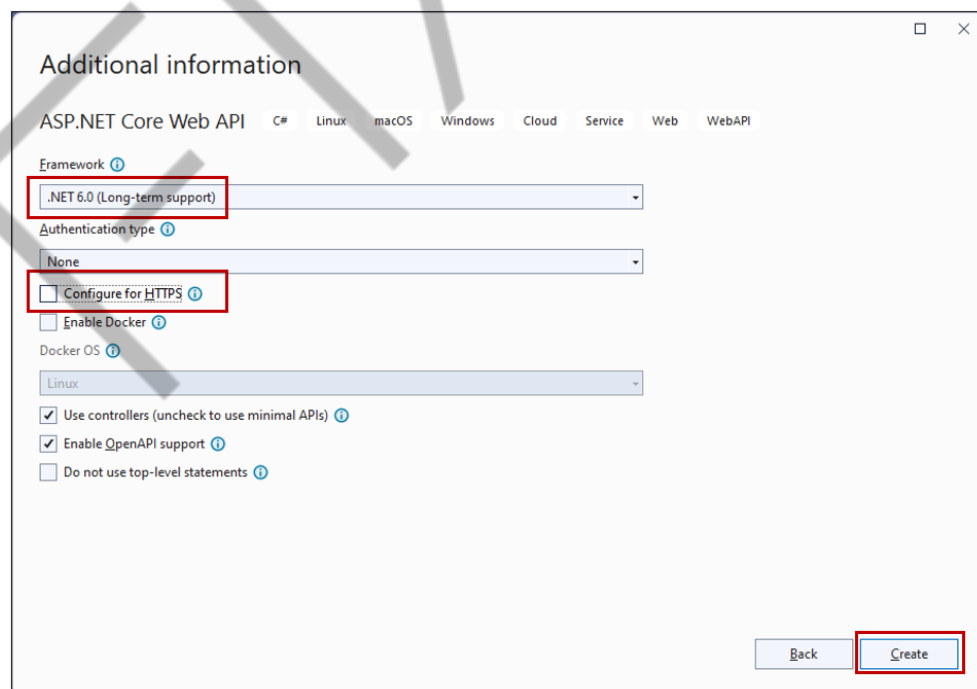
Solution name
Fiap.Api.AspNet

☐ Place solution and project in the same directory

Back Next

Figura 4 – Nome do projeto
Fonte: Elaborado pelo autor (2022)

Após o nome do projeto, escolha a versão do Target Framework (“.NET Core 6.0”) e desmarque a opção “Configure for Https”.



Additional information

ASP.NET Core Web API C# Linux macOS Windows Cloud Service Web WebAPI

Framework
.NET 6.0 (Long-term support)

Authentication type
None

☐ Configure for HTTPS

☐ Enable Docker

Docker OS
Linux

☒ Use controllers (unchecked to use minimal APIs)

☒ Enable OpenAPI support

☐ Do not use top-level statements

Back Create

Figura 5 – Target Framework
Fonte: Elaborado pelo autor (2022)

Finalizada 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.

IMPORTANTE: A partir do .NET 6.x a classe **Startup.cs** não é mais gerada por padrão. O motivo é simples: na nova versão, o código foi migrado e centralizado na classe **Program.cs**.

Observe a Figura “Estrutura de projeto ASP.NET WebAPI”:

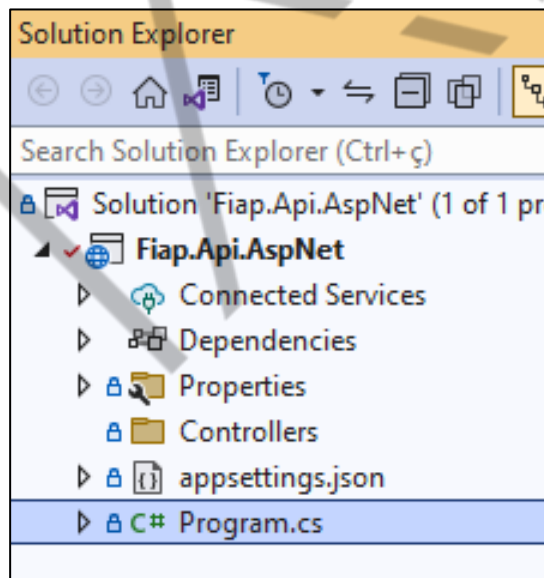


Figura 6 – Estrutura de projeto ASP.NET WebAPI
Fonte: Elaborado pelo autor (2022)

2.4 Modelos

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

Neste primeiro momento, começaremos pela camada de modelos, na qual criaremos a estrutura dos nossos dados. Em seguida, uma camada de acesso a dados e, para finalizar, nossos controladores.

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

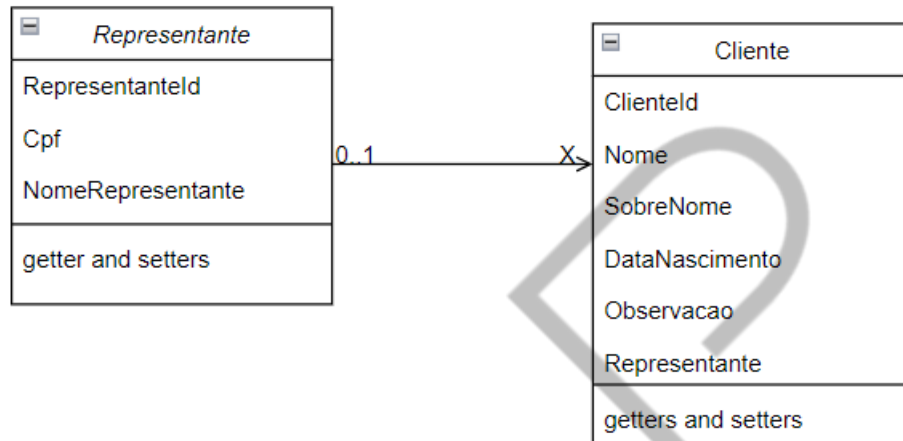


Figura 7 – Diagrama de classe Cliente e Representante
Fonte: Elaborado pelo autor (2022)

Os modelos devem ser adicionados no *namespace* Models do projeto. Para criar o modelo **Cliente** e **Representante**, clique com o botão direito na pasta **Models** e escolha a opção **Add > Class**. Defina os nomes como RepresentanteModel.cs e ClienteModel.cs, utilize o Diagrama de Classe e adicione os atributos com seus respectivos tipos. Seguem os códigos com as classes de modelos implementadas.

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using System.Xml.Linq;

namespace Fiap.Api.AspNet.Models
{
    [Table("REPRESENTANTE")]
    public class RepresentanteModel
    {
        [Key]
        [Column("REPRESENTANTEID")]
        public int Representanteld { get; set; }

        [Column("NOMEREPRESENTANTE")]
        [Required(ErrorMessage = "Nome do representante é obrigatório!")]
        [StringLength(80,
            MinimumLength = 2,
            ErrorMessage = "O nome deve ter, no mínimo, 2 e, no máximo, 80 caracteres")]
        [Display(Name = "Nome do Representante")]
    }
}
```

WebServices no universo Microsoft

```
public string? NomeRepresentante { get; set; }

[Column("CPF")]
[Required(ErrorMessage = "CPF é obrigatório!")]
[Display(Name = "CPF")]
public string? Cpf { get; set; }

// Essa anotação é apenas um exemplo de um propriedade não mapeada em uma coluna do banco de dados
[NotMapped]
public string? Token { get; set; }

//Navigation Property
public IList<ClienteModel> Clientes { get; set; }

public RepresentanteModel()
{
}

public RepresentanteModel(int representanteld, string nomeRepresentante)
{
    Representanteld = representanteld;
    NomeRepresentante = nomeRepresentante;
}

public RepresentanteModel(int representanteld, string cpf, string nomeRepresentante)
{
    Representanteld = representanteld;
    Cpf = cpf;
    NomeRepresentante = nomeRepresentante;
}
}
```

Código-fonte 1 – Modelo Representante
Fonte: Elaborado pelo autor (2022)

```
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using System.Xml.Linq;

namespace Fiap.Api.AspNet.Models
{
    [Table("CLIENTE")]
    public class ClienteModel
    {
        [HiddenInput]
        [Key]
        [Column("CLIENTEID")]
        public int Clienteld { get; set; }
    }
}
```

```
[Display(Name = "Nome do Cliente")]
[Required(ErrorMessage = "O nome do cliente é obrigatório")]
[MaxLength(70, ErrorMessage = "O tamanho máximo para o campo nome é de 70 caracteres.")]
[MinLength(2, ErrorMessage = "Digite um nome com 2 ou mais caracteres")]
[Column("NOME")]
public string? Nome { get; set; }

[Display(Name = "Data de Nascimento")]
[Required(ErrorMessage = "Data de nascimento é obrigatório")]
[DataType(DataType.Date, ErrorMessage = "Data de nascimento incorreta")]
[Column("DATANASCIMENTO")]
public DateTime DataNascimento { get; set; }

[Display(Name = "Observação")]
[Column("OBSERVACAO")]
public string? Observacao { get; set; }

//Campo de chave Estrangeira - Foreign Key (FK)
[Display(Name = "Representante")]
[Column("REPRESENTANTEID")]
public int Representanteld { get; set; }

//Objeto - Navigation Object
public RepresentativeModel? Representante { get; set; }

}
}
```

Código-fonte 2 – Modelo Cliente
Fonte: Elaborado pelo autor (2022)

2.5 Funcionalidades

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

2.5.1 Repository ou DAL

Data Access Layer (Camada 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.

Neste exemplo de ASP.NET WebAPI, trabalharemos com **Representante** e **Clientes**, assim, vamos precisar de duas classes de repositório. Porém, como já vimos anteriormente que é muito mais simples manipular nossos dados de banco de dados usando o ORM Entity Framework, vamos baixar as bibliotecas do EF e do Oracle, configurar a *string* de conexão com o banco de dados e criar nossa classe de **Contexto**.

Seguem as imagens e código-fonte com os trechos relevantes de configuração:

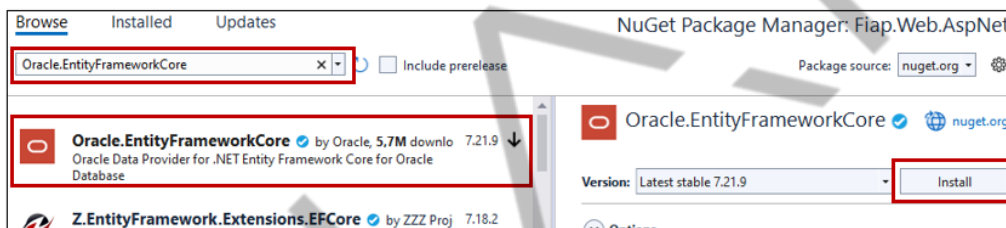


Figura 8 – Instalando o Oracle.EntityFrameworkCore
Fonte: Elaborado pelo autor (2022)

Abra o arquivo appsettings.json (raiz do projeto) e acrescente a configuração da String de conexão para acesso ao banco de dados Oracle da Fiap. O nome da string de conexão será **DatabaseConnection**.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DatabaseConnection": "Data Source=(DESCRIPTION = (ADDRESS_LIST = (ADDRESS = (PROTOCOL = TCP)(HOST = oracle.fiap.com.br)(PORT = 1521)))(CONNECT_DATA = (SID = orcl)));Persist Security Info=True;User ID=rm9999;Password=senha;Pooling=True;Connection Timeout=60;"
  }
}
```

```
}
```

Código-fonte 3 – appsettings.json criando String de conexão Oracle
Fonte: Elaborado pelo autor (2022)

IMPORTANTE: É preciso trocar os dados de conexão de acordo com o local do seu banco de dados Oracle, usuário e senha. Para o arquivo appsettings.json, é obrigatório seguir algumas estruturas para a declaração das *strings* de conexão e da configuração da fonte de dados. Por isso, evite trocar o posicionamento dessas seções.

Dentro do *namespace Repository*, adicione uma pasta com o nome **Context** e, em seguida, adicione uma classe com o nome de **DataBaseContext**. Veja a Figura “Classe DataBaseContext”:

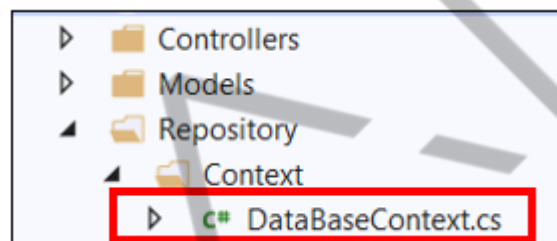


Figura 9 – Classe DataBaseContext
Fonte: Elaborado pelo autor (2022)

O Código-Fonte “Implementação DataBaseContext” apresenta a implementação da classe **DataBaseContext**:

```
using Fiap.Api.AspNet.Models;
using Microsoft.EntityFrameworkCore;

namespace Fiap.Api.AspNet.Repository.Context
{
    public class DataBaseContext : DbContext
    {
        // Propriedade que será responsável pelo acesso a tabela de Representantes
        public DbSet<RepresentanteModel> Representante { get; set; }

        // Propriedade que será responsável pelo acesso a tabela de Cliente
        public DbSet<ClienteModel> Cliente { get; set; }

        public DataBaseContext(DbContextOptions options) : base(options)
        {
        }
    }
}
```

```
protected DbContext()
{
}

}
```

Código-fonte 4 – Implementação DbContext
Fonte: Elaborado pelo autor (2022)

Configurando o banco de dados utilizado no Projeto e qual será a classe de contexto para nossa aplicação. O Código-Fonte “Configurando e instanciando o DbContext” demonstrar a implementação na classe Program.cs:

```
using Fiap.Api.AspNet.Repository.Context;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
var connectionString = builder.Configuration.GetConnectionString("DatabaseConnection");
builder.Services.AddDbContext<DbContext>(options =>
    options.UseOracle(connectionString).EnableSensitiveDataLogging(true)
);

builder.Services.AddControllers();
// 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.UseAuthorization();

app.MapControllers();

app.Run();
```

Código-fonte 5 – Configurando e instanciando o DbContext
Fonte: Elaborado pelo autor (2022)

Chegou a hora de implementarmos nossos repositórios, segue abaixo trechos de código da implementação do RepresentaRepository e ClienteRepository que devem ser adicionados ao projeto.

WebServices no universo Microsoft

```
using Fiap.Api.AspNet.Models;
using Fiap.Api.AspNet.Repository.Context;
using Microsoft.EntityFrameworkCore;

namespace Fiap.Api.AspNet.Repository
{
    public class RepresentanteRepository
    {
        private readonly DataBaseContext dataBaseContext;

        public RepresentanteRepository(DataBaseContext ctx)
        {
            dataBaseContext = ctx;
        }

        public IList<RepresentanteModel> Listar()
        {
            var lista = new List<RepresentanteModel>();
            lista = dataBaseContext.Representante.ToList<RepresentanteModel>();
            return lista;
        }

        public IList<RepresentanteModel> ListarRepresentantesComClientes()
        {
            var lista = new List<RepresentanteModel>();

            lista = dataBaseContext.Representante
                .Include(r => r.Clientes)
                .ToList<RepresentanteModel>();

            return lista;
        }

        public IList<RepresentanteModel> ListarOrdenadoPorNome()
        {
            var lista = new List<RepresentanteModel>();

            lista = dataBaseContext
                .Representante
                .OrderBy(r => r.NomeRepresentante).ToList<RepresentanteModel>();

            return lista;
        }

        public IList<RepresentanteModel> ListarOrdenadoPorNomeDescendente()
        {
            var lista = new List<RepresentanteModel>();

            lista = dataBaseContext
                .Representante
                .OrderByDescending(r => r.NomeRepresentante).ToList<RepresentanteModel>();

            return lista;
        }
    }
}
```

```
}

public RepresentanteModel ConsultarPorCpf(String cpf)
{
    var representante = dataBaseContext.Representante.
        Where(r => r.Cpf == cpf).
       FirstOrDefault<RepresentanteModel>();

    return representante;
}

public RepresentanteModel ConsultarPorParteNome(String nomeParcial)
{
    var representante = dataBaseContext.Representante.
        Where(r => r.NomeRepresentante.Contains(nomeParcial)).
       FirstOrDefault<RepresentanteModel>();

    return representante;
}

public RepresentanteModel Consultar(int id)
{
    var representante = dataBaseContext.Representante.Find(id);

    return representante;
}

public void Inserir(RepresentanteModel representante)
{
    dataBaseContext.Representante.Add(representante);
    dataBaseContext.SaveChanges();
}

public void Alterar(RepresentanteModel representante)
{
    dataBaseContext.Representante.Update(representante);
    dataBaseContext.SaveChanges();
}

public void Excluir(int id)
{
    var representante = new RepresentanteModel(id, "");

    dataBaseContext.Representante.Remove(representante);
    dataBaseContext.SaveChanges();
}
}
```

Código-fonte 6 – Classe RepresentanteRepository
Fonte: Elaborado pelo autor (2022)

WebServices no universo Microsoft

```
using Fiap.Api.AspNet.Models;
using Fiap.Api.AspNet.Repository.Context;
using Microsoft.EntityFrameworkCore;

namespace Fiap.Api.AspNet.Repository
{
    public class ClienteRepository
    {
        private readonly DataBaseContext dataBaseContext;

        public ClienteRepository(DataBaseContext ctx)
        {
            dataBaseContext = ctx;
        }

        public IList<ClienteModel> Listar()
        {
            var lista = dataBaseContext.Cliente
                .Include(c => c.Representante)
                .ToList<ClienteModel>();

            return lista;
        }

        public ClienteModel Consultar(int id)
        {
            var cliente = new ClienteModel();

            cliente = dataBaseContext.Cliente
                .Include(c => c.Representante)
                .Where(c => c.ClientId == id)
                .FirstOrDefault();

            return cliente;
        }

        public void Inserir(ClienteModel cliente)
        {
            dataBaseContext.Cliente.Add(cliente);
            dataBaseContext.SaveChanges();
        }

        public void Alterar(ClienteModel cliente)
        {
            dataBaseContext.Cliente.Update(cliente);
            dataBaseContext.SaveChanges();
        }
    }
}
```

```
public void Excluir(int id)
{
    var cliente = new ClienteModel { ClientId = id };

    dataBaseContext.Cliente.Remove(cliente);
    dataBaseContext.SaveChanges();

}

}
```

Código-fonte 7 – Classe ClienteRepository
Fonte: Elaborado pelo autor (2022)

DICA: Os componentes Repository podem ser usados no projeto Fiap.Web.AspNet (MVC e Entity Framework). Basta baixar as bibliotecas do EF e configurar o acesso ao banco de dados no projeto de WebAPI.

Você pode fazer o download do projeto configura com as etapas anteriores nos links abaixo:

Git: <https://github.com/FIAPON/Fiap.Api.AspNet/tree/config>

Zip: <https://github.com/FIAPON/Fiap.Api.AspNet/archive/refs/heads/config.zip>

2.6 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 nosso *Controller*.

Com um clique do botão direito na pasta *Controllers* do projeto, selecione a opção **Add > Controller**, como mostra a Figura “Adicionar Controller”. O Visual Studio apresentará a janela **Add Scaffold**, selecione, então, a opção API do lado esquerdo da janela e a opção “**API Controller – Empty**”, conforme a Figura “Selecionar o Scaffold do Controller”.

WebServices no universo Microsoft

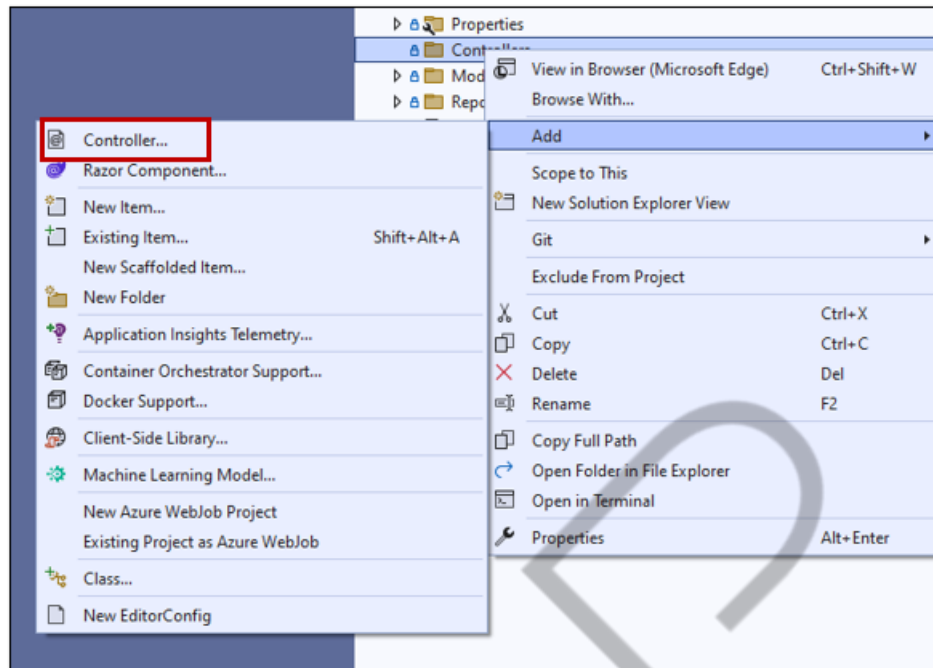


Figura 10 – Adicionar *Controller*
Fonte: Elaborado pelo autor (20122)

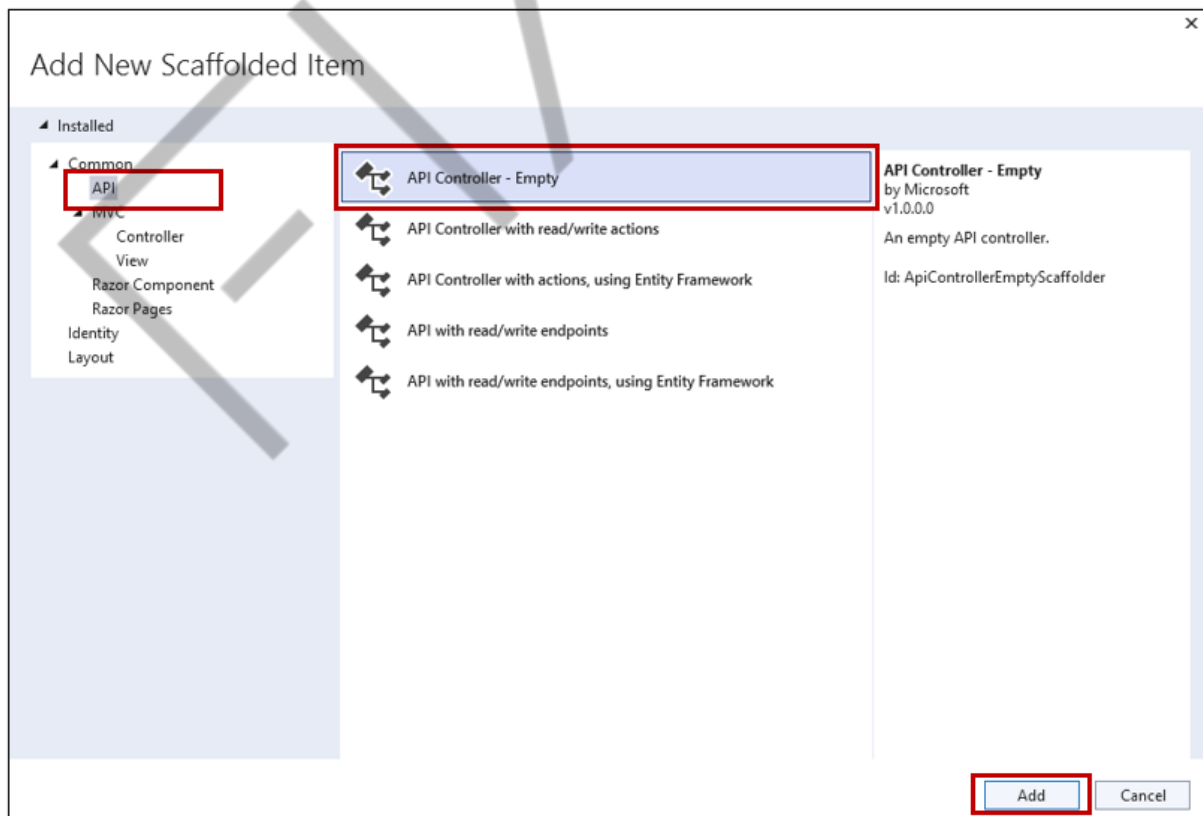


Figura 11 – Selecionar o *Scaffold* do *Controller*
Fonte: Elaborado pelo autor (2022)

O próximo passo é definir o nome do controlador, que será **RepresentanteController** em nosso projeto. Clique no botão Add e aguarde a criação. Lembre-se, todo controlador 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-fonte da classe Controller, é possível ver a importação do *namespace Microsoft.AspNetCore.Mvc* e a extensão da classe **ControllerBase**.

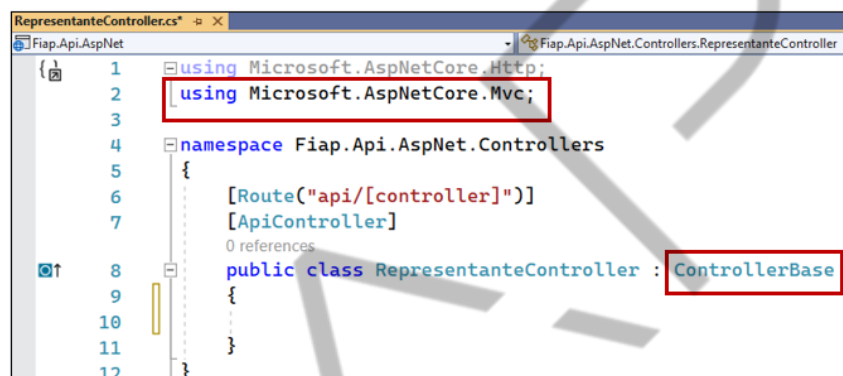


Figura 12 – Detalhes de um Controller
Fonte: Elaborado pelo autor (2022)

2.7 Requisição GET

Com nosso *Controller* criado, elaboraremos nossa primeira requisição, baseada nos **Verbos HTTP**: a requisição **GET**.

Nosso método GET será implementado capturando o Id para o Representante e, depois, consultando nossa camada dos dados com o Id capturado e retornando um objeto Representante. Por convenção, o nome do nosso método será Get(). Veja o Código-Fonte “Controller Requisição Get” abaixo:

```
using Fiap.Api.AspNet.Models;
using Fiap.Api.AspNet.Repository;
using Fiap.Api.AspNet.Repository.Context;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Fiap.Api.AspNet.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
```

```
public class RepresentanteController : ControllerBase
{
    private readonly RepresentanteRepository representanteRepository;

    public RepresentanteController(DataBaseContext context)
    {
        representanteRepository = new RepresentanteRepository(context);
    }

    [HttpGet("{id:int}")]
    public ActionResult<RepresentanteModel> Get([FromRoute] int id)
    {
        try
        {
            var representanteModel = representanteRepository.Consultar(id);
            return Ok(representanteModel);
        }
        catch (KeyNotFoundException e)
        {
            throw e;
        }
    }
}
```

Código-fonte 8 – *Controller* Requisição Get
Fonte: Elaborado pelo autor (2022)

Com o *Controller* criado e requisição **GET** elaborada, podemos, então, fazer o primeiro teste. Pressione a tecla **F5** e aguarde o navegador-padrão do seu computador ser aberto. Com o navegador aberto, digite o endereço `http://localhost:5179/api/Representante/1` na url para efetuar a pesquisa, e veja o resultado da consulta na janela abaixo. (Obs: a porta disponibilizada pelo Asp.NET Core foi 5179 para esse exemplo, verifique a porta disponibilizada na sua execução):

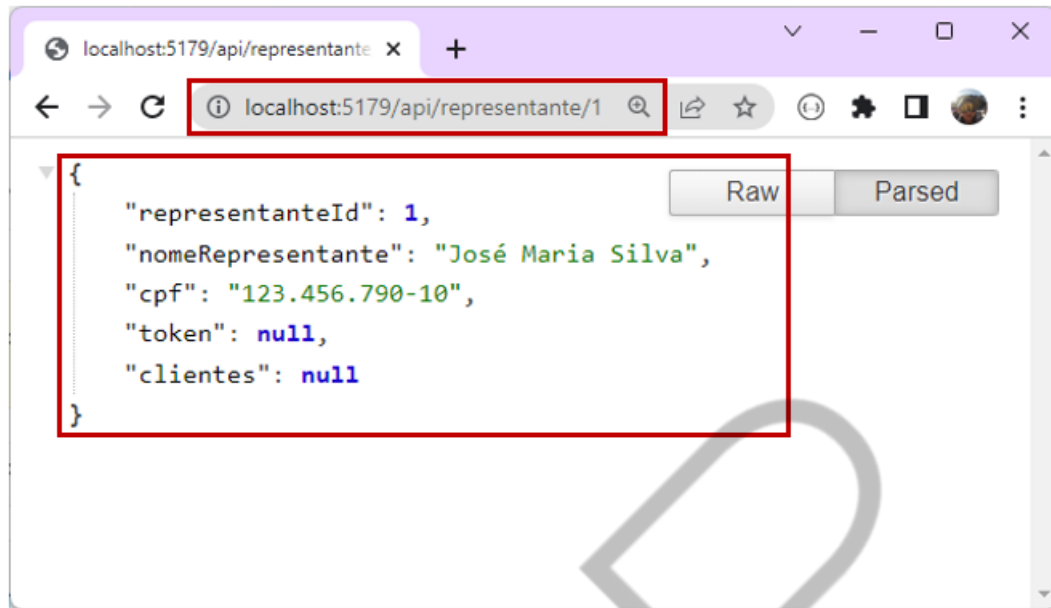


Figura 13 – Endereço e resultado da execução
Fonte: Elaborado pelo autor (2022)

Para entender um pouco sobre como é definido a URL da nossa API, a figura abaixo mostra a classe `RepresentanteController` e a anotação `Route` responsável pelo mapeamento da nossa URL.

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Fiap.Api.AspNet.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class RepresentanteController : ControllerBase
    {
        private readonly RepresentanteRepository representanteRepository;

        public RepresentanteController(DataBaseContext context)
        {
        }
    }
}
```

Figura 14 – Caminho de configuração da rota para o controller
Fonte: Elaborado pelo autor (2022)

15IMPORTANTE: No endereço `http://localhost:5179/api/Representante/1`, o número final (1) indica qual é o código do Representante que consultaremos. Lembre-se que a porta é gerada aleatoriamente.

3 MELHORAR A REQUISIÇÃO GET

E se executarmos uma requisição GET no nosso projeto, passando um **id do representante**, nosso retorno não está adequado, pois nesse momento não retornamos nenhuma mensagem de erro ou falha. Para melhorar esse fluxo, vamos ajustar o retorno usando um conceito no ASP.NET Core WebApi chamado de interface uniforme.

3.2 Interface uniforme

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

Brincadeiras à parte, interface uniforme nada mais é do que um retorno unificado. Diferente de um modelo único de 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 em prática, usaremos como retorno dos métodos o tipo **ActionResult**, que nada mais é do que uma *factory* de respostas **HTTP** usando **HttpResponseMessage**.

Veja o Código-Fonte “*Controller* implementar o retorno uniforme” abaixo:

```
using Fiap.Api.AspNet.Models;
using Fiap.Api.AspNet.Repository;
using Fiap.Api.AspNet.Repository.Context;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Fiap.Api.AspNet.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class RepresentanteController : ControllerBase
    {
        private readonly RepresentanteRepository representanteRepository;

        public RepresentanteController(DataBaseContext context)
        {
            representanteRepository = new RepresentanteRepository(context);
        }
    }
}
```

```
[HttpGet("{id:int}")]
public ActionResult<RepresentanteModel> Get([FromRoute] int id)
{
    try
    {
        var representanteModel = representanteRepository.Consultar(id);

        if (representanteModel != null )
        {
            return Ok(representanteModel);
        } else
        {
            return NotFound();
        }
    }
    catch (Exception e)
    {
        return StatusCode(StatusCode.Status500InternalServerError);
    }
}
```

Código-fonte 9 – *Controller* implementar o retorno uniforme
Fonte: Elaborado pelo autor (2022)

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

Caso o fluxo para a consulta de um Representante seja executado com sucesso, o método **Ok ()** devolverá o objeto Representante 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?

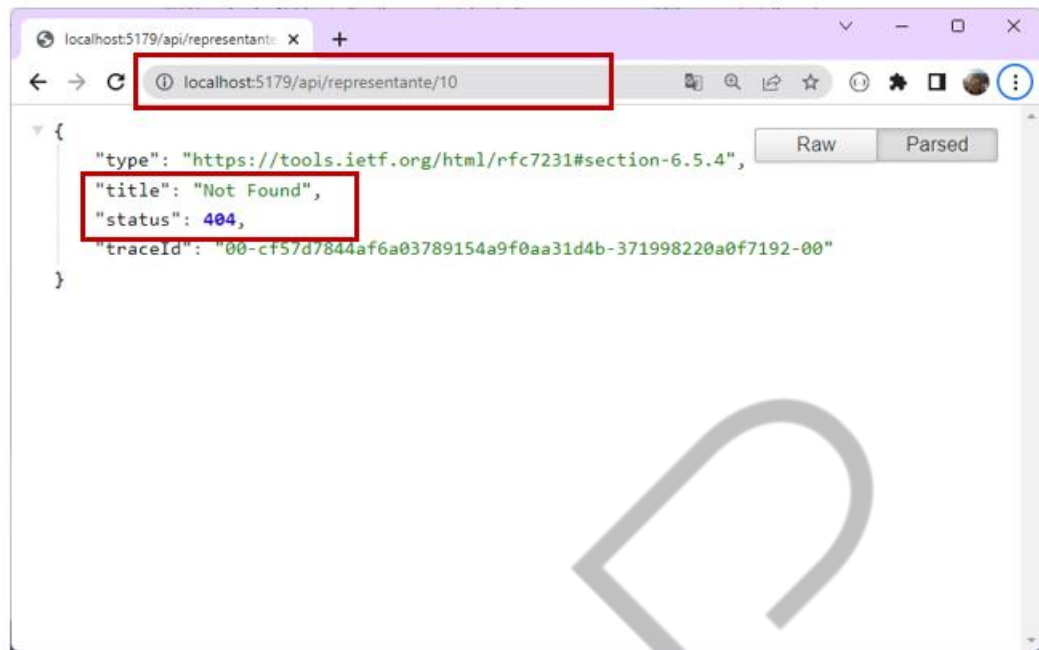


Figura 15 – Erro tratado na requisição GET
Fonte: Elaborado pelo autor (2022)

A requisição retornou o HTTP Status Code 404 esperado, conforme podemos ver na Figura “Erro tratado na requisição GET”.

3.3 Get – Listar os dados

O nosso *Controller* Representantes possui um método Get () que recebe o Id como parâmetro e consulta a informação de um determinado representante. Podemos ter mais um método Get (), porém, é obrigatório que a assinatura seja diferente. Implementaremos um método Get () sem nenhum parâmetro e seu objeto será o retorno de uma lista de representantes. Veja o Código-Fonte “Método Get para listar todos os dados” abaixo:

```
using Fiap.Api.AspNet.Models;
using Fiap.Api.AspNet.Repository;
using Fiap.Api.AspNet.Repository.Context;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Fiap.Api.AspNet.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class RepresentanteController : ControllerBase
    {
```

```
private readonly RepresentanteRepository representanteRepository;

public RepresentanteController(DataBaseContext context)
{
    representanteRepository = new RepresentanteRepository(context);
}

[HttpGet]
public ActionResult<List<RepresentanteModel>> Get()
{
    try
    {
        var lista = representanteRepository.Listar();

        if (lista != null)
        {
            return Ok(lista);
        }
        else
        {
            return NotFound();
        }
    }
    catch (Exception e)
    {
        return StatusCode(StatusCode.Status500InternalServerError);
    }
}

[HttpGet("{id:int}")]
public ActionResult<RepresentanteModel> Get([FromRoute] int id)
{
    try
    {
        var representanteModel = representanteRepository.Consultar(id);

        if (representanteModel != null )
        {
            return Ok(representanteModel);
        }
        else
        {
            return NotFound();
        }
    }
    catch (Exception e)
    {
        return StatusCode(StatusCode.Status500InternalServerError);
    }
}
```

```
}  
}
```

Código-fonte 10 – Método Get para listar todos os dados
Fonte: Elaborado pelo autor (2022)

3.4 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, simularemos uma requisição de sucesso e poderemos ver, na aba **Body**, o resultado no corpo da requisição, e o **Status: 200 OK**, conforme nossa configuração.

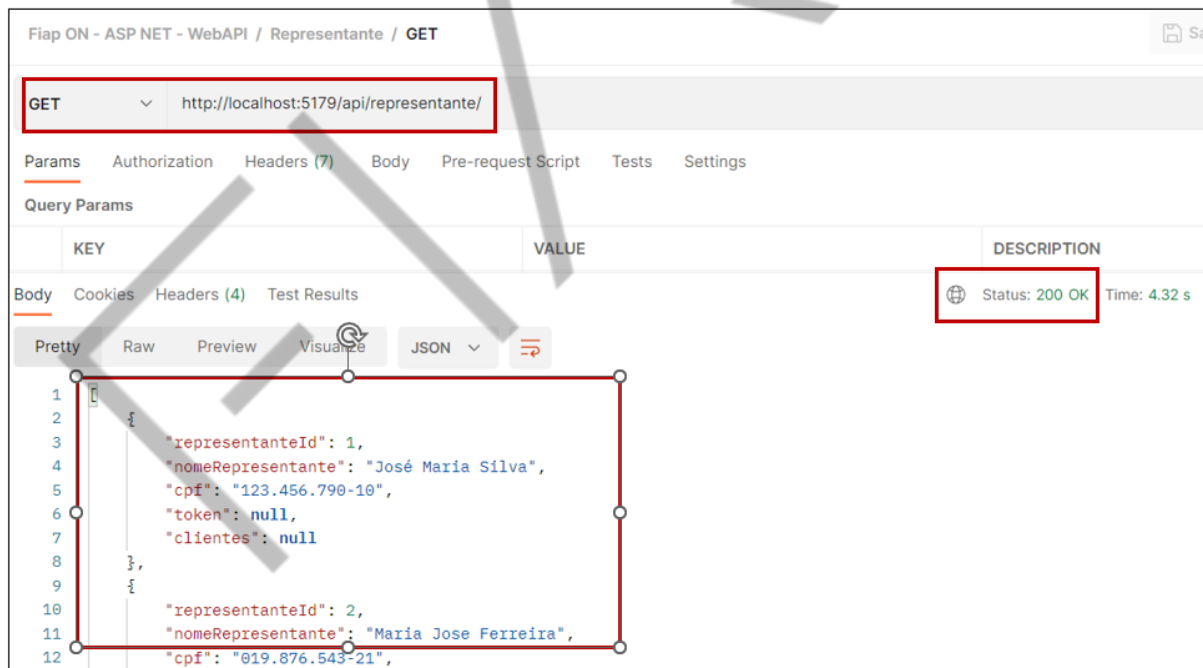


Figura 16 – Requisição GET no Postman
Fonte: Elaborado pelo autor (2022)

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.).

3.5 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 Representante 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 Repository e executar o método **Inserir()**. Veja o Código-Fonte “Controller Requisição POST” abaixo:

```
[HttpPost]
public ActionResult<RepresentanteModel> Post([FromBody] RepresentanteModel
representanteModel)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    try
    {
        representanteRepository.Inserir(representanteModel);
        var location = new Uri(Request.GetEncodedUrl() + "/" + representanteModel.RepresentanteId);
        return Created(location, representanteModel);
    }
    catch (Exception error)
    {
        return BadRequest(new { message = $"Não foi possível o Representante. Detalhes:
{error.Message}" });
    }
}
```

Código-fonte 11 – Controller Requisição POST
Fonte: Elaborado pelo autor (2022)

No Postman, temos que passar no corpo (**Body**) o JSON com os novos dados. Neste caso, vamos cadastrar um novo **Representante** e dois **Produtos** que estarão relacionados a ele.

Segue o exemplo do **JSON**:

```
{
  "nomeRepresentante": "Sergio Manuel",
  "cpf": "123.456.790-99",
  "token": null
}
```

Código-fonte 12 – JSON de dados Representante
Fonte: Elaborado pelo autor (2022)

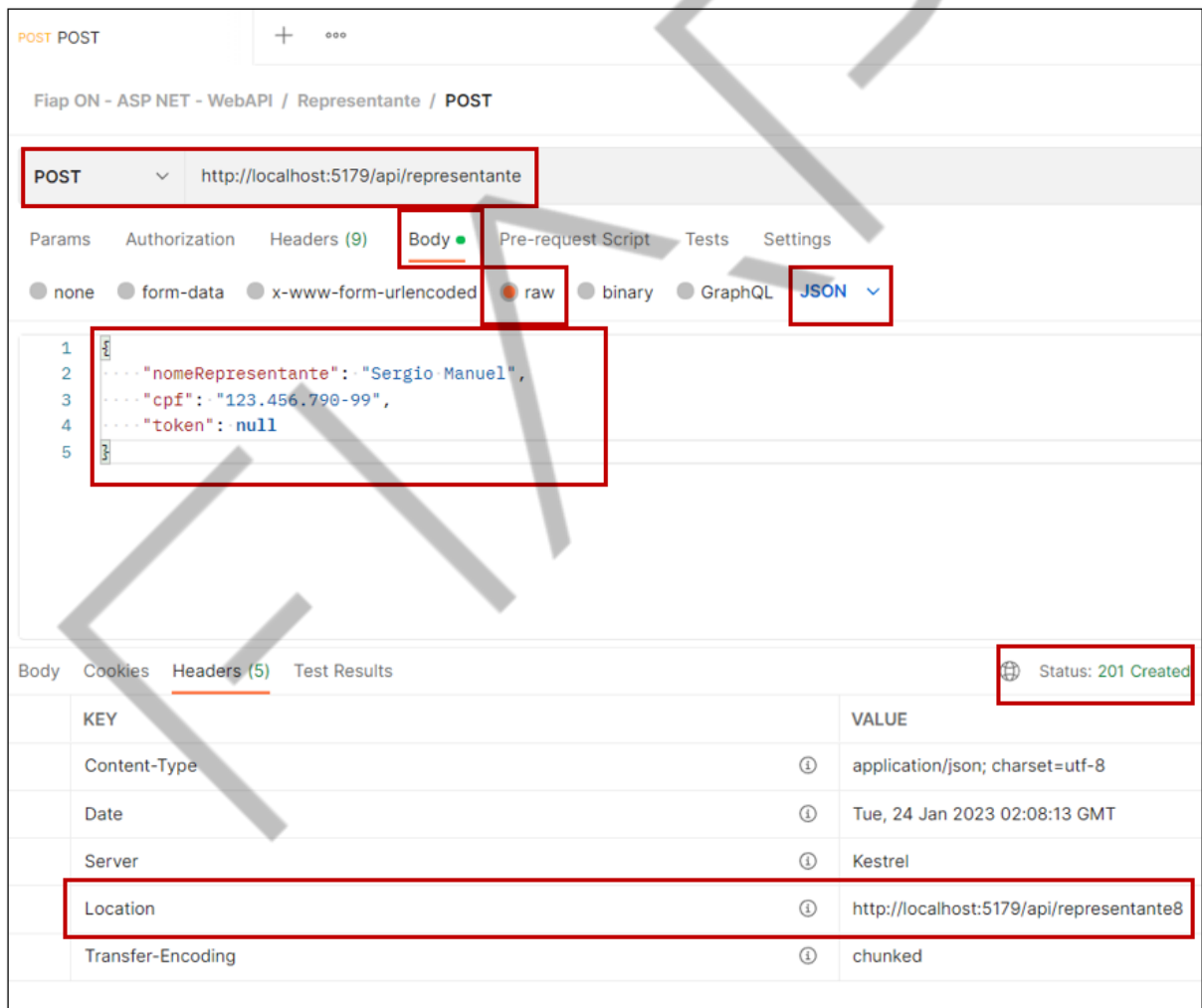


Figura 17 – Postman com resposta de dados em formato JSON
Fonte: Elaborado pelo autor (2022)

Como podemos ver, a requisição **POST** comportou-se conforme o esperado, retornando o **HTTP Status Code: 201 Created**, a propriedade **location** do **Headers**

com o caminho de consulta do recurso por meio da requisição **GET** <http://localhost:5179/api/Representante/8>.

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

3.6 Requisição DELETE

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

Veja o exemplo no Código-Fonte “*Controller* Requisição DELETE” abaixo, lembrando que a convenção exige que o nome do método seja Delete(), os métodos de retorno são NoContent() para o fluxo de sucesso, e BadRequest() ou NotFound(), caso algum erro seja capturado.

Veja:

```
[HttpDelete("{id:int}")]
public ActionResult<RepresentanteModel> Delete([FromRoute] int id)
{
    try {
        var representanteModel = representanteRepository.Consultar(id);

        if (representanteModel != null)
        {
            representanteRepository.Excluir(id);
            // Retorno Sucesso.
            // Efetuou a exclusão, porém sem necessidade de informar os dados.
            return NoContent();
        } else
        {
            return NotFound();
        }
    } catch (Exception e)
    {
        return BadRequest();
    }
}
```

Código-fonte 13 – *Controller* Requisição DELETE
Fonte: Elaborado pelo autor (2022)

Neste exemplo, excluiríamos o **Representante** de **id = 8**, ficando assim no Postman:

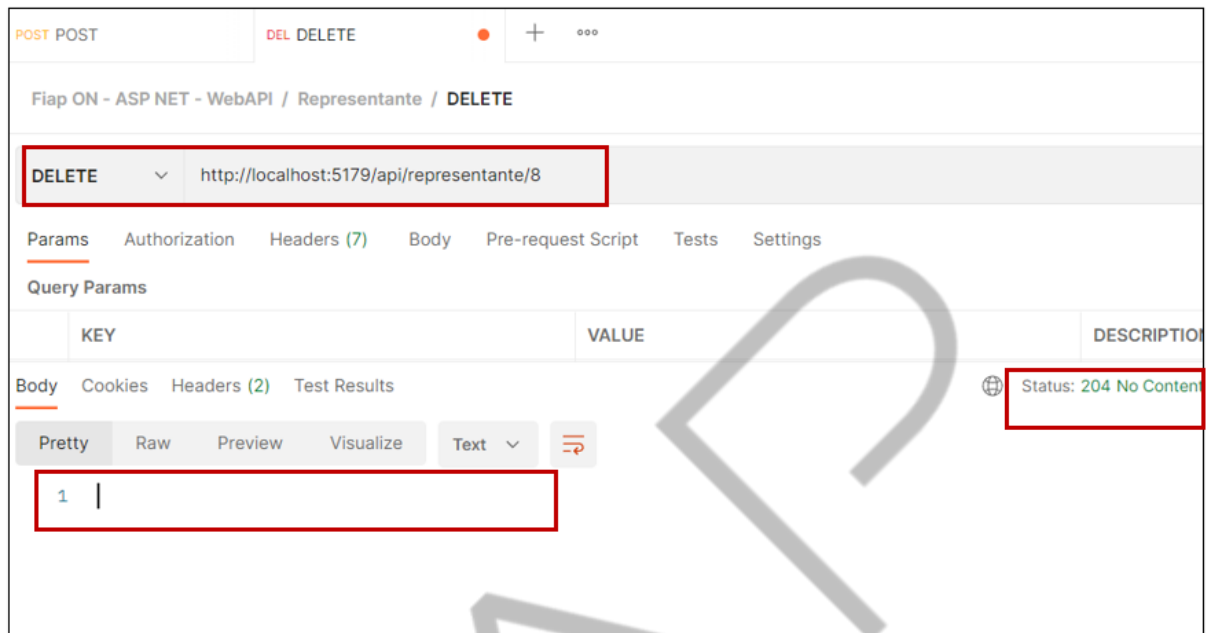


Figura 18 – Requisição DELETE no Postman

Fonte: Elaborado pelo autor (2022)

19

3.7 Requisição PUT

A requisição **PUT** tem a função de atualizar dados de um recurso e é a última requisição que abordaremos neste capítulo.

Veja o Código-Fonte “*Controller* Requisição PUT” abaixo:

```
[HttpPut("{id:int}")]
public ActionResult<RepresentanteModel> Put([FromRoute] int id, [FromBody] RepresentanteModel representanteModel)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    if (representanteModel.RepresentantId != id)
    {
        return NotFound();
    }

    try
    {

```

```
representanteRepository.Alterar(representanteModel);  
return NoContent();  
}  
catch (Exception error)  
{  
    return BadRequest(new { message = $"Não foi possível alterar Representante. Detalhes:  
{error.Message}" });  
}  
}
```

Código-fonte 14 – *Controller* Requisição PUT
Fonte: Elaborado pelo autor (2022)

A requisição PUT é muito similar à requisição POST, pois 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 a Chave Primária que será usada para atualizar o registro correto. O exemplo a seguir vai alterar o Representante com o Id 2. Veja, no Código-Fonte “JSON de alteração do Representante”, o JSON com os novos dados:

```
{  
    "representanteId": 1,  
    "nomeRepresentante": "José Maria Souza",  
    "cpf": "123.456.790-33",  
    "token": null  
}
```

Código-fonte 15 – JSON de alteração do Representante
Fonte: Elaborado pelo autor (2022)

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

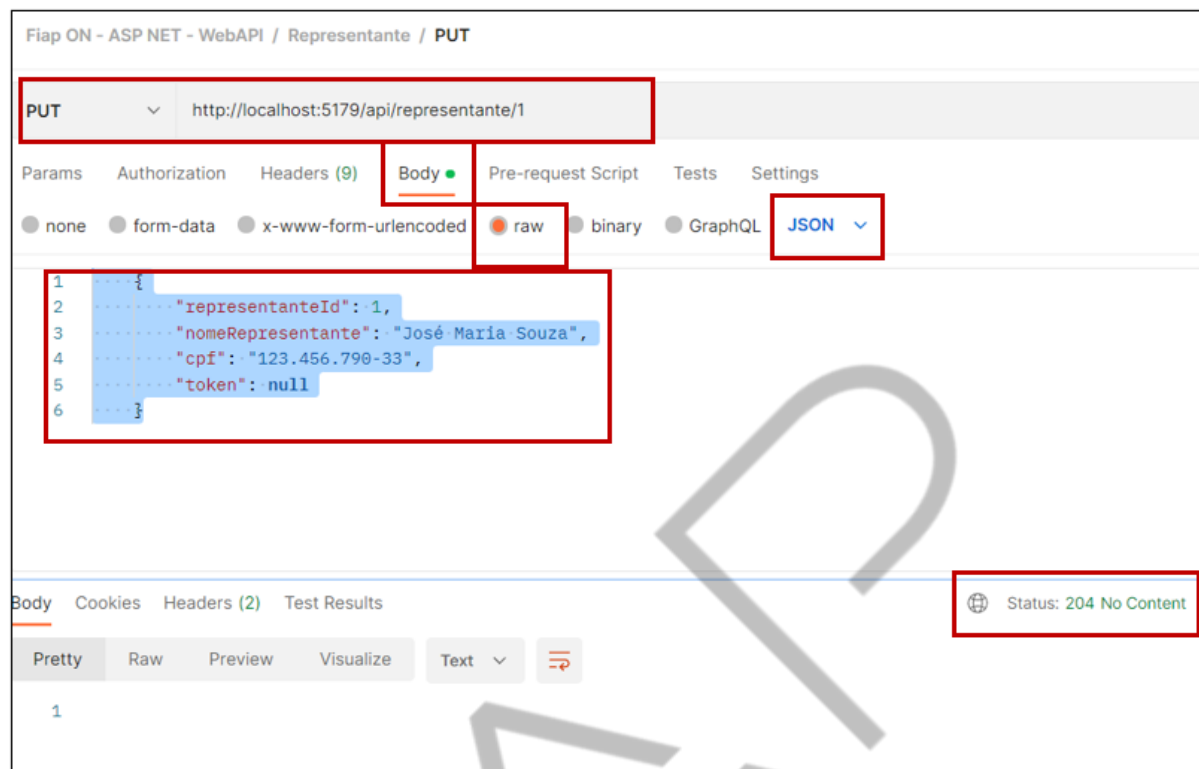


Figura 19 – Requisição PUT no Postman
Fonte: Elaborado pelo autor (2022)

20Seguem os links para download da solução Fiap.Api.AspNet:

Git: <https://github.com/FIAPON/Fiap.Api.AspNet/tree/controller>

Zip <https://github.com/FIAPON/Fiap.Api.AspNet/archive/refs/heads/controller.zip>

A coleção Postman para execução dos testes no Controller de Representante pode ser baixada no link:

<https://github.com/FIAPON/Fiap.Api.AspNet/blob/postman/Fiap.Api.AspNet/postman-collection.json>

21Seguem os links para download da solução Fiap.Api.AspNet com a funcionalidade de cliente implementada:

Git: <https://github.com/FIAPON/Fiap.Api.AspNet/tree/controller-cliente/>

WebServices no universo Microsoft

Zip: <https://github.com/FIAPON/Fiap.Api.AspNet/archive/refs/heads/controller-cliente.zip>

FIAPON

4 CONSUMIR UMA API REST

4.1 Padrão de retorno JSON

Por padrão, uma **ASP.NET Web API** retorna suas respostas com a formatação dos dados **JSON**, assim como muitas das API disponíveis para uso no mercado. O padrão JSON comporta-se melhor em alguns cenários, por exemplo, nas requisições **POST** e **PUT**, nas quais 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 de maneira perfeita.

4.2 Criar a aplicação Cliente

Para este exemplo, vamos precisar de uma aplicação do tipo Console C#. O nome da nossa aplicação será **Fiap.Api.WebClient** e terá como objetivo a inserção dos novos tipos de produtos e a consulta dos tipos cadastrados. Assim, realizaremos a execução das APIs no método POST e GET, e, para as duas execuções, trabalharemos apenas com dados no formato JSON.

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

4.3 Cliente de requisição GET com JSON

A forma mais simples de executar requisições das APIs com C# é usando as classes **System.Net.Http.HttpClient** e **System.Net.Http.HttpResponseMessage**. A classe **HttpClient** é a responsável por 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 uso e manipulação.

O exemplo, a seguir, executará o método GET da nossa API de Cliente, e caso haja sucesso na execução, será exibido o conteúdo JSON com todos os registros cadastrados.

Veja o Código-Fonte “Cliente para requisição GET” da classe Program.cs:

```
namespace Fiap.Api.WebClient;
class Program
{
    private static readonly string urlEndPoint = "http://localhost:5179/api/cliente/";

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

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

        // Execute o método Get passando a url da API e salvando o resultado.
        // em um objeto do tipo HttpResponseMessage
        HttpResponseMessage resposta = client.GetAsync(urlEndPoint).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 16 – Cliente para requisição GET
Fonte: Elaborado pelo autor (2022)

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



Figura 20 – Requisição GET no *Client*
Fonte: Elaborado pelo autor (2022)

4.3.1 Requisição POST com JSON

Seguindo a mesma linha da requisição GET, usaremos 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. Por isso, faremos o uso da classe **System.Web.Net.StringContent** para transformar um texto em um conteúdo JSON, que será entendido pelo *Controller* WebAPI.

O Código-Fonte “Cliente para requisição POST” da classe **Program.cs** apresenta o exemplo de conversão de um texto em conteúdo JSON e a execução do método POST da **WebAPI** Representante:

```
using System.Text;

namespace Fiap.Api.WebClient;
class Program
{
    private static readonly string urlEndPoint = "http://localhost:5179/api/cliente/";

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

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

        // Conteúdo do tipo de produto em JSON.
        string json = "{ \"nome\": \"Cauani Sanches\", \"dataNascimento\": \"1990-03-01T00:00:00\", \"observacao\": \"Texto de Observação\", \"representanteId\": 1}";
    }
}
```

```
// 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.
HttpResponseMessage resposta = client.PostAsync(urlEndPoint, conteudo).Result;

// Verifica que o POST foi executado com sucesso
if (resposta.IsSuccessStatusCode)
{
    Console.WriteLine("Cliente criado com sucesso");
    Console.WriteLine("Link para consulta: " + resposta.Headers.Location);
} else
{
    Console.WriteLine("Erro: " + resposta.StatusCode + " - " + resposta.Content);
}
}
```

Código-fonte 17 – Cliente para requisição POST
Fonte: Elaborado pelo autor (2022)

Segue também a imagem com o resultado da execução na Figura “Requisições POST no *Client*”:

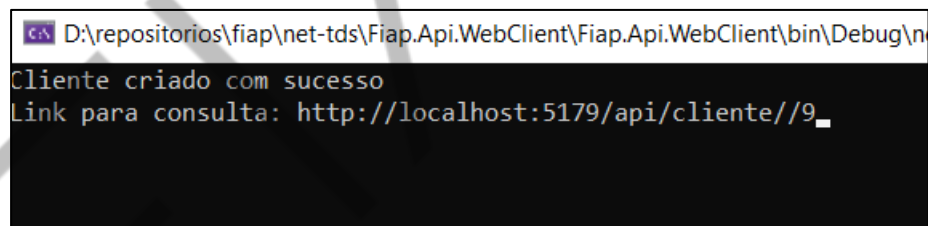


Figura 21 – Requisições POST no *Client*
Fonte: Elaborado pelo autor (2022)

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 **Cliente** com uma requisição **GET**.

Seguem os links para download da solução implementada até esse momento:

Git: <https://github.com/FIAPON/Fiap.Api.WebClient/tree/get-post>

Zip: <https://github.com/FIAPON/Fiap.Api.WebClient/archive/refs/heads/get-post.zip>

4.3.2 Transformação de dados (Parse)

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

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

Antes de apresentar as transformações, criaremos, em nosso projeto **Fiap.Api.WebClient**, o *namespace* Models e adicionaremos as duas classes, Representante e Cliente. Essas classes devem ter os atributos do mesmo tipo e nomes usados nos projetos **Fiap.Web.AspNet** (MVC) e **Fiap.Api.AspNet**.

Além disso, a classe Representante deve conter a linha *using System.Linq*.

A Figura “Classes de modelo utilizadas para Parse”, a seguir, apresenta o conteúdo das duas classes de modelo:

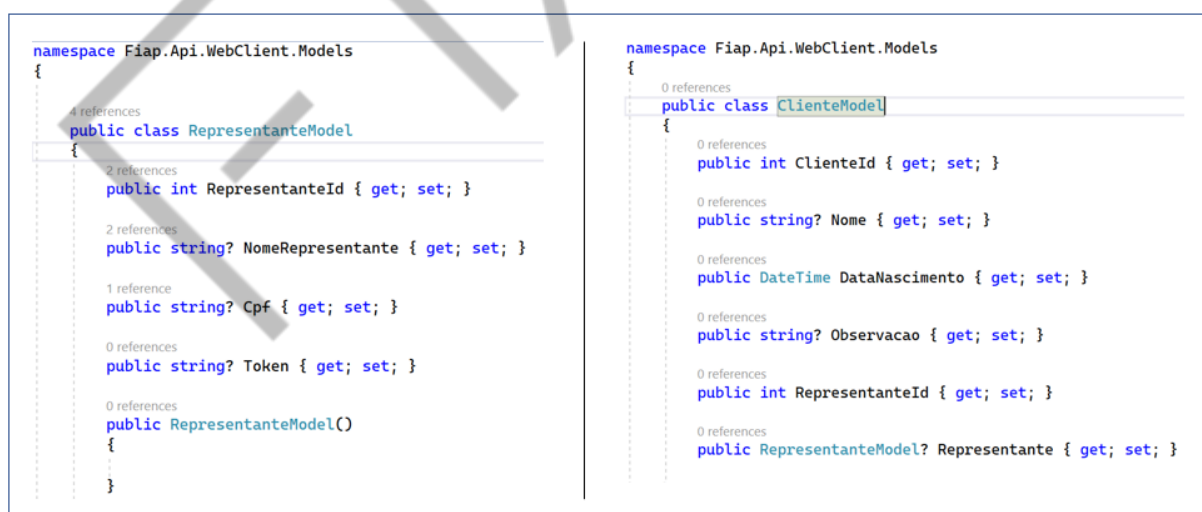


Figura 22 – Classes de modelo utilizadas para Parse
Fonte: Elaborado pelo autor (2022)

4.3.3 Desserialização

A palavra **Desserialização** é um pouco estranha, certo? Mas sua função é bem simples. Desserialização significa transformar conteúdo de 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 Cliente. Se a requisição for efetuada com sucesso, a API retorna uma lista dos objetos Cliente com suas propriedades e seus 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 Cliente e, por fim, interagir sobre a lista e imprimir os resultados. Veja o Código-Fonte “Desserialização JSON em C#”:

```
using System.Text;
using Fiap.Api.WebClient.Models;
using Newtonsoft.Json;

namespace Fiap.Api.WebClient;
class Program
{
    private static readonly string urlEndPoint = "http://localhost:5179/api/cliente/";

    static void Main(string[] args)
    {
        get();
    }
}
```



```
        Console.Read();
    }

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

        // Execute o método Get passando a url da API e salvando o resultado.
        // em um objeto do tipo HttpResponseMessage
        HttpResponseMessage resposta = client.GetAsync(urlEndPoint).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 Cliente
            List<ClienteModel> lista =
                JsonConvert.DeserializeObject<List<ClienteModel>>(conteudo);

            // Imprime o conteúdo na janela Console
            foreach (var item in lista)
            {
                Console.WriteLine("Nome:" + item.Nome);
                Console.WriteLine("Nascimento:" + item.DataNascimento);
                Console.WriteLine("=====");
                Console.WriteLine("");
            }
        }
        else
        {
            Console.WriteLine("Erro:" + resposta.StatusCode + " - " + resposta.Content);
        }
    }
}
```

Código-fonte 18 – Desserialização JSON em C#
Fonte: Elaborado pelo autor (2022)

4.4 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, usaremos os mesmos componentes da biblioteca **Newtonsoft.Json**, alterando apenas a chamada para o método de serialização.

Veja o exemplo no Código-Fonte “Serialização JSON em C#” abaixo:

```
using System.Text;
using Fiap.Api.WebClient.Models;
using Newtonsoft.Json;

namespace Fiap.Api.WebClient;
class Program
{
    private static readonly string urlEndPoint = "http://localhost:5179/api/cliente/";

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

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

        ClienteModel clienteModel = new ClienteModel();
        clienteModel.Nome = "Fátima Silva";
        clienteModel.Representanteld = 2;
        clienteModel.DataNascimento = new DateTime();
        clienteModel.Observacao = "Obs Obs";

        // Conteúdo do Cliente em JSON
        var json = JsonConvert.SerializeObject(clienteModel);

        // 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.
        HttpResponseMessage resposta = client.PostAsync(urlEndPoint, conteudo).Result;

        // Verifica que o POST foi executado com sucesso
        if (resposta.IsSuccessStatusCode)
        {
        }
```

```
        Console.WriteLine("Cliente criado com sucesso");  
        Console.WriteLine("Link para consulta: " + resposta.Headers.Location);  
    } else  
    {  
        Console.WriteLine("Erro:" + resposta.StatusCode + " - " + resposta.Content);  
    }  
}  
  
}
```

Código-fonte 19 – Serialização JSON em C#
Fonte: Elaborado pelo autor (2022)

Observe 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 Cliente que será convertida e postada no corpo da requisição POST.

23 Seguem os links para download da solução completa:

Git: <https://github.com/FIAPON/Fiap.Api.WebClient/tree/serialize-deserialize>

Zip: <https://github.com/FIAPON/Fiap.Api.WebClient/archive/refs/heads/serialize-deserialize.zip>

CONCLUSÃO

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 (por exemplo: *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# e, para isso, um projeto do tipo Console exemplificou as chamadas de API, as facilidades de trabalho com o formato JSON e as formas de transformação dos dados (Parse).

REFERÊNCIAS

ARAÚJO, E. C. Orientação a Objetos em C# – Conceitos e implementações em .NET. São Paulo: Casa do Código, 2016.

ARAÚJO, E. C. ASP.NET MVC5 – Crie aplicações web na plataforma Microsoft. São Paulo: Casa do Código, 2016.

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: 13 jan. 2021.

MICROSOFT. **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: 13 jan. 2021.

SIÉCOLA, P. Web Services REST com ASP.NET Web API e Windows Azure. São Paulo: Casa do Código, 2016.

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: 13 jan. 2021.