

## **Desenvolver um aplicativo Web no ASP.NET Core que consome uma API.**

### **1 - Desenvolver um aplicativo Web no ASP.NET Core que consome uma API.**

- 1.1 Introdução.
- 1.2 Objetivos de aprendizagem.
- 1.3 Explorar APIs do ASP.NET Core.
- 1.4 API baseada em controlador.
- 1.5 API mínima.
- 1.6 Documentar uma API usando o Swashbuckle.
- 1.7 Adicionar e configurar o middleware do Swagger.
- 1.8 Personalizar e estender a documentação do Swagger.
- 1.9 Teste.
- 1.10       Resumo

### **2 - Interaja com uma API mínima do ASP.NET Core.**

- 2.1 Objetivos de aprendizagem.
- 2.2 Pré-requisito.
- 2.3 Introdução.
- 2.4 Objetivo e aprendizagem.
- 2.5 Explorar APIs do ASP.NET Core.
- 2.6 API baseada em controlador.
- 2.7 API mínima.
- 2.8 Documentar uma API usando o Swashbuckle.
- 2.9 Adicionar e configurar o middleware do Swagger.
- 2.10       Personalizar e estender a documentação do Swagger.
- 2.11       Teste.
- 2.12       Resumo.

### **3 - Implementar operações HTTP do Razor Pages no ASP.NET.**

- 3.1 Objetivos de aprendizagem.
- 3.2 Pré-requisito.
- 3.3 Introdução.
- 3.4 Objetivo e aprendizagem.
- 3.5 Explorar clientes HTTP no .NET Core.
- 3.6 Implementar com a classe HttpClient.
- 3.7 Implementar com IHttpClientFactory.
- 3.8 Executar operações HTTP no Razor Pages.
- 3.9 Registrar IHttpClientFactory no seu aplicativo.
- 3.10       Identificar os requisitos de operação na API.
- 3.11       Métodos do manipulador no Razor Pages
- 3.12       Executar uma operação GET.
- 3.13       Executar uma operação POST.
- 3.14       Executar outras operações REST.
- 3.15       Teste.
- 3.16       Resumo

### **4 - Renderizar respostas de API no Razor Pages do ASP.NET Core.**

- 4.1 Objetivos de aprendizagem.

- 4.2 Pré-requisito.
- 4.3 Introdução.
- 4.4 Objetivo e aprendizagem.
- 4.5 Explorar a estrutura do projeto do Razor Pages
- 4.6 Estrutura de projeto do Razor Pages.
- 4.7 Descobrir a sintaxe do Razor Pages.
- 4.8 Sintaxe Razor.
- 4.9 Adicionar código a uma página usando o caractere @.
- 4.10 Teste.
- 4.11 Resumo.

## **1 - Desenvolver um aplicativo Web no ASP.NET Core que consome uma API**

### **1.1- Introdução**

Uma API, ou Interface de Programação de Aplicativo, é um conjunto de regras e protocolos que permite que diferentes aplicativos de software se comuniquem entre si.

As APIs fornecem uma maneira para os desenvolvedores acessarem a funcionalidade de outra parte do software, como um serviço Web, e usarem essa funcionalidade em seus próprios aplicativos.

As APIs são usadas por vários motivos, incluindo:

- Simplificar o desenvolvimento: usando APIs, os desenvolvedores podem acessar a funcionalidade de outro software sem precisar entender os detalhes de como esse software funciona. Isso pode economizar tempo e esforço ao criar novos aplicativos.
- Habilitar a integração: as APIs permitem que diferentes aplicativos de software trabalhem juntos, mesmo que não tenham sido originalmente projetados para fazer isso. Isso pode ajudar as empresas a integrar seus sistemas e dados, melhorando a eficiência e a produtividade.

### **1.2-Objetivos de aprendizagem**

Depois de concluir este módulo, você poderá:

- Descrever os dois tipos de modelo de APIs no ASP.NET Core.
- Crie a documentação do Swagger para uma API usando o Swashbuckle.
- Interaja com uma API usando a interface do Swagger.

### **1.3-Explorar APIs do ASP.NET Core**

O ASP.NET Core dá suporte a duas abordagens para criar APIs: uma abordagem baseada em controlador e APIs mínimas. Uma API baseada em controlador é uma abordagem tradicional para criar APIs em que cada ponto de extremidade é mapeado para uma classe de controlador específica.

O controlador manipula a solicitação, executa qualquer lógica de negócios necessária e retorna uma resposta.

### **1.4-API baseada em controlador**

Uma API Web baseada em controlador consiste em uma ou mais classes de controlador que derivam de ControllerBase.

Veja a seguir um exemplo de um controlador:

```
C# Copiar

[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Os controladores de API Web normalmente devem derivar de ControllerBase, não de Controller.

Controller é derivado de ControllerBase e agrega suporte para exibições; portanto, serve para manipulação de páginas da Web, não para solicitações de API Web.

A classe ControllerBase fornece muitas propriedades e métodos úteis para lidar com solicitações HTTP.

Por exemplo, no exemplo de código CreatedAtAction a seguir retorna um código de status 201:

```
C# Copiar

[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Pet> Create(Pet pet)
{
    pet.Id = _petsInMemoryStore.Any() ?
        _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
    _petsInMemoryStore.Add(pet);

    return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);
}
```

## 1.5-API mínima

APIs mínimas são uma abordagem simplificada para criar APIs de HTTP rápidas com o ASP.NET Core.

Você pode criar pontos de extremidade REST totalmente funcionais com o mínimo de código e configuração. Declare fluentemente rotas e ações de API para ignorar o scaffolding tradicional.

Por exemplo, o código a seguir cria uma API na raiz do aplicativo Web que retorna o texto "Olá, Mundo!".

```
C# Copiar

var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.Run();
```

A maioria das APIs aceita parâmetros como parte da rota.

```
C# Copiar

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/users/{userId}/books/{bookId}",
    (int userId, int bookId) => $"The user id is {userId} and book id is {bookId}");

app.Run();
```

APIs mínimas dão suporte à configuração e à personalização necessárias para dimensionar para várias APIs, lidar com rotas complexas, aplicar regras de autorização e controlar o conteúdo das respostas à API.

### 1.6-Documentar uma API usando o Swashbuckle.

Swashbuckle é um pacote NuGet que fornece uma maneira de gerar automaticamente a documentação do Swagger para projetos de ASP.NET Web API.

O Swagger é uma ferramenta que ajuda os desenvolvedores a projetar, criar, documentar e consumir APIs RESTful. Com o Swashbuckle, você pode facilmente adicionar a documentação do Swagger ao seu projeto de API Web, anotando seu código com atributos que descrevem os pontos de extremidade, os parâmetros e as respostas da API. Em seguida, o Swashbuckle utiliza essas informações para gerar um arquivo JSON do Swagger, que pode ser utilizado para gerar uma documentação interativa da API, SDKs de clientes e muito mais.

O Swashbuckle tem três componentes principais:

- Swashbuckle.AspNetCore.Swagger: um modelo de objeto Swagger e middleware para expor objetos SwaggerDocument como pontos de extremidade JSON.
- Swashbuckle.AspNetCore.SwaggerGen: um gerador Swagger que cria objetos SwaggerDocument diretamente das suas rotas, controladores e modelos. Normalmente, ela é combinada com o middleware de ponto de extremidade de Swagger para expor automaticamente o JSON do Swagger.
- Swashbuckle.AspNetCore.SwaggerUI: uma versão incorporada da ferramenta de interface do usuário do Swagger. Ele interpreta o JSON do Swagger a fim de criar uma experiência rica e personalizável para descrever a funcionalidade da API Web. Ele inclui os agentes de teste internos para os métodos públicos.

O comando dotnet run a seguir instala o pacote NuGet do Swashbuckle:


```
CLI do .NET Copiar

dotnet add <name>.csproj package Swashbuckle.AspNetCore -v 6.5.0
```

### 1.7-Adicionar e configurar o middleware do Swagger.

Adicionar o gerador Swagger à coleção de serviços em Program.cs.

C#

 Copiar


```
builder.Services.AddEndpointsApiExplorer();  
builder.Services.AddSwaggerGen();
```

### 🕒 Observação

A chamada à `AddEndpointsApiExplorer` mostrada no exemplo anterior é obrigatória apenas para APIs mínimas.

Habilite o middleware para servir o documento JSON gerado e a interface do usuário do Swagger, também em Program.cs:

C#

 Copiar

```
app.UseSwagger();  
app.UseSwaggerUI();
```

O ponto de extremidade padrão para a interface do usuário do Swagger é `http:<hostname>:<port>/swagger`.


## 1.8-Personalizar e estender a documentação do Swagger.

O Swagger fornece opções para documentar o modelo de objeto e personalizar a interface do usuário.

A ação de configuração passada do método `AddSwaggerGen` pode incluir informações adicionais por meio da classe `OpenApiInfo`.

O código de exemplo a seguir mostra como adicionar as informações a serem exibidas na documentação da API.

C#

 Copiar

```
app.MapPost("/fruitlist", async (Fruit fruit, FruitDb db) =>  
{  
    db.Fruits.Add(fruit);  
    await db.SaveChangesAsync();  
  
    return Results.Created($" /fruitlist/{fruit.Id}", fruit);  
})  
    .WithTags("Add fruit to list");
```

A interface do usuário do Swagger exibe a versão e as informações adicionadas:

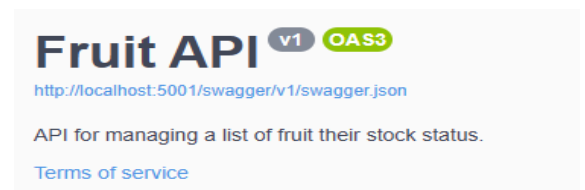
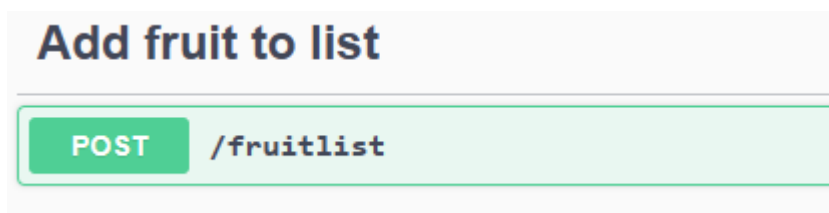
```
C# Copiar

// Add using statement for the OpenApiInfo class
using Microsoft.OpenApi.Models;

builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "Fruit API",
        Description = "API for managing a list of fruit their stock status.",
        TermsOfService = new Uri("https://example.com/terms")
    });
});
```

Você pode adicionar títulos descritivos às diferentes funções na sua API utilizando a opção `WithTags`. O código de exemplo a seguir mostra a adição de "Adicionar fruta à lista" como um título de um mapeamento POST:

A interface do usuário do Swagger exibe a marca "Adicionar fruta à lista" como um título acima da ação POST.



## 1.9-Teste.

1.Qual é a diferença entre uma API baseada em controlador e uma API mínima no ASP.NET Core?

- ☐ As APIs mínimas exigem mais código e configuração do que as APIs baseadas em controladores.
- ☐ As APIs mínimas são utilizadas para lidar com páginas da Web, e não com solicitações de API da Web.
- ☐ As APIs baseadas em controlador mapeiam cada ponto de extremidade para uma classe de controlador específica, enquanto as APIs mínimas declaram rotas e ações de API para criar pontos de extremidade REST com funcionalidade total com o mínimo de código e configuração.

2.O que é o Swashbuckle?

- ☐ Uma ferramenta que ajuda os desenvolvedores a projetar, criar, documentar e consumir APIs SOAP.
- ☐ Um pacote NuGet que fornece uma maneira de gerar automaticamente a documentação XML de projetos de API Web ASP.NET.
- ☐ Um pacote NuGet que fornece uma maneira de gerar automaticamente a documentação do Swagger para projetos ASP.NET Web API.

## 1.10-Resumo.

Neste módulo, você aprendeu a:

- Descrever os dois tipos de modelo de APIs no ASP.NET Core.
- Crie a documentação do Swagger para uma API usando o Swashbuckle.
- Interaja com uma API usando a interface do Swagger.

## 2 - Interaja com uma API mínima do ASP.NET Core.

Saiba como coletar informações da documentação da API e executar operações HTTP em um aplicativo Web do Razor Pages no ASP.NET Core.

### 2.1 - Objetivos de aprendizagem.

Depois de concluir este módulo, você poderá:

- Descrever os dois tipos de modelo de APIs na CLI do .NET Core
- Criar uma documentação Swagger para uma API utilizando o Swashbuckle
- Interaja com uma API utilizando a interface Swagger

### 2.2 – Pré-requisitos.

- Conhecimento de serviços RESTful e verbos de ação HTTP.
- Experiência com JSON em um nível iniciante.

### 2.3 – Introdução.

Uma API, ou Interface de Programação de Aplicativo, é um conjunto de regras e protocolos que permite que diferentes aplicativos de software se comuniquem entre si.

As APIs fornecem uma maneira para os desenvolvedores acessarem a funcionalidade de outra parte do software, como um serviço Web, e usarem essa funcionalidade em seus próprios aplicativos.

As APIs são usadas por vários motivos, incluindo:

- Simplificar o desenvolvimento: usando APIs, os desenvolvedores podem acessar a funcionalidade de outro software sem precisar entender os detalhes de como esse software funciona. Isso pode economizar tempo e esforço ao criar novos aplicativos.
- Habilitar a integração: as APIs permitem que diferentes aplicativos de software trabalhem juntos, mesmo que não tenham sido originalmente projetados para fazer isso. Isso pode ajudar as empresas a integrar seus sistemas e dados, melhorando a eficiência e a produtividade.

### 2.4 - Objetivos de aprendizagem.

Depois de concluir este módulo, você poderá:

- Descrever os dois tipos de modelo de APIs no ASP.NET Core.
- Crie a documentação do Swagger para uma API usando o Swashbuckle.
- Interaja com uma API usando a interface do Swagger.

## 2.5 - Explorar APIs do ASP.NET Core.

O ASP.NET Core dá suporte a duas abordagens para criar APIs: uma abordagem baseada em controlador e APIs mínimas.

Uma API baseada em controlador é uma abordagem tradicional para criar APIs em que cada ponto de extremidade é mapeado para uma classe de controlador específica.

O controlador manipula a solicitação, executa qualquer lógica de negócios necessária e retorna uma resposta.

## 2.6 - API baseada em controlador.

Uma API Web baseada em controlador consiste em uma ou mais classes de controlador que derivam de ControllerBase. Veja a seguir um exemplo de um controlador:

```
C# Copiar

[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Os controladores de API Web normalmente devem derivar de ControllerBase, não de Controller.

Controller é derivado de ControllerBase e agrega suporte para exibições; portanto, serve para manipulação de páginas da Web, não para solicitações de API Web.

A classe ControllerBase fornece muitas propriedades e métodos úteis para lidar com solicitações HTTP. Por exemplo, no exemplo de código **CreatedAtAction** a seguir retorna um código de status 201:

```
C# Copiar

[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Pet> Create(Pet pet)
{
    pet.Id = _petsInMemoryStore.Any() ?
        _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
    _petsInMemoryStore.Add(pet);

    return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);
}
```

## 2.7 - API mínima.

APIs mínimas são uma abordagem simplificada para criar APIs de HTTP rápidas com o ASP.NET Core.

Você pode criar pontos de extremidade REST totalmente funcionais com o mínimo de código e configuração. Declare fluentemente rotas e ações de API para ignorar o scaffolding tradicional.

Por exemplo, o código a seguir cria uma API na raiz do aplicativo Web que retorna o texto "Olá, Mundo!".



```
C# Copiar

var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.Run();
```

A maioria das APIs aceita parâmetros como parte da rota.

```
C# Copiar

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/users/{userId}/books/{bookId}",
    (int userId, int bookId) => $"The user id is {userId} and book id is {bookId}");

app.Run();
```

APIs mínimas dão suporte à configuração e à personalização necessárias para dimensionar para várias APIs, lidar com rotas complexas, aplicar regras de autorização e controlar o conteúdo das respostas à API.

## 2.8 - Documentar uma API usando o Swashbuckle.

Swashbuckle é um pacote NuGet que fornece uma maneira de gerar automaticamente a documentação do Swagger para projetos de ASP.NET Web API.

O Swagger é uma ferramenta que ajuda os desenvolvedores a projetar, criar, documentar e consumir APIs RESTful.


Com o Swashbuckle, você pode facilmente adicionar a documentação do Swagger ao seu projeto de API Web, anotando seu código com atributos que descrevem os pontos de extremidade, os parâmetros e as respostas da API. Em seguida, o Swashbuckle utiliza essas informações para gerar um arquivo JSON do Swagger, que pode ser utilizado para gerar uma documentação interativa da API, SDKs de clientes e muito mais.

O Swashbuckle tem três componentes principais:

- Swashbuckle.AspNetCore.Swagger: um modelo de objeto Swagger e middleware para expor objetos SwaggerDocument como pontos de extremidade JSON.
- Swashbuckle.AspNetCore.SwaggerGen: um gerador Swagger que cria objetos SwaggerDocument diretamente das suas rotas, controladores e modelos. Normalmente, ela é combinada com o middleware de ponto de extremidade de Swagger para expor automaticamente o JSON do Swagger.
- Swashbuckle.AspNetCore.SwaggerUI: uma versão incorporada da ferramenta de interface do usuário do Swagger. Ele interpreta o JSON do Swagger a fim de criar uma experiência rica e personalizável para descrever a funcionalidade da API Web. Ele inclui os agentes de teste internos para os métodos públicos.

O comando dotnet run a seguir instala o pacote NuGet do Swashbuckle:

CLI do .NET


 Copiar

```
dotnet add <name>.csproj package Swashbuckle.AspNetCore -v 6.5.0
```

## 2.9 - Adicionar e configurar o middleware do Swagger

Adicionar o gerador Swagger à coleção de serviços em Program.cs.

C#

 Copiar


```
builder.Services.AddEndpointsApiExplorer();  
builder.Services.AddSwaggerGen();
```

### ⓘ Observação

A chamada à `AddEndpointsApiExplorer` mostrada no exemplo anterior é obrigatória apenas para APIs mínimas.

Habilite o middleware para servir o documento JSON gerado e a interface do usuário do Swagger, também em Program.cs:

C#

 Copiar

```
app.UseSwagger();  
app.UseSwaggerUI();
```

O ponto de extremidade padrão para a interface do usuário do Swagger é `http:<hostname>:<port>/swagger`.

## 2.10 - Personalizar e estender a documentação do Swagger

O Swagger fornece opções para documentar o modelo de objeto e personalizar a interface do usuário.

A ação de configuração passada do método **AddSwaggerGen** pode incluir informações adicionais por meio da classe **OpenApiInfo**.

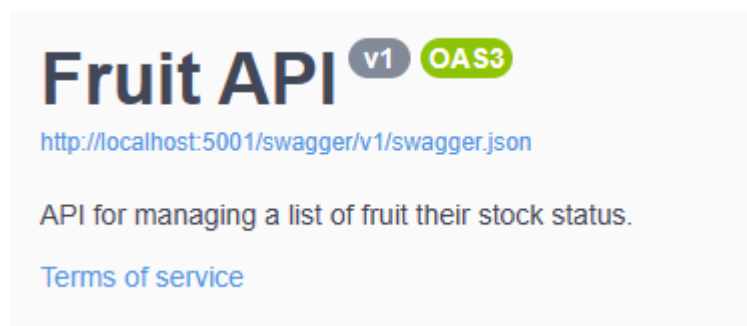
O código de exemplo a seguir mostra como adicionar as informações a serem exibidas na documentação da API.

```
C# Copiar

// Add using statement for the OpenApiInfo class
using Microsoft.OpenApi.Models;

builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "Fruit API",
        Description = "API for managing a list of fruit their stock status.",
        TermsOfService = new Uri("https://example.com/terms")
    });
});
```

A interface do usuário do Swagger exibe a versão e as informações adicionadas:



Você pode adicionar títulos descritivos às diferentes funções na sua API utilizando a opção **.WithTags**.

O código de exemplo a seguir mostra a adição de "Adicionar fruta à lista" como um título de um mapeamento POST:

```
C# Copiar

app.MapPost("/fruitlist", async (Fruit fruit, FruitDb db) =>
{
    db.Fruits.Add(fruit);
    await db.SaveChangesAsync();

    return Results.Created($"/fruitlist/{fruit.Id}", fruit);
})
.WithTags("Add fruit to list");
```

A interface do usuário do Swagger exibe a marca "Adicionar fruta à lista" como um título acima da ação POST.

## Add fruit to list

POST

/fruitlist

### 2.11 - Teste

1. Qual é a diferença entre uma API baseada em controlador e uma API mínima no ASP.NET Core?

- ☐ As APIs mínimas exigem mais código e configuração do que as APIs baseadas em controladores.
- ☐ As APIs mínimas são utilizadas para lidar com páginas da Web, e não com solicitações de API da Web.
- ☐ As APIs baseadas em controlador mapeiam cada ponto de extremidade para uma classe de controlador específica, enquanto as APIs mínimas declaram rotas e ações de API para criar pontos de extremidade REST com funcionalidade total com o mínimo de código e configuração.

2. O que é o Swashbuckle?

- ☐ Uma ferramenta que ajuda os desenvolvedores a projetar, criar, documentar e consumir APIs SOAP.
- ☐ Um pacote NuGet que fornece uma maneira de gerar automaticamente a documentação XML de projetos de API Web ASP.NET.
- ☐ Um pacote NuGet que fornece uma maneira de gerar automaticamente a documentação do Swagger para projetos ASP.NET Web API.

### 2.12 - Resumo

Neste módulo, você aprendeu a:

- Descrever os dois tipos de modelo de APIs no ASP.NET Core.
- Criar a documentação do Swagger para uma API usando o Swashbuckle.
- Interagir com uma API usando a interface do Swagger.

## 3 - Implementar operações HTTP do Razor Pages no ASP.NET

Saiba como implementar clientes HTTP com base em HttpClient e IHttpClientFactory. E como fazer para implementar o código para executar operações HTTP no Razor Pages do ASP.NET Core.

### 3.1 - Objetivos de aprendizagem

Depois de concluir este módulo, você poderá:

- Implementar clientes HTTP no .NET Core
- Usar clientes HTTP para executar operações seguras e inseguras
- Adicionar o código para dar suporte a operações HTTP em um aplicativo Razor Pages do ASP.NET Core

### 3.2 - Pré-requisitos

- Experiência na escrita em C# em um nível intermediário

- Conhecimento de serviços RESTful e verbos de ação HTTP

Este módulo faz parte destes roteiros de aprendizagem

- Desenvolver um aplicativo Web no ASP.NET Core que consome uma API

### 3.3 - Introdução

As solicitações HTTP (Hypertext Transfer Protocol) são uma maneira de solicitar que um servidor Web forneça ou atualize um recurso.

HTTP define um conjunto de métodos para enviar solicitações a um servidor Web.

O servidor responde à solicitação enviando de volta o recurso solicitado ou uma mensagem de erro se o recurso não estiver disponível ou se a solicitação for inválida.

### 3.4 - Objetivos de aprendizagem

Depois de concluir este módulo, você poderá:

- Implementar clientes HTTP no .NET Core.
- Use clientes HTTP para executar operações seguras e não seguras.
- Adicione código para dar suporte a operações HTTP em um aplicativo ASP.NET Core Razor Pages.

### 3.5 - Explorar clientes HTTP no .NET Core

O Protocolo de Transferência de Hipertexto (ou protocolo HTTP) é usado para solicitar recursos de um servidor web.

Muitos tipos de recursos estão disponíveis na Web, e o HTTP define um conjunto de métodos de solicitação para acessar esses recursos.

No .NET Core, essas solicitações são feitas por meio de uma instância `HttpClient`.

Uma instância `HttpClient` é uma coleção de configurações aplicadas a todas as solicitações executadas por essa instância e cada instância usa seu próprio pool de conexões, o que isola suas solicitações de outras.

Há duas opções para implementar o `HttpClient` em seu aplicativo e é recomendável escolher a implementação com base nas necessidades de gerenciamento de tempo de vida dos clientes:

- Para clientes de longa duração, crie uma instância **singleton** ou **static** usando a classe `HttpClient` e defina **PooledConnectionLifetime**.
- Para clientes de curta duração, use clientes criados pelo **IHttpClientFactory**.

### 3.6 - Implementar com a classe HttpClient

A classe `System.Net.Http.HttpClient` envia solicitações HTTP e recebe respostas HTTP de um recurso identificado por um URI.

Uma instância `HttpClient` é uma coleção de configurações aplicadas a todas as solicitações executadas por essa instância e cada instância usa seu próprio pool de conexões, o que isola suas solicitações de outras.

A partir do .NET Core 2.1, a classe `SocketsHttpHandler` fornece a implementação, tornando o comportamento consistente em todas as plataformas.

HttpClient só resolve entradas DNS quando uma conexão é criada. Ele não controla as durações do TTL (tempo de vida útil) especificadas pelo servidor DNS. Se as entradas de DNS forem alteradas regularmente, o cliente não estará ciente dessas atualizações.

Para resolver esse problema, você pode limitar o tempo de vida da conexão definindo a propriedade **PooledConnectionLifetime**, de modo que a pesquisa DNS seja repetida quando a conexão for substituída.

No exemplo a seguir, o **HttpClient** está configurado para reutilizar conexões por **15 minutos**. Depois que o **TimeSpan** especificado por **PooledConnectionLifetime** tiver decorrido, a conexão será fechada e uma nova será criada.

```
C# Copiar

var handler = new SocketsHttpHandler
{
    PooledConnectionLifetime = TimeSpan.FromMinutes(15) // Recreate every 15 minutes
};
var sharedClient = new HttpClient(handler);
```

### 3.7 - Implementar com IHttpHttpClientFactory

O **IHttpClientFactory** serve como uma abstração de fábrica que pode criar instâncias **HttpClient** com configurações personalizadas. **IHttpClientFactory** foi introduzido no .NET Core 2.1. Cargas de trabalho comuns do .NET baseadas em HTTP podem aproveitar o middleware de terceiros com facilidade.

Quando você chama qualquer um dos métodos de extensão **AddHttpClient**, você está adicionando **IHttpClientFactory** e serviços e relacionados ao **IServiceCollection**. O tipo **IHttpClientFactory** oferece os seguintes benefícios:

- Expõe a classe **HttpClient** como um tipo pronto para injeção de dependência.
- Fornece um local central para nomear e configurar instâncias lógicas de **HttpClient**.
- Codifica o conceito de middleware de saída por meio da delegação de manipuladores em **HttpClient**.
- Fornece métodos de extensão para o middleware baseado em Polly para aproveitar a delegação de manipuladores em **HttpClient**.
- Gerencia o cache e o tempo de vida das instâncias **HttpClientHandler** subjacentes. O gerenciamento automático evita problemas comuns do DNS (Sistema de Nomes de Domínio) que ocorrem ao gerenciar manualmente o tempos de vida **HttpClient**.
- Adiciona uma experiência de registro em log configurável para todas as solicitações enviadas por meio de clientes criados pelo alocador.

Você deve deixar que o **HttpClientFactory** e a estrutura gerenciem os tempos de vida e a instanciação de instâncias **HttpClient**. Isso ajuda a evitar problemas comuns, como problemas de DNS (Sistema de Nomes de Domínio) que podem ocorrer ao gerenciar manualmente tempos de vida do **HttpClient**.

Há várias maneiras de usar o **IHttpClientFactory** em um aplicativo:

- Uso básico
- Clientes nomeados
- Clientes com tipo

- Clientes gerados

A melhor abordagem depende dos requisitos do aplicativo.

### 3.8 - Executar operações HTTP no Razor Pages.

Nesta unidade, você aprenderá como usar o `IHttpClientFactory` para lidar com a criação e o descarte do cliente HTTP e para usar esse cliente para executar operações REST em um aplicativo Razor Pages do ASP.NET. Os códigos de exemplo utilizados ao longo desta unidade são baseados na interação com uma API que habilita o gerenciamento de uma lista de frutas armazenadas em um banco de dados.

O código a seguir representa o modelo de dados que é referenciado nos exemplos de código:

```
C# Copiar

public class FruitModel
{
    // An id assigned by the database
    public int id { get; set; }
    // The name of the fruit
    public string? name { get; set; }
    // A boolean to indicate if the fruit is in stock
    public bool instock { get; set; }
}
```

### 3.9 - Registrar `IHttpClientFactory` no seu aplicativo.

Para adicionar `IHttpClientFactory` ao seu aplicativo, registre o `AddHttpClient` no arquivo `Program.cs`. O exemplo de código a seguir utiliza o tipo de cliente nomeado e define o endereço base da API utilizado nas operações REST e é referenciado em todo o restante desta unidade.

```
C# Copiar

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

// Add IHttpClientFactory to the container and sets the name of the factory
// to "FruitAPI", and the also sets the base address used in calls
builder.Services.AddHttpClient("FruitAPI", httpClient =>
{
    httpClient.BaseAddress = new Uri("https://www.example.com");
});

var app = builder.Build();
```

### 3.10 - Identificar os requisitos de operação na API.

Antes de executar operações com uma API, você precisa identificar o que a API está esperando:

- Ponto de extremidade da API: identifique o ponto de extremidade da operação para que você possa ajustar adequadamente o URI armazenado no endereço base, se necessário.
- Requisitos de dados: identifique se a operação está retornando/esperando uma matriz ou apenas uma única parte dos dados.

Por exemplo, uma operação GET pode retornar uma matriz se o ponto de extremidade retornar todos os dados, ou qualquer número maior que um, armazenados no banco de dados. Uma

operação GET também pode retornar um único dado.

#### 🚨 Observação

Os códigos de exemplo no restante desta unidade assumem que cada operação HTTP é tratada em uma página separada na solução.

### 3.11 - Métodos do manipulador no Razor Pages

No Razor Pages, os métodos do manipulador são métodos que manipulam solicitações e eventos HTTP. Eles são definidos no arquivo code-behind de uma Razor Page e são utilizados para executar ações em resposta à entrada do usuário ou a outros eventos. Os métodos do manipulador seguem a convenção de nomenclatura da palavra "On" adicionada ao início do verbo de solicitação HTTP. A tabela a seguir lista alguns dos métodos manipuladores mais utilizados no Razor Pages:

Método do manipulador	Descrição
OnGet()	Esse método é chamado quando a página é solicitada utilizando o método HTTP GET.
OnPost()	Esse método é chamado quando a página é enviada utilizando o método HTTP POST.
OnGetAsync()	Esse método é chamado de forma assíncrona quando a página é solicitada utilizando o método HTTP GET.
OnPostAsync()	Esse método é chamado de forma assíncrona quando a página é enviada utilizando o método HTTP POST.
OnGetHandler()	Esse método é chamado quando um manipulador específico é solicitado utilizando o método HTTP GET.
OnPostHandler()	Esse método é chamado quando um manipulador específico é solicitado usando o método HTTP POST.

Existem também outros métodos de manipulação que podem ser utilizados para manipular solicitações e eventos HTTP, e você também pode definir seus próprios métodos de manipulador personalizados.

#### 🚨 Observação

Você pode ter apenas um método manipulador de página para uma operação HTTP específica. Por exemplo, uma exceção será gerada se você tiver `OnGet()` e `OnGetAsync()` no mesmo arquivo code-behind.

### 3.12 - Executar uma operação GET.

Uma operação GET não deve enviar um corpo e é utilizada (como o nome do método indica) para recuperar os dados de um recurso.

Para executar uma operação HTTP GET, dado um `HttpClient` e um URI, utilize o método `HttpClient.GetAsync`. Por exemplo, se você desejasse criar uma tabela na página inicial de um aplicativo Razor Page (`Index.cshtml`) para exibir os resultados de uma operação GET, você precisaria adicionar o seguinte ao code-behind (`Index.cshtml.cs`):

- Use a injeção de dependência para adicionar o `IHttpClientFactory` ao modelo da página.
- Usar o atributo `[BindProperty]` para associar os dados do formulário das páginas às propriedades do modelo de dados.
- Criar uma instância do `HttpClient`
- Execute a operação GET e desserialize os resultados no seu modelo de dados.

O exemplo de código a seguir mostra como executar uma operação GET. Não deixe de ler os comentários



no código.

```
namespace FruitWebApp.Pages
{
    public class IndexModel : PageModel
    {
        // IHttpClientFactory set using dependency injection
        private readonly IHttpClientFactory _httpClientFactory;

        public IndexModel(IHttpClientFactory httpClientFactory)
        {
            _httpClientFactory = httpClientFactory;
        }

        // Adds the data model and binds the form data to the model properties
        // Enumerable since an array is expected as a response
        [BindProperty]
        public IEnumerable<FruitModel> FruitModels { get; set; }

        // OnGet() is async since HTTP operations should be performed async
        public async Task OnGet()
        {
            // Create the HTTP client using the FruitAPI named factory
            var httpClient = _httpClientFactory.CreateClient("FruitAPI");

            // Execute the GET operation and store the response, the empty parameter
            // in GetAsync doesn't modify the base address set in the client factory
            using HttpResponseMessage response = await httpClient.GetAsync("");

            // If the operation is successful deserialize the results into the data model
            if (response.IsSuccessStatusCode)
            {
                using var contentStream = await response.Content.ReadAsStreamAsync();
                FruitModels = await JsonSerializer.DeserializeAsync<IEnumerable<FruitModel>>(contentStream);
            }
        }
    }
}
```

### 3.13 - Executar uma operação POST.

Uma operação POST deve enviar um corpo e é utilizada para adicionar dados a um recurso.

Para executar uma operação HTTP POST, dado um `HttpClient` e um `URI`, utilize o método `HttpClient.PostAsync`. Seguindo nosso modelo de projeto de uma página separada para cada página, se você desejar adicionar itens aos dados na sua página inicial, você precisará:

Por exemplo, se você desejasse criar uma tabela na página inicial de um aplicativo Razor Page (`Index.cshtml`) para exibir os resultados de uma operação GET, você precisaria adicionar o seguinte ao code-behind (`Index.cshtml.cs`):

- Use a injeção de dependência para adicionar o `IHttpClientFactory` ao modelo da página.
- Usar o atributo `[BindProperty]` para associar os dados do formulário das páginas às propriedades do modelo de dados.
- Serialize os dados que você deseja adicionar utilizando o método `JsonSerializer.Serialize`.
- Criar uma instância do `HttpClient`
- Executar a operação POST.

#### 📌 Observação

Seguindo nosso modelo de projeto de uma página separada para cada operação REST, a operação **POST** é executada em uma página *Add.cshtml* com o exemplo de código no arquivo code-behind *Add.cshtml.cs*.

O exemplo de código a seguir mostra como executar uma operação POST. Não deixe de ler os comentários no código.

```
namespace FruitWebApp.Pages
{
    public class AddModel : PageModel
    {
        // IHttpClientFactory set using dependency injection
        private readonly IHttpClientFactory _httpClientFactory;

        public AddModel(IHttpClientFactory httpClientFactory) => _httpClientFactory = httpClientFactory;

        // Add the data model and bind the form data to the page model properties
        [BindProperty]
        public FruitModel FruitModels { get; set; }

        // OnPost() is async since HTTP operations should be performed async
        public async Task<IActionResult> OnPost()
        {
            // Serialize the information to be added to the database
            var jsonContent = new StringContent(JsonSerializer.Serialize(FruitModels),
                Encoding.UTF8,
                "application/json");

            // Create the HTTP client using the FruitAPI named factory
            var httpClient = _httpClientFactory.CreateClient("FruitAPI");
```

```

// Execute the POST operation and store the response. The parameters in
// PostAsync direct the POST to use the base address and passes the
// serialized data to the API
using HttpResponseMessage response = await httpClient.PostAsync("", jsonContent);

        // If the POST was successful return to the home (Index) page
    if (response.IsSuccessStatusCode)
    {
        return RedirectToPage("Index");
    }
    return Page();
}
}

```

### 3.14 - Executar outras operações REST.

Outras operações, como PUT e DELETE, seguem o mesmo modelo mostrado anteriormente. A tabela a seguir define operações REST comuns juntamente com o método HttpClient associado:

Solicitar	Método	Definição
GET	HttpClient.GetAsync	Recupera um recurso.
POST	HttpClient.PostAsync	Cria um recurso.
PUT	HttpClient.PutAsync	Atualizações de um recurso existente ou a criação de um novo recurso, se ele não existir.
DELETE	HttpClient.DeleteAsync	Exclui um recurso.
PATCH	HttpClient.PatchAsync	Atualizações parciais de um recurso existente.

### 3.15 Teste

- Qual é a abordagem recomendada para implementar HttpClient em um aplicativo com clientes de longa duração?
  - ☐ Crie uma nova instância de HttpClient para cada solicitação.
  - ☐ Use clientes criados por IHttpConnectionFactory.
  - ☐ Crie uma instância static ou singleton ou usando a classe HttpClient e defina PooledConnectionLifetime.
- Qual é a finalidade de IHttpConnectionFactory em um aplicativo do Razor Pages no ASP.NET?
  - ☐ Para lidar com a criação e o descarte de operações REST.

☐ Para lidar com a criação e o descarte de solicitações HTTP.

☐ Para lidar com a criação e o descarte de clientes HTTP.

3. Qual é a convenção de nomenclatura para métodos de manipulador no Razor Pages?

☐ A palavra "Ação" adicionada ao início do verbo de solicitação HTTP.

☐ A palavra 'Handler' adicionada ao início do verbo de solicitação HTTP.

☐ A palavra "On" adicionada ao início do verbo de solicitação HTTP.

### 3.16 Resumo

Neste módulo, você aprendeu a:

- Implementar clientes HTTP no .NET Core.
- Use clientes HTTP para executar operações seguras e não seguras.
- Adicione código para dar suporte a operações HTTP em um aplicativo ASP.NET Core Razor Pages.

## 4 - Renderizar respostas de API no Razor Pages do ASP.NET Core

Saiba como renderizar respostas de API do Razor Pages no ASP.NET Core e executar operações HTTP usando manipuladores de página.

### 🚨 Observação

Este conteúdo foi parcialmente criado com a ajuda de IA. Um autor examinou e revisou o conteúdo conforme necessário. [Leia mais.](#)

### 4.1 - Objetivos de aprendizagem.

Depois de concluir este módulo, você poderá:

- Combinar HTML e C# para definir a lógica de renderização dinâmica da página
- Renderização de respostas da API no Razor Pages
- Criar páginas que executam operações HTTP

### 4.2 – Pré-requisitos.

- Experiência de escrita em C# em um nível intermediário
- Capacidade de escrever HTML no nível intermediário
- Conhecimento de serviços RESTful e verbos de ação HTTP

Este módulo faz parte destes roteiros de aprendizagem

- Desenvolver um aplicativo Web no ASP.NET Core que consome uma API

### 4.3 – Introdução.

O Razor Pages é um modelo de programação baseado em página que facilita a criação de páginas dinâmicas da Web.

As páginas do Razor Pages são criadas com base na sintaxe Razor, que é uma linguagem de modelagem poderosa que permite criar marcação HTML com código C#.

## 4.4 - Objetivos de aprendizagem

Depois de concluir este módulo, você poderá:

- Combinar HTML e C# para definir a lógica de renderização dinâmica da página.
- Renderização de respostas da API no Razor Pages.
- Criar páginas que executam operações HTTP.

## 4.5 - Explorar a estrutura do projeto do Razor Pages.

O Razor Pages é um modelo de programação do lado do servidor, centrado em páginas, voltado à criação de interfaces do usuário Web usando ASP.NET Core. Os benefícios incluem:

- Configuração fácil para aplicativos Web dinâmicos usando HTML, CSS e C#.
- Arquivos organizados por recurso para facilitar a manutenção.
- Combina a marcação com código C# do lado do servidor usando a sintaxe Razor.

A sintaxe Razor combina HTML e C# para definir a lógica de renderização dinâmica. Você pode usar variáveis e métodos C# em sua marcação HTML para gerar conteúdo da Web dinâmico no servidor em runtime. O Razor Pages não substituem HTML, CSS ou JavaScript. Ele é uma maneira de combinar essas tecnologias para criar conteúdo da Web dinâmico.

## 4.6 - Estrutura de projeto do Razor Pages.

A tabela a seguir descreve a estrutura do projeto que é gerada ao criar um novo projeto do Razor Pages, como executar o comando `dotnet new webapp`.

Expandir a tabela

Nome	Descrição
Pages/	Contém Razor Pages e os arquivos de suporte.
wwwroot/	Contém arquivos de ativos estáticos, como imagens, HTML, JavaScript e CSS.
<project_name>.csproj	Contém metadados de configuração do projeto, como dependências.
Module.vb	Serve como o ponto de entrada do aplicativo e configura o comportamento dele.

Veja a seguir mais informações sobre a estrutura do projeto, o roteamento de solicitações e os recursos compartilhados:

- Arquivos do Razor Pages e os arquivos de classe emparelhados PageModel: as páginas do Razor são armazenadas no diretório Páginas. Cada página Razor tem um arquivo `.cshtml` e um arquivo de classe `.cshtml.csPageModel`. A classe PageModel permite a separação da lógica e da apresentação de uma página Razor, define manipuladores de página para solicitações e encapsula propriedades de dados e lógica com escopo para sua página Razor.
- A estrutura de diretório Pages: o Razor Pages usa a estrutura de diretório Páginas como a convenção para as solicitações de roteamento. A seguinte tabela mostra como as URLs são mapeadas para

nomes de arquivo:

URL	Mapeada para a página Razor
www.example.com	Pages/Index.cshtml
www.example.com/Index	Pages/Index.cshtml
www.example.com/Privacy	Pages/Privacy.cshtml
www.example.com/Error	Pages/Error.cshtml

Layout e outros arquivos compartilhados: há vários arquivos que são compartilhados em várias páginas. Eles determinam elementos de layout comuns e importações de página. A tabela a seguir descreve a finalidade de cada arquivo.

Arquivo	Descrição
_ViewImports.cshtml	Importa namespaces e classes usados em várias páginas.
_ViewStart.cshtml	Especifica o layout padrão de todas as páginas Razor.
Pages/Shared/_Layout.cshtml	O layout especificado pelo arquivo _ViewStart.cshtml. Implementa elementos de layout comuns em várias páginas.
Pages/Shared/_ValidationScriptsPartial.cshtml	Fornecer funcionalidade de validação para todas as páginas.

#### 4.7 - Descobrir a sintaxe do Razor Pages

Razor é uma sintaxe de marcação para inserir código baseado .NET em páginas da Web. A sintaxe Razor é composta pela marcação Razor, por C# e por HTML. A sintaxe Razor é semelhante aos mecanismos de modelagem de várias estruturas de SPA (aplicativo de página única) JavaScript, como Angular, React, VueJs e Svelte.

A linguagem padrão do Razor é HTML. A renderização de HTML da marcação de Razor não é diferente de renderizar HTML de um arquivo HTML. O servidor renderiza a marcação HTML em .cshtml arquivos Razor inalterados.

#### 4.8 - Sintaxe Razor

O Razor dá suporte a C# e usa o símbolo **@** para fazer a transição de HTML para C#. O Razor avalia expressões em C# e as renderiza na saída HTML.

Quando um símbolo **@** é seguido por uma palavra-chave reservada do Razor, ele faz a transição para marcação específica do Razor. Caso contrário, a transição será para HTML simples. Para fazer o escape de um símbolo **@** na marcação do Razor, use um segundo símbolo **@**. O exemplo de código a seguir renderizaria o valor de `@Username` na saída HTML.

## Sintaxe

`<p>@Username</p>`


Renderiza o valor de @Username na saída HTML.

`<p>@@Username</p>`

Renderiza "@@Username" na saída HTML.

Conteúdo e atributos HTML que contêm endereços de email não tratam o símbolo @ como um caractere de transição. Por exemplo, os endereços de email no exemplo a seguir não são alterados pela análise do Razor:

HTML


 Copiar

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

## 4.9 - Adicionar código a uma página usando o caractere @

Os exemplos de código a seguir mostram como o caractere @ pode ser usado para implementar expressões internas, blocos de instrução única e blocos de várias instruções:

HTML

 Copiar


```
<!-- Single statement blocks -->
@{ var myMessage = "Hello World"; }

<!-- Inline expressions -->
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@{
    var greeting = "Welcome to our site!";
    var weekDay = DateTime.Now.DayOfWeek;
    var greetingMessage = greeting + " Today is: " + weekDay;
}
<p>The greeting is: @greetingMessage</p>
```

O exemplo de código a seguir mostra como usar uma combinação de código .NET e HTML para criar o corpo de uma tabela a partir de um modelo de dados. A instrução @foreach itera por meio do modelo de dados Model.FruitModels e gera uma linha de tabela que contém o nome da fruta e se ela está disponível.

CSHTML

 Copiar

```
@* Code is truncated for readability. *@
<tbody>
    @foreach (var obj in Model.FruitModels)
    {
        <tr>
            <td>@obj.name</td>
            <td>@obj.instock</td>
        </tr>
    }
</tbody>
```

## Leitura adicional

Para saber mais sobre a sintaxe Razor, visite as páginas a seguir:

- Referência da sintaxe Razor para ASP.NET Core
- Introdução à Programação Web do ASP.NET por meio da sintaxe Razor (C#)

1. O que é o Razor Pages?

- ☐ Um modelo de programação do lado do cliente para criar interface do usuário da Web com ASP.NET Core.
- ☐ Uma substituição para HTML, CSS e JavaScript.
- ☐ O Razor Pages é um modelo de programação do lado do servidor, centrado em páginas, voltado à criação de interfaces do usuário Web usando ASP.NET Core.

2. O que é a sintaxe Razor?

- ☐ É uma sintaxe de marcação para incorporar um código baseado em .NET em páginas da Web.
- ☐ Um mecanismo de modelagem para estruturas PHP de aplicativo de página única.
- ☐ É uma sintaxe de marcação para incorporar um código baseado em .NET em páginas da Web.

## 4.10 Resumo

Neste módulo, você aprendeu a:

- Combinar HTML e C# para definir a lógica de renderização dinâmica da página
- Renderização de respostas da API no Razor Pages
- Criar páginas que executam operações HTTP.