



Asp.NET Core

Bootcamp Speed Wiz Dev

Samuel Alves dos Santos

2021

Asp.NET Core

Bootcamp Speed Wiz Dev

Samuel Alves dos Santos

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

Sumário

Capítulo 1. Introdução ASP.NET Core	7
ASP.NET Core para Web API.....	7
Injeção de Dependência no ASP.NET Core	7
Manutenção da API com ASP.NET Core.....	8
Por que escolher o ASP.NET Core?	8
Capítulo 2. Protocolo HTTP	9
Como funciona o HTTP?.....	9
O que é uma requisição (Request) HTTP?	11
O que é um método HTTP?	12
O que é um cabeçalho de requisição HTTP?	12
O que há em um corpo de requisição HTTP?.....	12
O que é uma resposta (response) HTTP?	13
O que é um código de status HTTP?	13
O que são cabeçalhos de resposta HTTP?	14
O que há em um corpo de resposta (response) HTTP?	15
E sobre HTTPS?	15
Capítulo 3. API (Application Programming Interface)	16
O que é uma API?	16
O que é REST e SOAP API?	17
SOAP API vs REST API	17
Benefícios da REST API sobre SOAP	18
Usando métodos HTTP para serviços RESTFUL	19

Capítulo 4. Arquitetura REST	21
Servidor Cliente	21
Sem Estado (Stateless)	21
Cache	22
Interface Uniforme	22
Sistema em Camadas.....	22
Código sob demanda (opcional)	23
 Capítulo 5. Ambientes de Desenvolvimento	24
Visual Studio Community 2019	24
Visual Studio Code	25
Visual Studio For Mac	25
 Capítulo 6. Criando um projeto no Visual Studio	27
Setup	27
Gerenciador de Soluções	30
Controllers.....	34
Arquivo AppSettings.json	36
Arquivo AppSettings.Development.json	37
Classe Program.cs.....	38
Classe Startup.cs.....	39
Método ConfigureServices()	40
Método Configure()	40
Classe WeatherForecast.cs	40
 Capítulo 7. Controllers (Controladores)	42
Classes Controller x ControllerBase no ASP.NET Core	44

Classe ControllerBase	44
Atributo [ApiController]	45
Capítulo 8. Rotas.....	46
Roteamento Convencional.....	46
Roteamentos por Atributo	47
Métodos HTTP	48
Prefixos de Rota	49
Restrições de Rota	52
Inferência de Parâmetros (Binding)	52
Capítulo 9. Action Results	54
Tipo Específico	54
Retornar IEnumerable<T> ou IEnumerableAsync<T>	55
Tipo IActionResult.....	57
Action Síncrona.....	57
Action Assíncrona	58
Capítulo 10. Convenções	60
Aplicando Convenções Web Api.....	62
Atributos para Tipos de Response	65
Capítulo 11. CRUD (Create, Read, Update e Delete)	66
Adicionar um contexto de Banco de Dados	67
Classe do contexto DbContext e Geração de dados	68
Classes Startup.cs e Program.cs.....	70
Controller LivroController.cs.....	72

Método POST	72
Métodos GET	73
Método PUT	75
Métodos DELETE	76
Roteamento	76
Instale o Postman	77
Capítulo 12. Segurança	79
Autenticação	79
Autorização	80
Autenticação vs Autorização	82
Token	83
JWT - JSON Web Token	84
Capítulo 13. Documentando APIs	89
Swagger / Open API	89
Especificação Swagger	89
Swagger UI	90
Integrado a UI Swagger	90
Referências	95

Capítulo 1. Introdução ASP.NET Core

ASP.NET Core para Web API

A tecnologia legada da Microsoft existe desde o .NET framework 1.0 até os dias atuais com o .NET Core. Com a criação de uma nova arquitetura da web, o ASP.NET Core, elimina uma grande parte da tecnologia legada para oferecer aos desenvolvedores um desempenho aprimorado, mesmo ainda usando o assembly System.Web que hospeda todas as bibliotecas de algumas abordagens de desenvolvimento como o WebForms, por exemplo. Com as dependências legadas sendo removidas, o desenvolvedor pode criar melhores resultados e isto permite uma arquitetura de saída ou execução em diferentes plataformas (**cross-platform**), ou seja, ao utilizar o ASP.Net Core, as soluções criadas funcionarão tão bem no Linux quanto no Windows.

Injeção de Dependência no ASP.NET Core

Um dos melhores recursos do ASP.Net Core é a injeção de dependência (**DI**). Se você já usou soluções .Net Framework ou ASP.Net anteriormente, você saberá que eles também tinham DI. No entanto, eles vieram de fornecedores comerciais de terceiros ou bibliotecas de código aberto. Isso funcionou, mas grande parte dos desenvolvedores .NET tinham uma grande curva de aprendizado. Além disso, as bibliotecas DI tinham processos diferentes.

Com o ASP.Net Core, o DI é integrado à estrutura desde o início. Isso torna a vida dos desenvolvedores muito mais fácil porque é simples e você pode tirá-lo da própria estrutura. Esse componente é vital na criação de APIs porque permite que os desenvolvedores tenham uma experiência tranquila e contínua ao separar camadas de arquitetura.

Manutenção da API com ASP.NET Core

A manutenção para qualquer processo de engenharia não é algo simples ou pequena, mas com ASP.Net Core os recursos são integrados para garantir que os desenvolvedores possam fazer a manutenção com facilidade. Os desenvolvedores serão capazes de encontrar erros, inspecionar componentes defeituosos sem substituir as peças que estão funcionando, evitar funcionamento incorreto, atender a novos requisitos, maximizar a vida útil de um produto, lidar com mudanças no ambiente e facilitar a manutenção futura.

Ao construir uma API moderna, a manutenção deve fazer parte de uma arquitetura bem planejada, pois não adotar essa premissa causará dificuldades no processo de construção dela. A capacidade de manutenção é um problema de longo prazo e planejar uma arquitetura visando o futuro é o cenário ideal.

Por que escolher o ASP.NET Core?

Milhões de desenvolvedores usam e novos estão cada vez mais utilizando o ASP.Net Core para criar aplicações web e móveis, pois a abordagem fornece uma estrutura muito mais simples e modular. Além de fornecer um conceito unificado para a construção de UI e APIs para web, também possui uma estrutura totalmente arquitetada para testes, além de possuir estruturas modernas integradas do lado do cliente com um sistema de configuração baseado em ambiente totalmente pronto para nuvem.

ASP.Net Core também pode ser desenvolvido e executado em diferentes plataformas como Windows, macOS e Linux, oferecendo suporte para hospedagem de serviços de *Chamada de Procedimento Remoto (RPC)* usando **gRPC** e pode ser hospedado em Kestrel, IIS, HTTP.sys, Nginx, Apache e Docker. Por último, traz o seu código aberto e foco na comunidade, com um pipeline de requisição HTTP leve, alto desempenho e modularidade e também ferramentas que simplificam o desenvolvimento moderno da web.

Capítulo 2. Protocolo HTTP

É impossível falar de REST sem falar do protocolo HTTP, antes de atingirmos o ápice da tecnologia no curso atual precisamos responder a esta pergunta, o que é HTTP?

A abreviação HTTP vem do Inglês “**Hypertext Transfer Protocol**” que é a fundação da internet e sua principal forma de recebimento é o envio de informações através deste veículo.

Como funciona o HTTP?

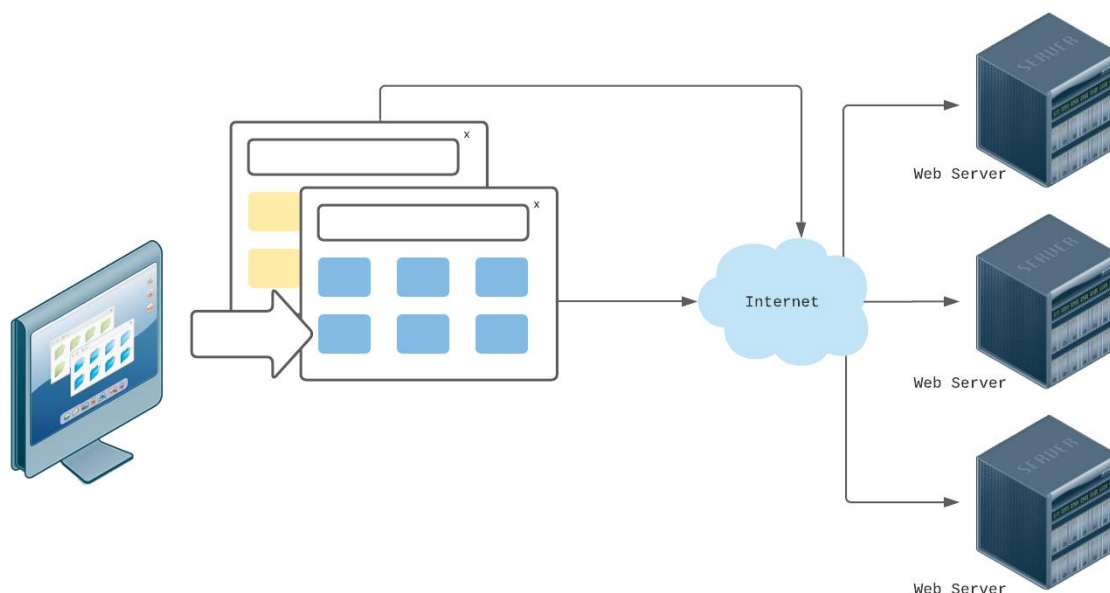
Assim que um usuário abre seu navegador da web, ele está indiretamente usando HTTP. HTTP é um protocolo de aplicação executado sobre o conjunto de protocolos TCP/IP, que formam a base da Internet. A versão mais atual do HTTP é o HTTP/2, publicado em maio de 2015 que é uma alternativa ao seu predecessor, o HTTP 1.1, que mesmo não sendo o atual não o torna obsoleto.

Por meio do protocolo HTTP, os recursos são trocados entre dispositivos clientes e servidores pela Internet. Os dispositivos clientes enviam requisições aos servidores dos recursos necessários para carregar uma página da web ou habilitar um processo de comunicação; os servidores enviam respostas de volta ao cliente para atender às requisições. Neste processo requisições e respostas compartilham subdocumentos que podem ser dados em imagens, texto, layouts de texto etc. Um bom exemplo seria os documentos reunidos por um navegador da web do cliente para exibir uma página HTML ou resultados de processamento esperados por um outro serviço/sistema.

Além dos artefatos produzidos que podem ser servidos, um servidor da web contém um **daemon (processo)** HTTP, que espera por requisições HTTP e as trata quando chegam. Um navegador da web, um sistema ou serviço são clientes HTTP que enviam requisições aos servidores. Baseando-se em um navegador quando a url é acionada, é criada uma requisição HTTP e enviado para o endereço de protocolo

da Internet (endereço IP) indicado pela URL. O daemon HTTP no servidor de destino recebe a requisição e envia de volta o arquivo solicitado ou os arquivos associados à requisição.

Figura 1 - Comunicação HTTP.



Os dispositivos clientes usam HTTP para se comunicar com servidores on-line e acessar páginas da web.

Utilizando o mesmo exemplo anterior, mas entendendo o processo através de um verbo HTTP, podemos pensar na situação em que o usuário digita o endereço da web e o computador envia uma requisição "**GET**" a um servidor que hospeda esse endereço. Essa requisição **GET** é enviada usando HTTP e informa ao servidor que o usuário está procurando pelo código HTML (Hypertext Markup Language) usado para estruturar e dar à página de login sua aparência e comportamento. O texto dessa página de login é incluído na resposta HTML, mas outras partes da página - especialmente suas imagens e vídeos - são solicitadas por requisições e respostas HTTP separadas. Quanto mais requisições forem feitas - por exemplo, para chamar uma página com várias imagens - mais tempo o servidor levará para responder a essas requisições e para o sistema do usuário carregar a página.

Cada interação entre o cliente e o servidor é chamada de mensagem. As mensagens HTTP são requisições (**requests**) ou respostas (**responses**). Os dispositivos clientes enviam requisições (**requests**) HTTP aos servidores, que respondem enviando respostas (**responses**) HTTP de volta aos clientes.

O que é uma requisição (Request) HTTP?

A *Request* ou requisição, traduzindo diretamente para português, é o pedido que um cliente realiza ao servidor. Esse pedido contém uma série de dados que são usados para descrever exatamente o que o cliente precisa. Vamos pensar que um cliente precisa cadastrar um novo produto, ele deve passar todos os dados necessários para o cadastro acontecer de maneira correta, inclusive os dados que foram digitados pelo usuário em um formulário, no caso de uma aplicação web. No navegador, toda vez que trocamos de página ou apertamos <Enter> na barra de endereço uma nova requisição é feita. Independente se estamos apenas pedindo a exibição de uma página, cadastrando um novo recurso, atualizando ou excluindo.

Cada requisição HTTP carrega consigo uma série de dados codificados que contém diferentes tipos de informações. Uma requisição HTTP típica contém:

1. Versão HTTP
2. URL
3. Método HTTP
4. Cabeçalhos de requisição HTTP
5. Corpo HTTP opcional.

O que é um método HTTP?

Um método HTTP, às vezes chamado de verbo HTTP, indica a ação que a requisição HTTP espera do servidor consultado. Por exemplo, dois dos métodos HTTP mais comuns são 'GET' e 'POST'; quando o usuário digita um endereço e aperta enter na barra de endereço do navegador, ele realiza uma requisição do tipo GET, enquanto uma requisição 'POST' normalmente indica que o cliente está enviando informações para o servidor da web (como informações de formulário, por exemplo, um nome de usuário e senha enviados).

O que é um cabeçalho de requisição HTTP?

Os cabeçalhos HTTP contêm informações de texto armazenadas em pares de chave-valor e são incluídos em cada requisição HTTP (e resposta, mais sobre isso posteriormente). Esses cabeçalhos comunicam informações essenciais, como qual navegador o cliente está usando, quais dados estão sendo solicitados, o método utilizado, autoridade e entre outras.

Figura 2 - Representação do cabeçalho na requisição.

```
▼ Request Headers
:authority:
:method: GET
:path:
:scheme: https
accept: application/json, text/plain, */*
accept-encoding: gzip, deflate, br
accept-language: en-GB,en-US;q=0.9,en;q=0.8,es;q=0.7,pt;q=0.6
```

O que há em um corpo de requisição HTTP?

O corpo de uma requisição é a parte que contém o "corpo" de informações que a solicitação está transferindo. O corpo de uma requisição HTTP contém todas

as informações enviadas ao servidor da web, como nome de usuário e senha, ou quaisquer outros dados inseridos em um formulário.

O que é uma resposta (response) HTTP?

Vimos que o cliente envia uma requisição (**Request**) ao servidor. Essa requisição possui todas as informações acerca do que o cliente espera receber de volta. O servidor web ao receber essas informações precisa enviar uma resposta ao cliente, nesse ponto entra o response. A resposta (**Response**) nada mais é do que a resposta que o servidor envia ao cliente. Essa resposta pode conter os dados que realmente o cliente esperava receber ou uma resposta informando que alguma coisa deu errado.

Uma resposta HTTP típica contém:

1. Código de status HTTP
2. Cabeçalhos de resposta HTTP
3. Corpo HTTP opcional

O que é um código de status HTTP?

Os códigos de status HTTP são códigos de 3 dígitos usados com mais frequência para indicar se uma requisição HTTP foi concluída com êxito. Os códigos de status são divididos nos 5 blocos a seguir:

1. **1xx** Informativo
2. **2xx** Sucesso
3. Redirecionamento **3xx**
4. **4xx** Erro do Cliente

5. 5xx Erro do servidor

O “xx” refere-se a diferentes números entre 00 e 99.

Os códigos de status que começam com o número ‘2’ indicam um sucesso. Por exemplo, depois que um cliente solicita uma página da web, as respostas mais comumente vistas têm um código de status de ‘**200 OK**’, indicando que a requisição foi devidamente concluída.

Se a resposta começar com ‘4’ ou ‘5’, isso significa que houve um erro. Um código de status que começa com um ‘4’ indica um erro do lado do cliente (é muito comum encontrar um código de status ‘**404 NÃO ENCONTRADO**’ ao cometer um erro de digitação em um URL). Um código de status começando com ‘5’ significa que algo deu errado no servidor. Os códigos de status também podem começar com um ‘1’ ou um ‘3’, que indicam uma resposta informativa e um redirecionamento, respectivamente.

O que são cabeçalhos de resposta HTTP?

Assim como uma requisição HTTP, uma resposta HTTP vem com cabeçalhos que transmitem informações importantes, como o idioma e o formato dos dados enviados no corpo da resposta.

Figura 3 - Representação do cabeçalho na resposta.

▼ Response Headers

```
access-control-allow-origin: *
content-length: 92273
content-type: application/json
date: Wed, 06 Oct 2021 17:13:14 GMT
x-amz-apigw-id: Gy7enFoISwMFgrA=
x-amzn-requestid: bd93feeb-689b-4cbb-9172-781743710007
x-amzn-trace-id: Root=1-615dd92a-03039f272db6483e45f74911
```

O que há em um corpo de resposta (response) HTTP?

Respostas HTTP bem-sucedidas para requisições 'GET' geralmente têm um corpo que contém as informações solicitadas.

E sobre HTTPS?

Ao contrário do HTTP, o HTTPS usa um certificado seguro de um fornecedor terceirizado para proteger uma conexão e verificar se o site é legítimo. Esse certificado é conhecido como certificado SSL.

SSL é a abreviação de "**Secure Sockets Layer**" (camada de soquetes seguras). É isso que cria uma conexão segura e criptografada entre um navegador, serviço ou integração e um servidor, que protege a camada de comunicação entre os dois.

Esse certificado criptografa uma conexão com um nível de proteção designado no momento da compra de um certificado SSL. Um certificado SSL fornece uma camada extra de segurança para dados confidenciais que você não quer que terceiros acessem.

HTTPS foi desenvolvido pela Netscape. A migração de HTTP para HTTPS é considerada benéfica, pois oferece uma camada adicional de segurança e confiança.

Capítulo 3. API (Application Programming Interface)

O que é uma API?

Você já deve ter ouvido falar, pelo menos alguma vez, sobre este termo chamado “API” e pode ter pensado o que realmente vem a ser uma API e como ela pode ajudar os desenvolvedores no seu trabalho do dia a dia.

Uma API (**Application Programming Interface**) pode ser definida como um conjunto de padrões que permite a construção de aplicações ou a abstração que conecta aplicações, podendo ser utilizada nos mais variados tipos de negócios.

Apesar de ter essa integração, quando um usuário estiver navegando em um site que tem uma integração com uma API, por exemplo, o usuário nem saberá que sua aplicação está fazendo uma comunicação com uma API, pois ela é invisível ao usuário comum, já que ele enxerga apenas a interface dos softwares e aplicações.

Com a API você tem uma interface para que um sistema se comunique com outro sistema, compartilhando ações e ferramentas. A comunicação é feita através de um processo de codificação que define comportamentos específicos.

Além de todas as características supracitadas, ainda há um elemento muito importante associado a APIs que é a produtividade para as partes que a integram, pois o conceito de responsabilidade segregada permite desenvolvimentos independentes.

Imagine que você está navegando por um site para a compra de um produto e depois de escolher você deseja saber quanto isso custará para chegar até o seu endereço e quais são as opções de envio (PAC, SEDEX etc.) e o tempo estimado. Quando você digita o seu CEP para esse cálculo de frete, o site está utilizando provavelmente uma API dos Correios, isto é, a integração desses sistemas se dá por meio de uma API.

Muitas pessoas podem confundir uma API com um outro termo muito falado ultimamente, que são os endpoints. Um endpoint é basicamente o que um serviço

exposto e esse serviço pode ser acessado por uma aplicação, por isso muitas vezes acaba sendo confundido com uma API, mas vale ressaltar que não é. Um endpoint contém três principais características: Endereço (**Address**) (onde o serviço está hospedado), Ligação (**Binding**) (como o serviço pode ser acessado) e Contrato (**Contract**) (o que tem no serviço). Além disso, uma API pode existir sem um endpoint e vice-versa.

O que é REST e SOAP API?

SOAP é um protocolo para a maioria dos serviços da Web e significa **Simple Object Access Protocol**. No entanto, à medida que a Internet evoluiu, os desenvolvedores precisaram de uma maneira de construir rapidamente aplicativos móveis e web mais leves. Assim, foi desenvolvida a arquitetura alternativa da REST API.

A sigla REST (**Representational State Transfer**), em português significa “Transferência de Estado Representacional”, concebido como uma abstração da arquitetura da web, trata-se de um conjunto de princípios para a criação de um projeto com interfaces bem definidas. Mas ainda é possível ver que algumas decisões corporativas ainda escolhem o SOAP para a comunicação de serviços com sistemas legados.

SOAP API vs REST API

O REST opera por meio de uma interface consistente e solitária para acessar recursos nomeados. É mais comumente usado quando você está expondo uma API pública na Internet. O SOAP, por outro lado, expõe os componentes da lógica da aplicação como serviços em vez de dados. Além disso, opera por meio de diferentes interfaces. Para simplificar, o REST acessa os dados enquanto o SOAP executa operações por meio de um conjunto mais padronizado de padrões de mensagens. Ainda assim, na maioria dos casos, REST ou SOAP podem ser usados para atingir o

mesmo resultado (e ambos são infinitamente escaláveis), com algumas diferenças em como você os configura.

O SOAP foi originalmente criado pela Microsoft e existe há muito mais tempo que o REST. Isso lhe dá a vantagem de ser um protocolo legado estabelecido. Mas o REST já existe há um bom tempo também. Além disso, ele entrou em cena como uma forma de acessar serviços da web de uma maneira muito mais simples do que seria possível com o SOAP usando HTTP.

Benefícios da REST API sobre SOAP

Em termos gerais, o resultado do REST é que, para a grande maioria dos casos, usar a API REST significa que você pode fazer mais, mais rápido e com menos largura de banda.

REST funciona melhor para clientes modernos (como um navegador da web), portanto, se você estiver construindo um aplicativo em nuvem, REST é o tipo de API preferível para você. Funciona com todos os tipos de equipamentos de rede e com uma variedade de tipos de serviço (Java, .NET, PHP, Ruby, HTTP estático etc.). E ao criar esse aplicativo você precisará da flexibilidade do REST. Imagine o débito tecnológico de ter que manter seu aplicativo enquanto fica preso a requisitos de infraestrutura específicos por causa de como você configurou o acesso à SOAP API.

Da mesma forma, você deseja ser o mais flexível possível sobre o tipo de cliente no qual pode executar seu aplicativo na maioria dos casos; mesmo que agora seu aplicativo seja executado em uma infraestrutura conhecida, nem sempre será o caso. Por causa da implementação leve e flexível do REST, este é muito amigável à infraestrutura - balanceadores de carga de rede, firewalls, proxies etc. são todos otimizados para tráfego RESTful porque são otimizados para tráfego HTTP/S.

Por fim, as requisições REST não têm estado e, portanto, por definição, são altamente escaláveis, pois dependem de vários protocolos de comunicação. Eles também são muito mais eficientes - os serviços SOAP sempre retornam XML, mas

nosso serviço REST retorna JSON, que é o padrão de fato, e essas cargas JSON são menores do que suas contrapartes XML. Se você incluir a sobrecarga do envelope SOAP nesse cálculo, nossas cargas úteis REST + JSON serão drasticamente menores.

Usando métodos HTTP para serviços RESTFUL

Os verbos HTTP compreendem uma parte importante de nossa restrição de "interface uniforme" (separação de responsabilidades entre o cliente e o servidor) e fornecem a contrapartida de ação para o recurso baseado em substantivo. Os verbos HTTP primários ou mais comumente usados (ou métodos, como são chamados corretamente) são POST, GET, PUT, PATCH e DELETE. Eles correspondem às operações de criação, leitura, atualização e exclusão (ou CRUD), respectivamente. Existem vários outros verbos também, mas são usados com menos frequência e destes destacamos OPTIONS e HEAD.

Abaixo está uma tabela que resume os valores de retorno recomendados dos métodos HTTP primários em combinação com os URIs de recursos:

Tabela 1 - Principais métodos HTTP.

HTTP VERB	CRUD	COLEÇÃO (/clientes)	ITEM ESPECÍFICO (/clientes/{id})
POST	Create (Criar)	201 (Criado), no cabeçalho de resposta contém um atributo 'Location' com link o recurso recém-criado como /clientes/{id} contendo novo ID.	404 (não encontrado), 409 (conflito) se o recurso já existir.
GET	Read (Ler)	200 (OK), lista de clientes. Use paginação, classificação e filtragem para navegar por grandes listas.	200 (OK), cliente único. 404 (não encontrado), se ID não encontrado ou inválido.
PUT	Update/Replace	405 (Método não permitido), a menos que você queira atualizar/substituir todos os recursos em toda a coleção.	200 (OK) ou 204 (sem conteúdo). 404 (não encontrado), se ID não encontrado ou inválido.
PATCH	Update/Modify	405 (Método não permitido), a menos que você queira modificar parcialmente a coleção em si.	200 (OK) ou 204 (sem conteúdo). 404 (não encontrado), se ID não encontrado ou inválido.

DELETE	Delete (Deletar)	405 (Método não permitido), a menos que você queira excluir toda a coleção - o que não é desejável com frequência.	200 (OK). 404 (não encontrado), se ID não encontrado ou inválido.
---------------	------------------	--	---

Capítulo 4. Arquitetura REST

Embora a maioria das APIs afirme ser RESTful, elas ficam aquém dos requisitos e restrições. Existem seis restrições principais para o design da API REST que você deve conhecer ao decidir se este é o tipo de API correto para o seu projeto.

Servidor Cliente

A restrição cliente-servidor trabalha com o conceito de que o cliente e o servidor devem ser separados um do outro e ter permissão para evoluir de forma individual e independente. Em outras palavras, devo ser capaz de fazer alterações em meu aplicativo móvel sem afetar a estrutura de dados ou o design do banco de dados no servidor. Ao mesmo tempo, devo ser capaz de modificar o banco de dados ou fazer alterações em meu aplicativo de servidor sem impactar o cliente móvel. Isso cria uma separação de interesses, permitindo que cada aplicativo cresça e dimensione independentemente um do outro e permitindo que sua organização cresça de forma rápida e eficiente.

Sem Estado (Stateless)

As APIs REST são sem estado, o que significa que as chamadas podem ser feitas independentemente umas das outras e cada chamada contém todos os dados necessários para ser concluída com êxito. Uma API REST não deve depender de dados armazenados no servidor ou nas sessões para determinar o que fazer com uma chamada, mas apenas nos dados fornecidos na própria chamada. As informações de identificação não estão sendo armazenadas no servidor ao fazer chamadas. Em vez disso, cada chamada tem os dados necessários em si, como a chave da API, token de acesso, ID do usuário etc. Isso também ajuda a aumentar a confiabilidade da API por ter todos os dados necessários para fazer a chamada, em vez de depender de uma série de chamadas com estado de servidor para criar um objeto, o que pode resultar em falhas parciais. Em vez disso, para reduzir os requisitos

de memória e manter sua aplicação o mais escalonável possível, uma API RESTful requer que qualquer estado seja armazenado no cliente - não no servidor.

Cache

Como uma API sem estado pode aumentar a sobrecarga de requisições ao lidar com grandes cargas chamadas de entrada e saída, uma API REST deve ser projetada para incentivar o armazenamento de dados armazenáveis em cache. Isso significa que quando os dados podem ser armazenados em cache, a resposta deve indicar que os dados podem ser armazenados até um certo tempo (expira em) ou, nos casos em que os dados precisam ser em tempo real, que a resposta não deve ser armazenada em cache pelo cliente. Ao habilitar essa restrição crítica, você não apenas reduzirá muito o número de interações com sua API, reduzindo o uso do servidor interno, mas também fornecerá aos usuários da API as ferramentas necessárias para fornecer as aplicações mais rápidas e eficientes possíveis.

Interface Uniforme

A chave para o desacoplamento do cliente do servidor é ter uma interface uniforme que permite a evolução independente do aplicativo sem ter os serviços, modelos ou ações do aplicativo fortemente acoplados à própria camada de API. A interface uniforme permite que o cliente converse com o servidor em um único idioma, independente da arquitetura do backend de qualquer um deles. Essa interface deve fornecer um meio de comunicação padronizado e imutável entre o cliente e o servidor, como o uso de HTTP com recursos de URI, CRUD (Criar, Ler, Atualizar, Excluir) e JSON.

Sistema em Camadas

Um sistema em camadas é um sistema distribuído em camadas, cada qual com responsabilidades específicas. Um framework do tipo MVC (*Model-View-*

Controller), MVVM (*Model-View-ViewModel*) ou MVP (*Model-View-Presenter*), possui camadas com responsabilidades específicas, com modelos representando como os dados serão trafegados, validados e materializados, sendo a *Controller*, *ViewModel* ou *Presenter* responsável pelo controle das requisições e oferecendo uma resposta que será visualizada na saída, com as camadas interagindo umas com as outras através de mensagens. No design REST segue-se a mesma premissa, com camadas distintas da arquitetura trabalhando em conjunto para uma arquitetura e hierarquia a fim de que a aplicação seja mais escalável e modular.

Código sob demanda (opcional)

Bem, essa restrição é opcional. Na maioria das vezes, você enviará representações estáticas de recursos na forma de XML ou JSON. Mas quando você precisar, está livre para retornar o código executável para dar suporte a parte do seu aplicativo, por exemplo, os clientes podem chamar sua API para obter o código de renderização de um componente de interface UI. Isso é permitido.

Capítulo 5. Ambientes de Desenvolvimento

Visual Studio Community 2019

Figura 4 - Logo Visual Studio 2010.



Fonte: Microsoft.

O Visual Studio é o ambiente de desenvolvimento completo da Microsoft com suporte a várias linguagens e com diferentes tipos de projeto disponíveis. Com esse ambiente é possível desenvolver aplicações desktop, mobile, web e IoT (internet das coisas). Há algumas versões disponíveis, como Visual Studio Professional, Visual Studio Enterprise e o Visual Studio Community 2019, que é a versão gratuita da ferramenta e que será utilizada para o desenvolvimento do projeto deste curso. O link para download é <https://www.visualstudio.com/pt-br/downloads/>.

Durante o processo de instalação o .Net Framework será automaticamente instalado, permitindo que seu computador possua todas as principais bibliotecas e dependências pronta para o desenvolvimento e execução dos projetos.

Além do Visual Studio, existem outras versões que podem ser utilizadas em contextos e sistemas operacionais diferentes para o desenvolvimento, como Visual Studio Code e Visual Studio for Mac.

Visual Studio Code

Figura 5 - Logo Visual Studio Code



Fonte: Microsoft.

O Visual Studio Code é um editor de textos, geralmente visto como uma IDE devido à capacidade e recursos disponíveis. Possui uma grande biblioteca de plugins que podem ser adicionados para facilitar o processo de desenvolvimento, além de execução de testes, debugging e controle de versão. O VS Code é open source e multiplataforma, com diferentes tipos de projeto incluindo ASP.NET Core Web API, além de possuir suporte nativo a TypeScript, JavaScript e frameworks, JSON, YAML, HTML, CSS, entre outras tecnologias. Segue o link para download (<https://code.visualstudio.com/download>).

Visual Studio For Mac

Figura 6 - Logo Visual Studio for MAC.



Fonte: Microsoft.

Com uma versão prévia batizada como Xamarin Studio, o Visual Studio for Mac é uma melhoria do antecessor, sendo um ambiente de desenvolvimento integrado .NET no sistema operacional MacOS, utilizado para a construção de

aplicações .NET como Web API, MVC, Xamarin e outros. É um editor consistente, é utilizado para editar, depurar, testar e publicar código, integrado totalmente com XCode quando se trata de aplicações mobile iOS. Visualmente é uma IDE completa com os principais elementos e inclui recursos de intellisense (autocompletar).

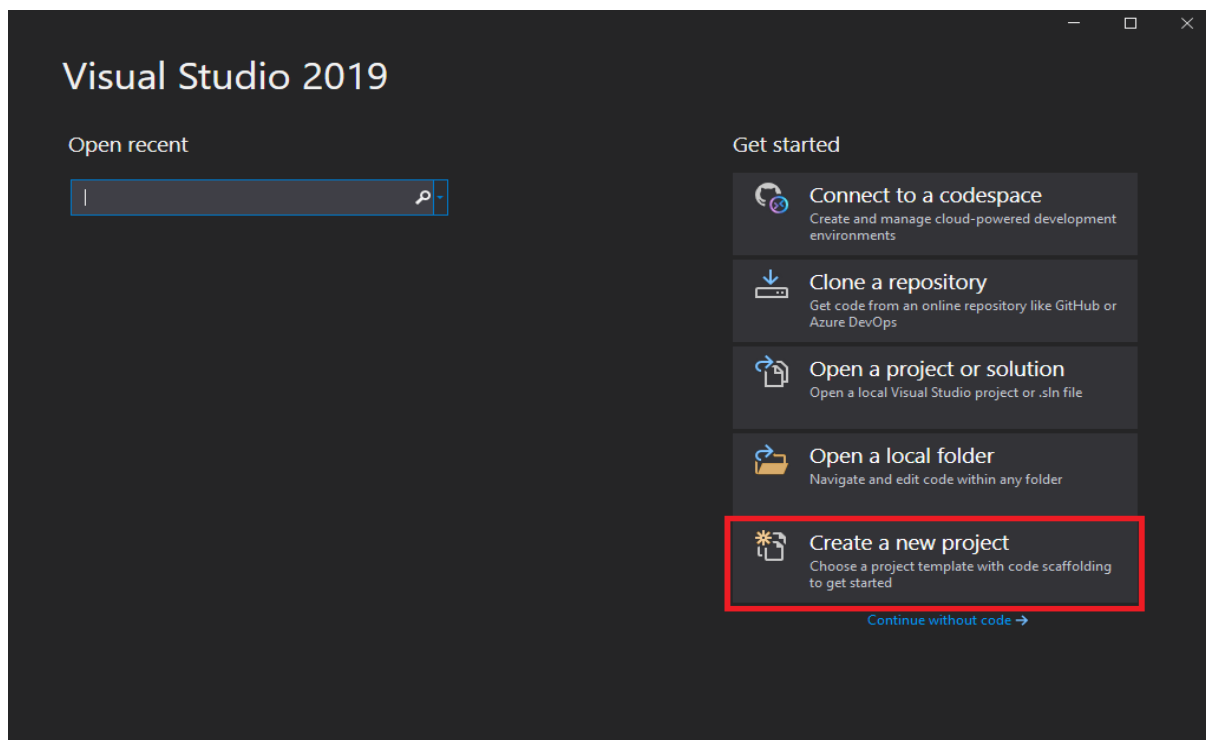
Segue o link para download <https://www.visualstudio.com/pt-br/downloads/>.

Capítulo 6. Criando um projeto no Visual Studio

Setup

- Abra o Visual Studio Community 2019;
- Na janela **Principal**, escolha na última opção **Criar um novo projeto**.

Figura 7 - Novo projeto no Visual Studio Community 2019.



- Na janela **Criar um novo projeto**, defina algumas opções para facilitar a busca pelo projeto específico, que neste caso será **C#** na lista de idiomas, **Windows** na lista de plataformas e **Web** na lista de tipos de projeto.
- Após selecionar as opções supracitadas, escolha o template ASP.NET Core Web API e, em seguida, escolha **Avançar**.

Figura 8 - Definindo opções de projetos.

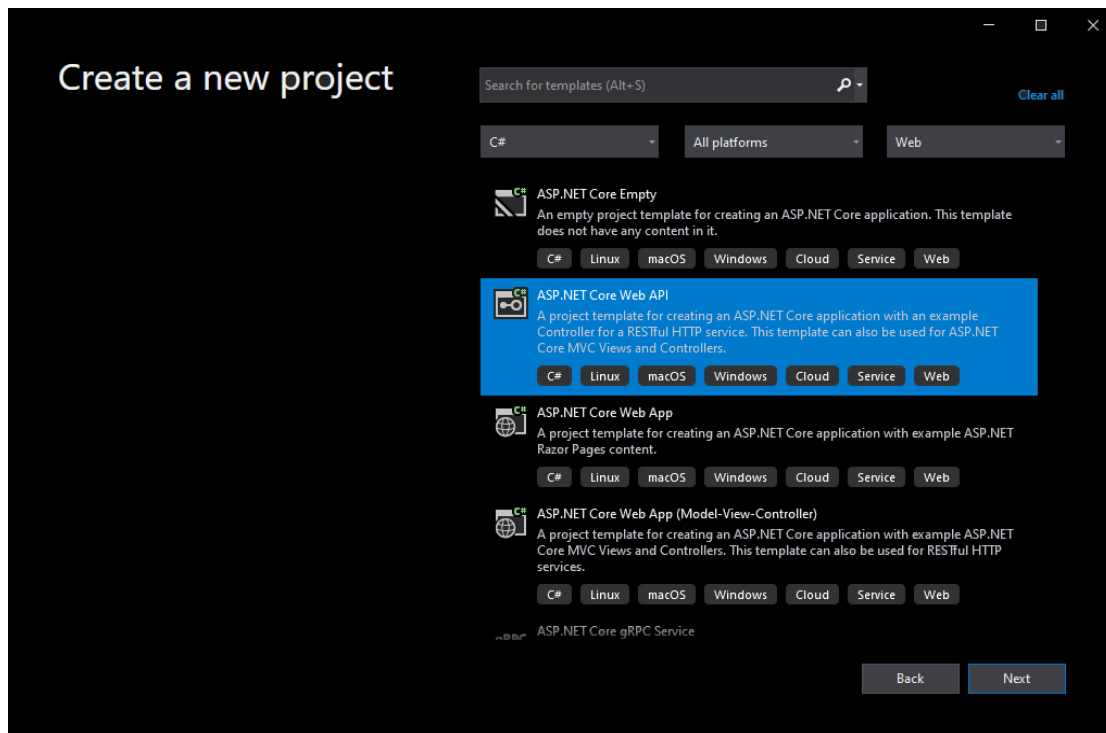
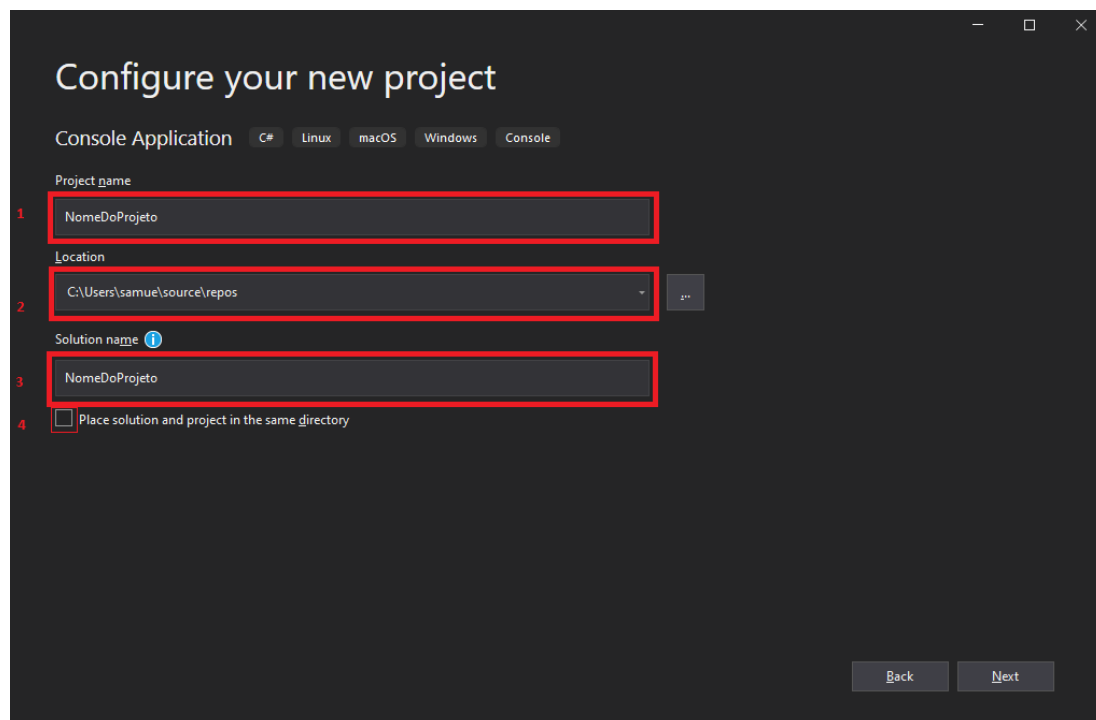
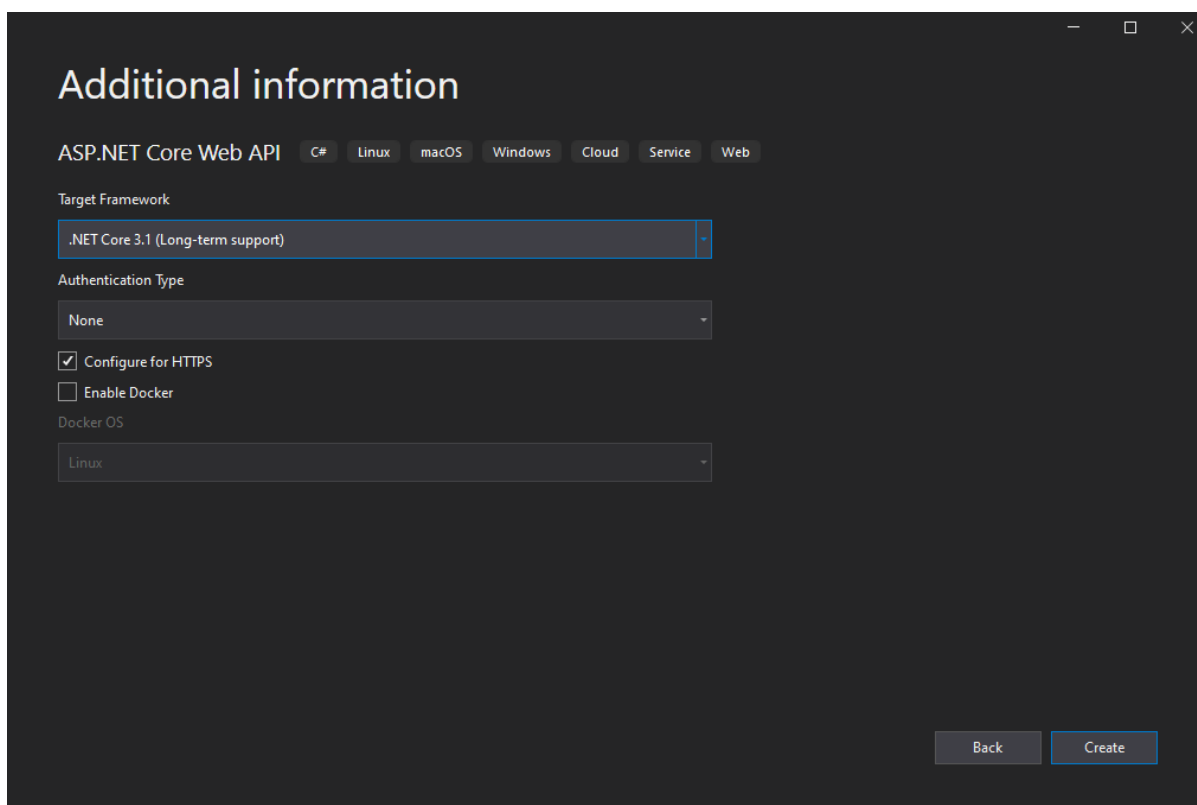


Figura 9 - Nomeando projeto e selecionando localização.



- Na janela **Configurar seu novo projeto**, defina um nome para o projeto e selecione uma localização. O nome do projeto será atribuído ao nome da solução e pode ser alterado ou mantido. A solução é um agrupador de projetos. Há um checkbox na parte inferior da janela que nos permite selecionar se queremos que os projetos fiquem dentro do mesmo diretório. Após as definições, clique em **Avançar**.

Figura 10 - Configurações Adicionais.



- Na janela **Informações Adicionais** escolha o framework.
- Para o tipo de autenticação, deixaremos como Nenhum (**None**).
 - Existem outras opções disponíveis:
 - **Microsoft Identity Platform:** A plataforma de identidade da Microsoft ajuda você a construir aplicativos nos quais seus usuários e clientes podem entrar usando suas identidades

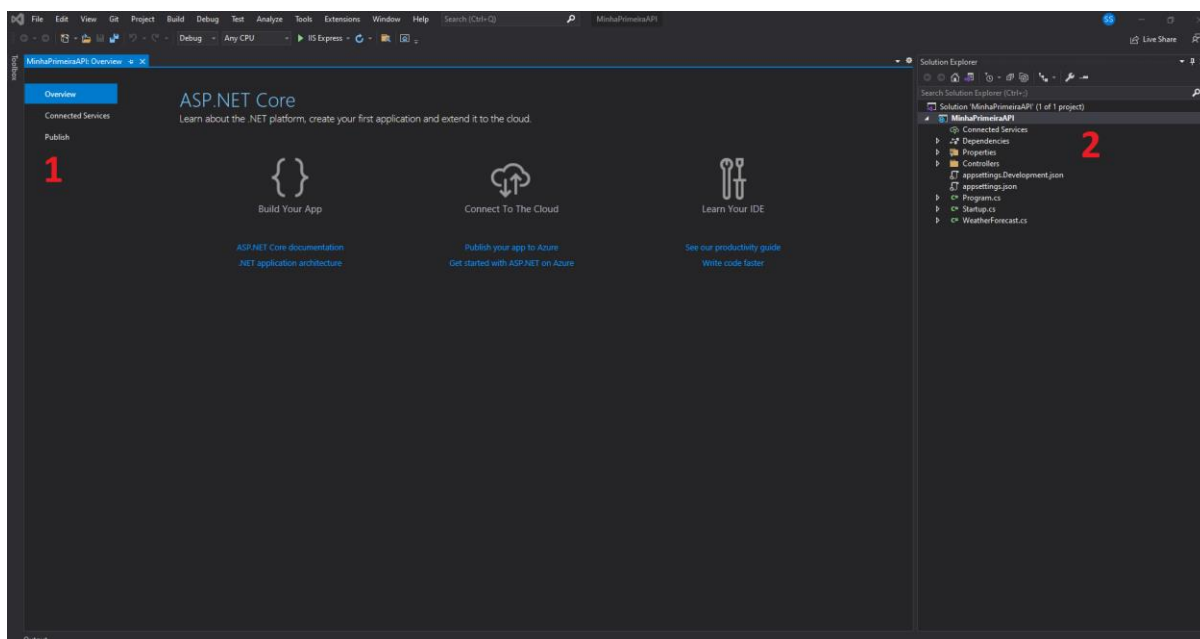
Microsoft ou contas sociais e fornecer acesso autorizado às suas próprias APIs ou APIs da Microsoft como o Microsoft Graph.

- **Windows:** consiste em permitir o acesso desde que o usuário faça parte do domínio/máquina. Essa opção torna o gerenciamento simples, facilita o Single Sign-On (SSO) interno, mas é útil em um ambiente controlado, como uma intranet.
- A opção **Configurar HTTPS** permite que a solução aplique uma configuração de HTTPS e quando a aplicação é executada, essa será escutada em 2 portas (HTTP e HTTPS).
- A opção **Enable Docker** permite colocar em contêiner um projeto ASP.NET Core. Há suporte para contêineres Linux e Windows. Ao adicionar suporte Docker a um projeto, escolha um contêiner Windows ou Linux. Não selecionaremos essa opção no momento atual.

Gerenciador de Soluções

Após a criação do projeto - isto pode levar algum tempo devido a adição das bibliotecas nativas - o ambiente abaixo é apresentado. A primeira área (grande área) é onde editamos o código e na segunda área podemos agregar diferentes painéis que facilitam o processo de desenvolvimento, mas no exemplo abaixo temos o explorador da solução que nos permite localizar os principais elementos da solução, incluindo os arquivos. Este é o lugar em que os arquivos são adicionados após a construção.

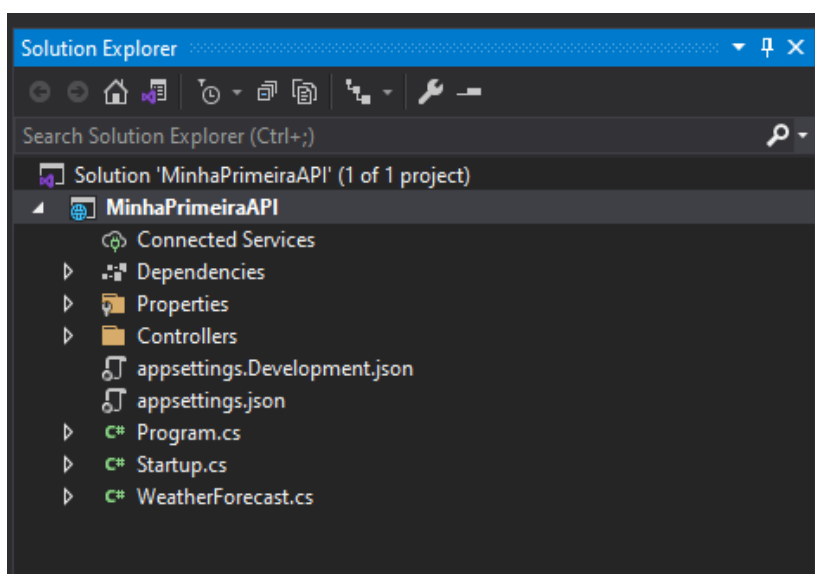
Figura 11 - Solução criado e apresentando 2 painéis principais.



Arquivos e pastas do projeto ASP.NET Core são sincronizados com arquivos e pastas físicas. Se você adicionar um novo arquivo ou pasta na pasta do projeto, ele refletirá diretamente no gerenciador de soluções. Você não precisa adicioná-lo explicitamente ao projeto clicando com o botão direito do mouse no projeto.

O gerenciador de soluções apresenta a seguinte estrutura:

Figura 12 - Gerenciador de Soluções.



Podemos editar as configurações .csproj clicando com o botão direito do mouse no projeto e selecionando **Editar Arquivo de Projeto**, como mostrado abaixo.

Figura 13 - Editar arquivo de Projeto.

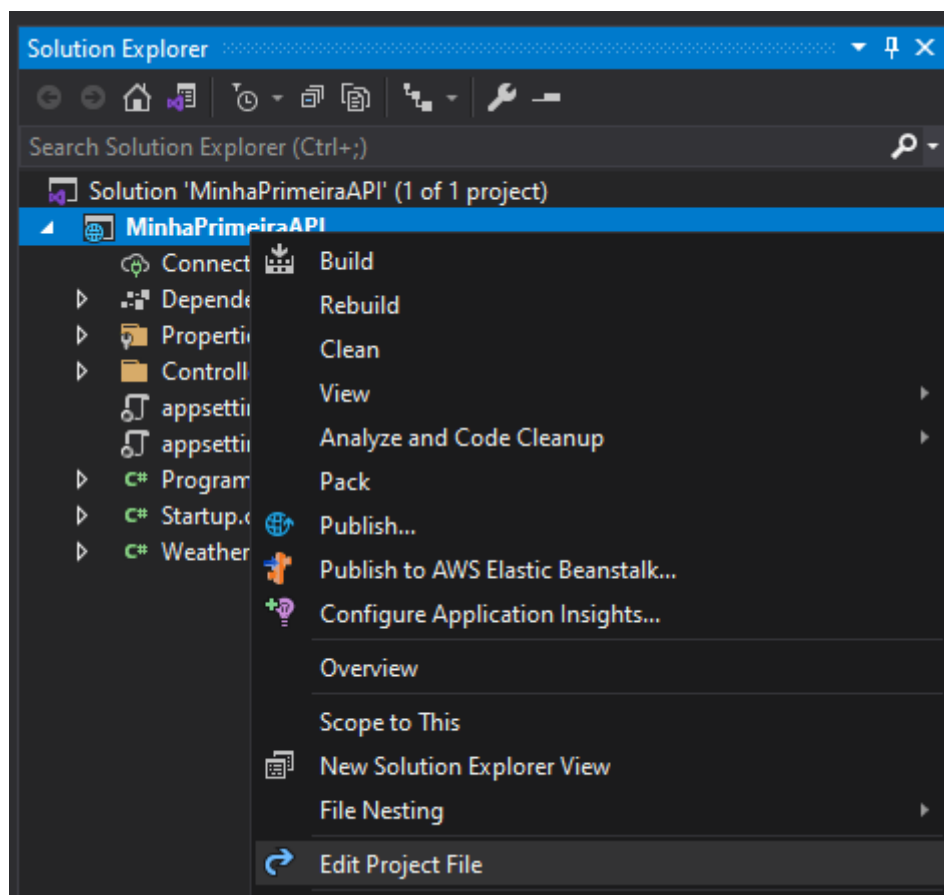


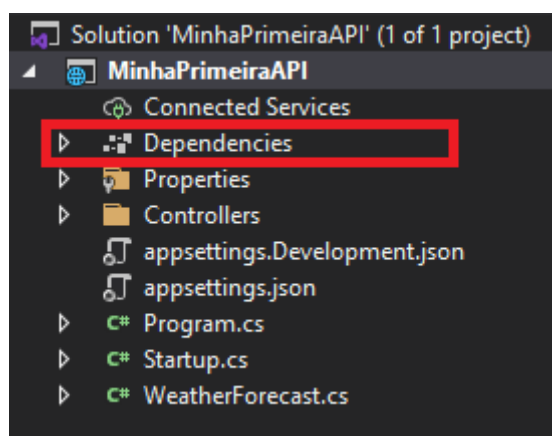
Figura 14 - O .csproj para o projeto acima se parece com o abaixo.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
</Project>
```

O arquivo csproj inclui configurações relacionadas a .NET Frameworks, pastas de projeto, referências de pacote NuGet etc.

As dependências do projeto estão agrupadas na seção abaixo.

Figura 15 - Seção de dependências.



A pasta **Propriedades** inclui o arquivo **launchSettings.json**, que inclui perfis do Visual Studio de configurações de depuração. O arquivo **launchSettings.json** contém algumas configurações que serão usadas pelo .NET Core Framework quando executarmos o aplicativo a partir do Visual Studio ou usando o .NET Core CLI. Outro ponto que você precisa ter em mente, o arquivo **launchSettings.json** só é usado na máquina de desenvolvimento local. Portanto, esse arquivo não é necessário ao publicar nosso aplicativo ASP.NET Core Web API no servidor de produção.

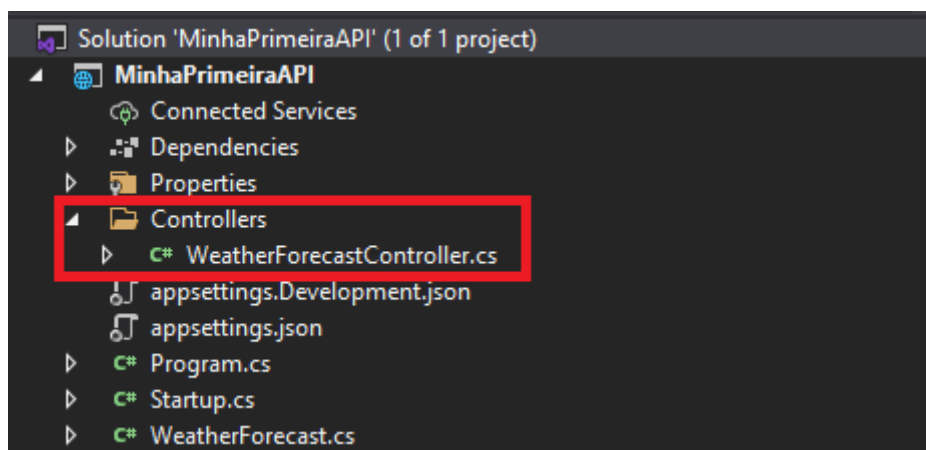
Figura 16 - Arquivo launchSettings.json.

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:9085",
      "sslPort": 44312
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "weatherforecast",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "MinhaPrimeiraAPI": {
      "commandName": "Project",
      "launchBrowser": true,
      "launchUrl": "weatherforecast",
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Controllers

A ASP.NET Core Web API é uma abordagem baseada em controllers. Essas devem residir na pasta Controllers.

Figura 17 - Pasta Controllers.



Se você abrir o arquivo **WeatherForecastController**, encontrará o seguinte código nele. A classe **Controller** é herdada da classe **ControllerBase** e decorada com **ApiController** e o atributo **Route**. Discutiremos todos esses elementos mais adiante.

Figura 18 - Controller de amostra.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace MinhaPrimeiraAPI.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class WeatherForecastController : ControllerBase
    {
        private static readonly string[] Summaries = new[]
        {
            "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
        };

        private readonly ILogger<WeatherForecastController> _logger;

        public WeatherForecastController(ILogger<WeatherForecastController> logger)
        {
            _logger = logger;
        }

        [HttpGet]
        public IEnumerable<WeatherForecast> Get()
        {
            var rng = new Random();
            return Enumerable.Range(1, 5).Select(index => new WeatherForecast
            {
                Date = DateTime.Now.AddDays(index),
                TemperatureC = rng.Next(-20, 55),
                Summary = Summaries[rng.Next(Summaries.Length)]
            })
            .ToArray();
        }
    }
}
```

Como você pode ver na imagem acima, temos um método **Get** decorado com o atributo **HttpGet**. E quando chamamos a seguinte URL usando qualquer cliente (Swagger, Postman e Browser), esse é o método que irá retornar os dados.

Ao executar o projeto no menu abaixo, obtemos o seguinte resultado na url localhost com uma porta definida pelo Visual Studio (<https://localhost:44312/weatherforecast>).

Figura 19 - Opção de execução do projeto.

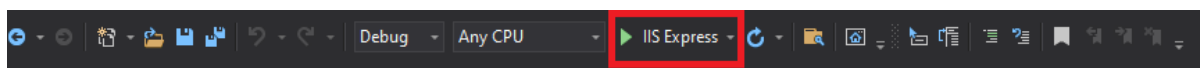
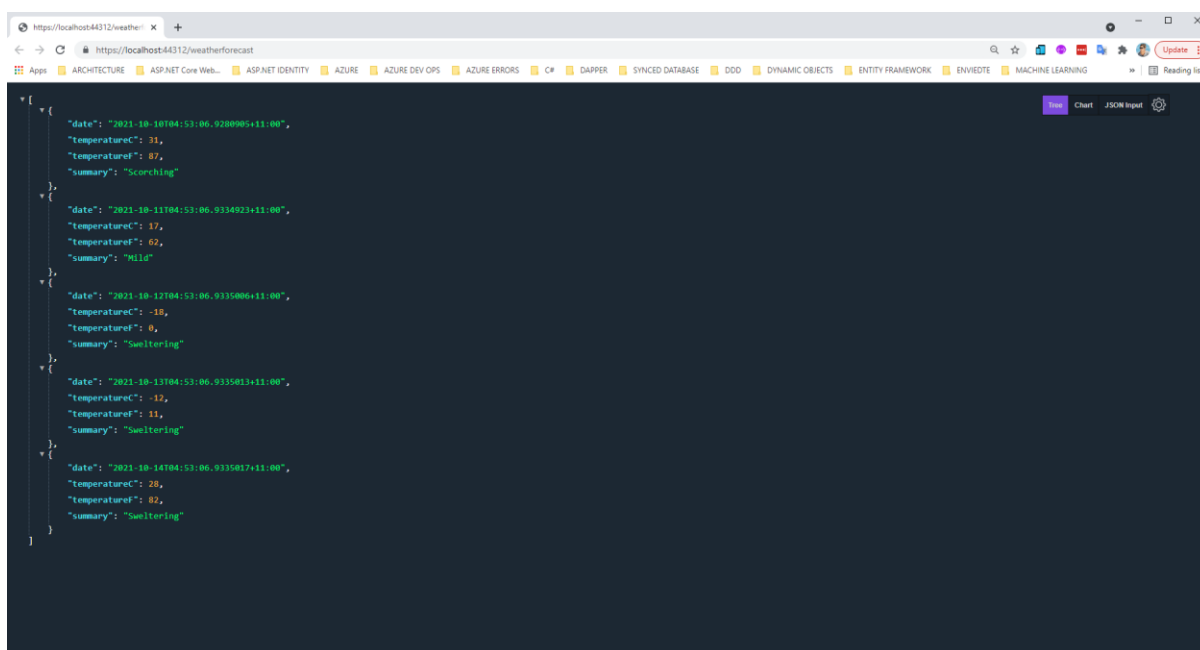


Figura 20 - Solução em execução no navegador.



Arquivo AppSettings.json

O próximo arquivo que vamos discutir é o arquivo **appsettings.json**. Ele é o arquivo de configuração na aplicação no ASP.NET Core Web usado para armazenar as configurações, como strings de conexão de banco de dados, qualquer variável global de escopo de aplicativo etc.

Se você abrir o arquivo `appsettings.json`, verá o seguinte código por padrão, criado pelo .NET Core Framework quando criamos o aplicativo ASP.NET Core Web API.

Figura 21 - `appsettings.json`.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Arquivo `AppSettings.Development.json`

Se quiser definir algumas configurações com base nos ambientes, você pode fazer essas configurações no arquivo **`appsettings.{Ambiente}.json`**. Você pode também criar vários ambientes, como desenvolvimento, homologação, produção etc.

Se você definir algumas configurações no arquivo `appsettings.Development.json`, elas não podem ser usadas em outros ambientes, só ser usadas no ambiente de desenvolvimento. Se você abrir o arquivo **`appsettings.Development.json`**, obterá o código a seguir.

Figura 22 - `appsettings.Development.json`.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

Classe Program.cs

O arquivo de classe Program.cs da aplicação ASP.NET Core Web API contém o código a seguir.

Figura 23 - Classe Program e o método estático Main.

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace MinhaPrimeiraAPI
{
    0 references
    public class Program
    {
        0 references
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        1 reference
        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Conforme mostrado no código de classe de **Program** acima, ele possui um método void **Main()** público estático. O método Main é o ponto de entrada da aplicação.

Cada aplicação no ASP.NET Core Web API começa como um aplicativo de console e o método **Main()** é o ponto de entrada. Portanto, quando executamos a

aplicação, primeiro procura o método **Main()**, daí o método configura o ASP.NET Core e o inicia. Nesse ponto torna-se um aplicativo ASP.NET Core Web API.

Classe Startup.cs

A classe Startup, como o nome sugere, é executada quando o aplicativo é iniciado. Você encontrará o seguinte código em sua classe de inicialização.

Figura 24 - Classe Startup.cs.

```
2 references
public class Startup
{
    0 references
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    1 reference
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    0 references
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseHttpsRedirection();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

Como você pode ver no código acima, a classe Startup inclui dois métodos públicos: **ConfigureServices** e **Configure**.

Método `ConfigureServices()`

O método **`ConfigureServices`** da classe `Startup` é o lugar onde podemos registrar nossas classes dependentes com o contêiner, que é o elemento que gerencia dependências. Depois de registrar as classes dependentes, elas podem ser usadas em qualquer lugar dentro do aplicativo. O método **`ConfigureServices`** inclui o parâmetro **`IServiceCollection`** para registrar serviços no gerenciador de dependências.

Se você notar, ele **`AddControllers`** ao contêiner que gerencia dependências, conforme mostrado na imagem acima.

Método `Configure()`

O método **`Configure`** da classe `Startup` é o local onde configuramos a pipeline de requisição do aplicativo usando a instância **`IApplicationBuilder`**, que é fornecida pelo contêiner que gerencia dependências. ASP.NET Core nesse ponto introduz os componentes de middleware para definir um pipeline de requisição, que será executado em cada solicitação. Se você observar o método `Configure`, ele registrou os componentes de middleware **`UseDeveloperExceptionPage`**, **`UseHttpsRedirection`**, **`UseRouting`**, **`UseAuthorization`** e **`UseEndpoints`** no pipeline de processamento de requisição, conforme mostrado na imagem acima.

Classe `WeatherForecast.cs`

Esta é a classe do modelo e você pode encontrar o seguinte código nela.

Figura 25 - Estrutura da classe WeatherForecast.cs.

```
public class WeatherForecast
{
    1 reference
    public DateTime Date { get; set; }

    2 references
    public int TemperatureC { get; set; }

    0 references
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

    1 reference
    public string Summary { get; set; }
}
```

Nota: O modelo pode ser colocado em qualquer lugar. Como por exemplo uma pasta chamada **Models** (Modelos). Diretamente no nível do projeto. Mesmo eles podem ser criados como projetos de biblioteca de classe separados.

Capítulo 7. Controllers (Controladores)

Quando o ASP .NET Core foi anunciado pela primeira vez (como ASP .NET 5 anos antes do lançamento do Core 1.0), um dos benefícios foi a unificação antecipada de controllers Web + API. Em vez de ter classes base separadas, agora você pode ter uma única classe pai ControllerBase para herdar, se você está construindo um MVC Web Controller ou um Web API Controller.

A partir do Core 2.0, as controllers do MVC e API derivam da classe Controller, que deriva de ControllerBase:

Figura 26 - Controller criada com a classe base Controller.

```
public class MinhaController : Controller
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool"
    };

    [HttpGet]
    0 references
    public IEnumerable<string> Get()
    {
        return Summaries;
    }
}
```

Pontos a serem lembrados ao trabalhar com a classe Controller na ASP.NET Core Web API:

- A classe Controller no ASP.NET Core Web API deve ter um sufixo “Controller”. Por exemplo, se você deseja adicionar uma controller com o nome Home, o nome deverá ser **HomeController**, da mesma forma, se você deseja adicionar um controlador para Compra, o nome deve ser **CompraController**.

- A classe **Controller** deve ser herdada da classe **ControllerBase**. Se você vem de um background do ASP.NET Core MVC, então, nesse caso, a classe Controller é herdada da classe Controller do framework.
- Precisamos decorar a classe do controlador com o atributo (**ApiController**).
- Também precisamos usar o atributo de roteamento (**Route**) para acessar o recurso usando URI.

Figura 27 - Atributos ApiController e Route.

```
[ApiController]
[Route("[controller]")]
0 references
public class MinhaController : Controller
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool"
    };

    [HttpGet]
    0 references
    public IEnumerable<string> Get()
    {
        return Summaries;
    }
}
```

Então, qual é o propósito do atributo **ApiController**? De acordo com a documentação do atributo, ele indica que todos os tipos derivados são usados para atender às respostas da API HTTP. A presença desse atributo pode ser usada para direcionar convenções, filtros e outros comportamentos com base na finalidade da controller.

Esse atributo, relativamente novo, não é necessário para construir uma controller da Web AP, mas tornará sua vida mais fácil. Na verdade, você pode criar uma controller base personalizada para que seja herdada por outras controllers, simplesmente usando o atributo [**ApiController**] na classe base.

Classes Controller x ControllerBase no ASP.NET Core

A classe **Controller** que usamos no ASP.NET Core MVC tem suporte para Views, ViewBag, ViewData, TempData etc. Mas aqui no ASP.NET Core Web API não exigimos tais conceitos. Portanto, pulamos a classe **Controller** como classe base e usamos a classe **ControllerBase**.

O ponto que você precisa lembrar é que a classe **ControllerBase** também serve como classe base para a classe **Controller**. Isso significa que, nos bastidores, a classe **Controller** é uma classe **ControllerBase**. Assim utilizando a classe **Controller** você também estará usando a classe **ControllerBase**.

Figura 28 - Classe ControllerBase.

```
[ApiController]
[Route("[controller]")]
0 references
public class MinhaController : ControllerBase
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool"
    };

    [HttpGet]
    0 references
    public IEnumerable<string> Get()
    {
        return Summaries;
    }
}
```

Classe ControllerBase

A classe **ControllerBase** no ASP.NET Core fornece muitos métodos e propriedades para lidar com solicitações e respostas HTTP. Por exemplo, se você deseja códigos de status **200** como retorno do seu método de ação (**action**), essa

classe fornece um método que abrevia a criação de um objeto que retorne o status 200. Da mesma forma, se você deseja retornar o código de status 201 do seu método de ação (action), a controller também fornece um método para esse fim.

Atributo [ApiController]

Indica que um tipo e todos os tipos derivados são usados para atender às respostas da API HTTP. Os controllers decorados com esse atributo são configurados com recursos e comportamento direcionados a melhorar a experiência do desenvolvedor para a construção de APIs. Quando decorados em um **assembly**, todos os controllers presente neste assembly serão tratados como controllers com o comportamento do atributo adicionado a elas. Podemos ver aqui alguns dos recursos oferecidos:

- Atributo de roteamento [Route].
- Tratativas de erro do cliente, por exemplo, do código de status HTTP 400.
- Inferência de solicitação multipart / form-data.
- Vinculando os dados de entrada com os parâmetros usando os conceitos de Model Binding.

Capítulo 8. Rotas

O roteamento é responsável por mapear as solicitações de entrada do navegador para ações específicas da controller.

No ASP.NET Core, o roteamento é gerenciado por um middleware de roteamento, que corresponde às URLs das requisições de entrada até as **Actions** ou outros pontos de extremidade.

As actions da Controller são roteadas de duas formas, sendo convencionalmente ou atributo. O roteamento convencional é semelhante à abordagem da tabela de rotas usada no ASP.NET MVC e Web API. Esteja você usando convencional, atributo ou ambos, precisará configurar seu aplicativo para usar o middleware de roteamento. Para usar o middleware, adicione o seguinte código ao método Startup.Configure:

Figura 29 - Aplicando middleware roteamento.

C#

```
app.UseRouting();
```

Roteamento Convencional

Com o roteamento convencional você configura uma ou mais convenções que serão usadas para corresponder URLs de entrada. No ASP.NET Core, esses pontos de extremidade podem ser ações da controller, como no ASP.NET MVC Web e API. Os endpoints também podem ser Razor Pages, Health Checks ou Signal Hubs. Todos esses recursos roteáveis são configurados de maneira semelhante usando endpoints:

Figura 30 - Roteamento Convencional.

```
C#

// in Startup.Configure()
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/healthz").RequireAuthorization();
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
    endpoints.MapRazorPages();
});
```

Roteamentos por Atributo

O roteamento por atributos no ASP.NET Core é a abordagem preferencial para configurar o roteamento em controllers. Se você estiver construindo APIs, o atributo **[ApiController]** deve ser aplicado aos seus controladores. Entre outras coisas, esse atributo requer o uso de roteamento de atributo para ações em tais classes de controlador.

Nessa abordagem o ASP.NET Core se comporta de maneira semelhante na ASP.NET MVC e Web API. Além de oferecer suporte ao atributo **[Route]**, as informações da rota também podem ser especificadas como parte do atributo do método HTTP:

Figura 31- Roteamento por Atributos I.

```
[HttpGet("api/produtos/{id}")]
0 references
public ActionResult<Produto> Detalhes(int id)
{
    Produto produto = Produtos.SingleOrDefault(p => p.Id == id);

    return Ok(produto);
}
```

Como nas versões anteriores, você pode especificar uma rota padrão com marcadores de posição e adicioná-la no nível da classe da controller ou mesmo em uma classe base. Você usa o mesmo atributo **[Route]** para todos esses casos. Por exemplo, uma classe controller base API pode ter a seguinte aparência:

Figura 32 - Roteamento por Atributos II.

```
[Route("api/{controller}/{action}/{id?:int}")]
0 references
public abstract class BaseApiController : ControllerBase
{
    ...
}
```

Usando esse atributo, as classes herdadas desse tipo rotearam URLs para ações com base no nome da controller, nome da action e um parâmetro opcional de id de número inteiro.

Métodos HTTP

A Web API também seleciona ações com base no método HTTP da solicitação (GET, POST etc.). Por padrão, a Web API procura uma correspondência sem distinção entre maiúsculas e minúsculas com o início do nome do método da controller. Por exemplo, um método de controlador denominado **PutClientes** corresponde a uma solicitação **HTTP PUT**.

Você pode substituir essa convenção, decorando o método com qualquer um dos seguintes atributos:

- **[HttpDelete]**
- **[HttpGet]**
- **[HttpHead]**
- **[HttpOptions]**

- [HttpPatch]
- [HttpPost]
- [HttpPut]

Prefixos de Rota

Frequentemente, todas as rotas em uma controller começam com o mesmo prefixo. Por exemplo:

Figura 33 - Prefixos em rota.

```
public class LivrosController : ControllerBase
{
    3 references
    public List<Livro> Livros { get; set; }

    [Route("api/livros")]
    0 references
    public IEnumerable<Livro> RecuperarLivros()
    {
        return Livros;
    }

    [Route("api/livros/{id:int}")]
    0 references
    public Livro RecuperarLivro(int id)
    {
        return Livros.SingleOrDefault(l => l.Id == id);
    }

    [Route("api/livros")]
    [HttpPost]
    0 references
    public HttpResponseMessage CriarLivro(Livro livro)
    {
        Livros.Add(livro);

        return new HttpResponseMessage(HttpStatusCode.OK);
    }
}
```

Você pode definir um prefixo comum para uma controller usando o atributo **[Route]** no nível da própria controller:

Figura 34 - Definindo prefixo ao controlador.

```
[Route("api/livros")]
0 references
public class LivrosController : ControllerBase
{
    3 references
    public List<Livro> Livros { get; set; }

    [Route("")]
    0 references
    public IEnumerable<Livro> RecuperarLivros()
    {
        return Livros;
    }

    [Route("{id:int}")]
    0 references
    public Livro RecuperarLivro(int id)
    {
        return Livros.SingleOrDefault(l => l.Id == id);
    }

    [Route("")]
    [HttpPost]
    0 references
    public HttpResponseMessage CriarLivro(Livro livro)
    {
        Livros.Add(livro);

        return new HttpResponseMessage(HttpStatusCode.OK);
    }
}
```

Use um til (~) no atributo do método para substituir o prefixo da rota:

Figura 35 - Substituição do prefixo.

```
[Route("api/livros")]
0 references
public class LivrosController : ControllerBase
{
    1 reference
    public List<Livro> Livros { get; set; }

    [Route("~/api/autores/{autorId:int}/livros")]
    0 references
    public Livro RecuperarAutor(int autorId)
    {
        return Livros.SingleOrDefault(l => l.AuthorId == autorId);
    }
}
```

O prefixo da rota pode incluir parâmetros:

Figura 36 - Inclusão de parâmetros.

```
[Route("clientes/{customerId}")]
0 references
public class ClientesController : ControllerBase
{
    1 reference
    public List<Pedido> Pedidos { get; set; }

    [Route("pedidos")]
    0 references
    public IEnumerable<Pedido> RecuperarPedidos(int clienteId)
    {
        return Pedidos.Where(c => c.ClienteId == clienteId);
    }
}
```

Restrições de Rota

As restrições de rota permitem que você restrinja como os parâmetros no modelo são combinados. A sintaxe geral é "{parâmetro: restrição}". Por exemplo:

Figura 37 - Inclusão de parâmetros.

```
[Route("clientes/{customerId}")]
0 references
public class ClientesController : ControllerBase
{
    2 references
    public List<Cliente> Clientes { get; set; }

    [Route("users/{id:int}")]
    0 references
    public Cliente RecuperarClientePorId(int id)
    {
        return Clientes.Single(c => c.Id == id);
    }

    [Route("users/name")]
    0 references
    public Cliente RecuperarClientePorNome(string nome)
    {
        return Clientes.Single(c => c.Nome == nome);
    }
}
```

Inferência de Parâmetros (Binding)

Tabela 2 - Tipos de Parâmetros nas Actions.

Atributo	Origem Ligação (Binding Source)
[FromBody]	Corpo da requisição (solicitação).
[FromForm]	Dados de um formulário no corpo da requisição.
[FromHeader]	Cabeçalho da requisição (solicitação).

[FromQuery]	Parâmetro de requisição na Query String.
[FromRoute]	Rota na requisição atual.
[FromServices]	Serviço injetado na requisição com um parâmetro na Action.

Capítulo 9. Action Results

ASP.NET Core oferece as seguintes opções para tipos de retorno das actions (método ao qual são configurados as rotas, verbos, entrada de parâmetros e a resposta (**response**)) nas controllers da Web API:

- **Tipo específico**
- **ActionResult**
- **ActionResult <T>**

Tipo Específico

A action mais simples retorna um tipo de dado primitivo ou complexo (por exemplo, string ou um tipo de objeto personalizado). Considere a seguinte ação que retorna uma coleção de objetos **Produtos** personalizados:

Figura 38 - Retornado dados primitivos.

```
[HttpGet]
0 references
public List<Produto> RecuperarProdutos()
{
    return Produtos;
}
```

Sem condições conhecidas (restrições) de proteção durante a execução da ação, retornar um tipo específico pode ser suficiente. No exemplo da imagem acima a action anterior não aceita parâmetros, então não será necessária a validação de entrada.

Quando vários tipos de retorno são possíveis, é comum misturar um tipo de retorno **ActionResult** com o tipo de retorno primitivo ou complexo. **ActionResult** ou **ActionResult<T>** são necessários para acomodar esse tipo de ação.

Retornar IEnumerable<T> ou IEnumerableAsync<T>

No ASP.NET Core 2.2 e anterior, retornar **IEnumerable<T>** de uma ação resulta em uma iteração síncrona no processo realizado pelo serializador. Isso resulta no bloqueio de chamadas e uma potencial privação do pool de threads. Para ilustrar, imagine que o **Entity Framework Core (EFC)** está sendo usado para interagir com o banco de dados e precise retornar elementos que atendam a uma condição específica. A imagem a seguir representa esse cenário:

Figura 39 - Retornando IEnumerable.

```
public IEnumerable<Produto> RecuperarProdutosEmProducao()
{
    return Produtos.Where(p => p.EstaEmPromocao);
}
```

Para evitar a enumeração síncrona e o bloqueio de esperas no banco de dados no ASP.NET Core 2.2 e anterior, utiliza-se o método **ToListAsync()**:

Figura 40 - Retornando IEnumerable com ToListAsync().

```
public async Task<IEnumerable<Produto>> RecuperarProdutosEmProducao()
{
    return await Produtos.Where(p => p.EstaEmPromocao).ToListAsync();
}
```

No ASP.NET Core 3.0 e posterior, retornando o tipo **IAsyncEnumerable <T>** em uma action, é permitido afirmar que:

- O processo não resulta mais em iteração síncrona.
- Torna-se tão eficiente quanto retornar **IEnumerable<T>**.

Considere declarar o tipo de retorno da assinatura de ação como **IAsyncEnumerable<T>** para garantir a iteração seja assíncrona. O modo de iteração é baseado no tipo concreto subjacente que está sendo retornado, o framework

armazena automaticamente qualquer tipo concreto que implemente `IAsyncEnumerable<T>`.

Figura 41 - Retornando `IAsyncEnumerable`.

```
[HttpGet("sincronizar-produtos-promocao")]
0 references
public async IEnumerable<Produto> GetOnSaleProductsAsync()
{
    var produtos = _repositorio.RecuperarProdutos();

    await foreach (var produto in produtos)
    {
        if (produto.EstaEmPromocao)
        {
            yield return produto;
        }
    }
}
```

O `IAsyncEnumerable<Produto>` equivale à próxima ação no qual o processo armazena em buffer o resultado da action antes de fornecê-lo ao serializador:

Figura 42 - Retornando `IEnumerable`.

```
[HttpGet("sincronizar-produtos-promocao")]
0 references
public IEnumerable<Produto> RecuperarProdutosEmPromocao()
{
    var produtos = _repositorio.RecuperarProdutos();

    foreach (var produto in produtos)
    {
        if (produto.EstaEmPromocao)
        {
            yield return produto;
        }
    }
}
```


Figura 43 - Eficiência em retornar IEnumerable.

```
public IEnumerable<Produto> RecuperarProdutosEmProducao()
{
    return Produtos.Where(p => p.EstaEmPromocao);
}
```

Tipo IActionResult

O tipo de retorno **IActionResult** é apropriado quando vários tipos de retorno **ActionResult** são possíveis em uma action. Os tipos **ActionResult** representam os diferentes códigos de status HTTP. Qualquer classe derivada de **ActionResult** se qualifica como um tipo de retorno válido. Alguns tipos de retorno comuns nessa categoria são **BadRequestResult (400)**, **NotFoundResult (404)** e **OkObjectResult (200)**. Como alternativa, os métodos de conveniência na classe ControllerBase podem ser usados para retornar os tipos **ActionResult** de uma ação. Por exemplo, `return BadRequest();` é uma forma abreviada de `return new BadRequestResult();`.

Como há vários tipos de retorno e caminhos para as actions, o uso do atributo **[ProducesResponseType]** pode ser necessário. Esse atributo produz detalhes da resposta mais descritivos e permitem expandir o que a action poderá retornar, além de prover detalhes para que os geradores de documentação como **Swagger** possam se enriquecer de informações.

Action Síncrona

Considere a seguinte action síncrona na qual existem dois tipos de retorno possíveis:

Figura 44 - Retorno síncrono.

```
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(Produto))]
[ProducesResponseType(StatusCodes.Status404NotFound)]
0 references
public IActionResult RecuperarPorId(int id)
{
    if (!_repositorio.TryGetProduct(id, out var produto))
    {
        return NotFound();
    }

    return Ok(produto);
}
```

Na action anterior:

- Um código de status 404 é retornado quando o produto representado por id não existe no banco de dados subjacente. O método de conveniência **NotFound** é chamado como uma abreviação para return **new NotFoundResult()**;
- Um código de status 200 é retornado com o objeto **Produto** quando o produto existe. O método de conveniência Ok é invocado como uma abreviação para retornar novo **OkObjectResult (produto)**;

Action Assíncrona

Considere a seguinte action assíncrona na qual existem dois tipos de retorno possíveis:

Figura 45 - Retorno assíncrono.

```
[HttpPost]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
1 reference
public async Task<IActionResult> CriarProdutoAssincro(Produto produto)
{
    if (produto.Descricao.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    await _repositorio.AddProductAsync(produto);

    return CreatedAtAction(nameof(CriarProdutoAssincro), new { id = produto.Id }, produto);
}
```

Na ação anterior:

- Um código de status 400 é retornado quando a descrição do produto contém "XYZ Widget". O método de conveniência **BadRequest** é invocado como uma abreviação para `return new BadRequestResult();`.
- Um código de status 201 é gerado pelo método de conveniência **CreatedAtAction** quando um produto é criado. Uma alternativa para chamar **CreatedAtAction** é retornar novo **CreatedAtActionResult** (`nameof(CriarProdutoAssincro), new {id = produto.Id}, produto`). Nesse caminho de código, o objeto **Produto** é fornecido no corpo da resposta. No cabeçalho de resposta existe um atributo chamado **location** (localização) contendo uma URL apontando para o produto recém-criado.

Capítulo 10. Convenções

ASP.NET Core 3.0 e posterior inclui uma maneira de extrair documentação e aplicá-la aos vários elementos, como: actions, controllers ou todos controllers em um assembly. As convenções da Web Api são um substituto ao decorar ações individuais com **[ProducesResponseType]**.

Uma convenção permite que você:

- Defina os tipos de retorno e códigos de status mais comuns retornados de um tipo específico de ação.
- Identifique as ações que se desviam do padrão definido.

ASP.NET Core 3.0 e posterior inclui um conjunto de convenções padrão em na biblioteca **Microsoft.AspNetCore.Mvc.DefaultApiConventions**. As convenções abaixo são demonstradas com **LivrosController.cs**:

Figura 46 - Convenções.

```
[Route("api/[Controller]")]
[ApiController]
0 references
public class LivrosController : ControllerBase
{
    #region Metodos

    //METODO GET api/livros
    [HttpGet]
    0 references
    public async Task<IActionResult> Get()
    {
        return await Task.FromResult(Ok(new string[] { "", "" }));
    }

    //METODO GET api/livros/10
    [HttpGet("{id}")]
    0 references
    public async Task<IActionResult> Get(int id)
    {
        return await Task.FromResult(Ok("value"));
    }

    // METODO POST api/livros
    [HttpPost]
    0 references
    public void Post([FromBody] string value)
    {
    }

    // METODO PUT api/livros/10
    [HttpPut]
    0 references
    public void Put(int id, [FromBody] string value)
    {
    }

    // METODO DELETE api/livros/10
    [HttpDelete]
    0 references
    public void Delete(int id)
    {
    }

    #endregion
}
```

As actions em **LivrosController.cs** seguem os padrões funcionando com as convenções padrão. Se as convenções padrão não atenderem às suas necessidades, consulte no site da Microsoft como criar convenções de Web Api.

Em tempo de execução, **Microsoft.AspNetCore.Mvc.ApiExplorer** entende as convenções e o **ApiExplorer** que é uma abstração para se comunicar com geradores de documentos **OpenAPI** (também conhecido como **Swagger**).

Aplicando Convenções Web Api

As convenções não são compostas como o atributo **[ProducesResponseType]**; cada action pode ser associada a exatamente uma convenção. Convenções mais específicas têm precedência sobre convenções menos específicas, isto é, primeiro as actions, seguindo pelas controllers e por último os assemblies. A seleção é não determinística quando duas ou mais convenções da mesma prioridade se aplicam a uma action. As seguintes opções existem para aplicar uma convenção a uma action, da mais específica à menos específica:

1. **Microsoft.AspNetCore.Mvc.ApiConventionMethodAttribute** - Aplica-se a ações individuais e especifica o tipo de convenção e o método de convenção que se aplica.

No exemplo a seguir, o método de convenção **Microsoft.AspNetCore.Mvc.DefaultApiConventions.Put** do tipo de convenção padrão é aplicado à action Atualizar:

Figura 47 – Convenções.

```
// METODO DELETE api/livros/10
[HttpPut("{id}")]
[ApiConventionMethod(typeof(DefaultApiConventions),
                      nameof(DefaultApiConventions.Put))]
0 references
public IActionResult Update(int id, Livro livro)
{
    if (id == 0)
    {
        return BadRequest();
    }

    if(id != livro.Id)
    {
        return NotFound();
    }

    var livroParaAtualizar = repositorio.Get(id);

    if (livroParaAtualizar == null)
    {
        return NotFound();
    }

    repositorio.Update(livroParaAtualizar);

    return NoContent();
}
```

O método da convenção **Microsoft.AspNetCore.Mvc.DefaultApiConventions.Put** aplica os seguintes atributos à action, ou seja, uma convenção padrão define diferentes tipos de retorno:

[ProducesDefaultResponseType]

[ProducesResponseType(StatusCodes.Status204NoContent)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

2. **Microsoft.AspNetCore.Mvc.ApiConventionTypeAttribute** é aplicado a uma controller, ou seja, aplica o tipo de convenção especificado a todas as action desta controller. No exemplo a seguir, o conjunto padrão de convenções é aplicado a todas as action em **ContatosConvencaoController**:

Figura 48 - Aplicando o atributo ApiConventionType.

```
[ApiController]
[ApiConventionType(typeof(DefaultApiConventions))]
[Route("api/[controller]")]
0 references
public class ContatosConvencaoController : ControllerBase
{
    ...
}
```

3. **Microsoft.AspNetCore.Mvc.ApiConventionTypeAttribute**: este, por sua vez, é aplicado a um assembly, ou seja, aplica o tipo de convenção especificado a todas as controllers no assembly atual. Como recomendação, aplique atributos nesse escopo no arquivo Startup.cs.

No exemplo a seguir, o conjunto padrão de convenções é aplicado a todos os controladores na montagem:

Figura 49 - Aplicando o atributo ApiConventionType no assembly.

```
using Microsoft.AspNetCore.Mvc;

[assembly: ApiConventionType(typeof(DefaultApiConventions))]
namespace ApiConventions
{
    0 references
    public class Startup
    {
        ...
    }
}
```


Atributos para Tipos de Response

Para definir os tipos de response, os métodos podem ser anotados com os atributos `[ProducesResponseType]` ou `[ProducesDefaultResponseType]`. Por exemplo:

Figura 50 - Diferentes tipos de Response.

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

0 references
public static class MyAppConventions
{
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status404NotFound)]
    0 references
    public static void Find(int id)
    {
    }
}
```

Desta forma define-se que o método **Find** possui 2 tipos de retornos, mas se atributos mais específicos estiverem ausentes. Se a aplicação dessa convenção for em nível de assembly é imposto que:

- O método de convenção se aplica a qualquer action chamada **Find**.
- Um parâmetro denominado id está presente na **action** Find.

Capítulo 11. CRUD (Create, Read, Update e Delete)

A definição de CRUD vem de (**Create**, Read, Update e Delete) que são verbos que representam ações para operações relacionadas a banco de dados. No exemplo a seguir vamos unir essa ideia aos conceitos básicos da criação de uma Web API com o ASP.NET Core, fazendo a ligação deste conceito com os verbos Http.

Neste capítulo, você aprenderá como:

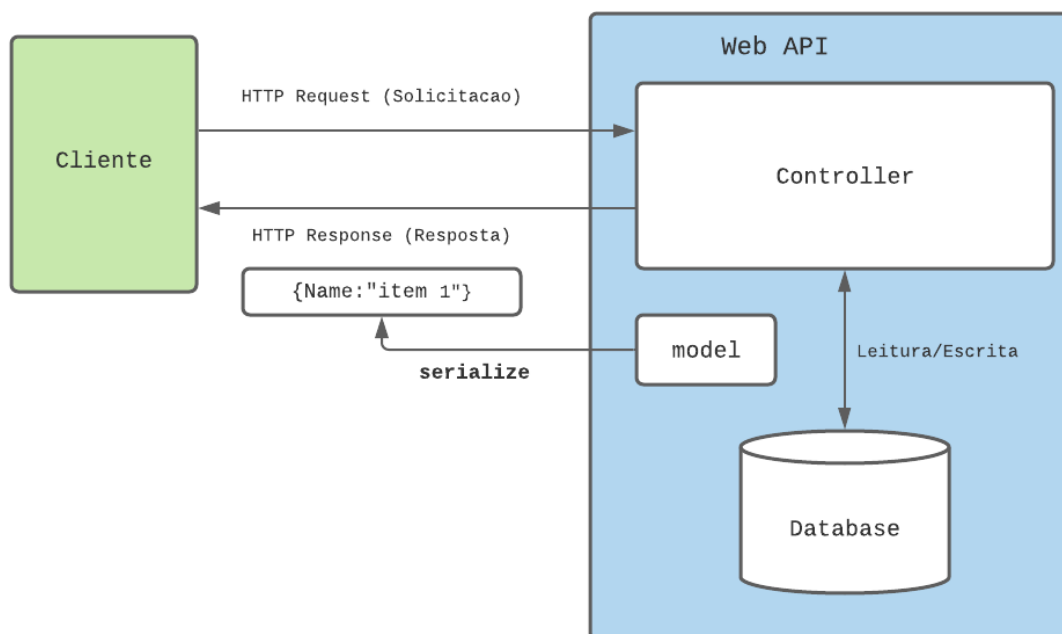
- Criar um projeto um Web Api (siga o mesmo procedimento adotado para o tópico **Criando projeto com Visual Studio 2019**). A aplicação será uma representação de cadastro de livros, sendo a solução nomeada de **BookStore**.
- Criar uma classe que vai representar um **Livro**, essa classe será um simples modelo (classe sem métodos somente com propriedades representando as informações principais) e uma classe que vai representar o contexto do banco de dados (classe que representa o banco de dados).
- Adicionar uma controller com métodos CRUD (GET, POST, DELETE, PUT).
- Definir as rotas (caminhos de URL) e como valores serão retornados.
- Chamar a aplicação Web API com o Postman (Postman é um aplicativo usado para testes de API): <https://www.postman.com/downloads/>.

Tabela 3 - Verbos HTTP para a aplicação exemplo.

API	Descrição	Corpo da Solicitação	Corpo da Resposta
GET /api/livros	Obter todos os livros	Nenhum	Coleção de itens de livros.
GET /api/livros/{id}	Obter um livro por ID	Nenhum	Livro
POST	Adiciona um novo livro	Um novo livro	Livro
PUT /api/livros/{id}	Atualizar um item existente	Item de tarefas pendentes	Nenhum
DELETE /api/livros/{id}	Exclui um livro	Nenhum	Nenhum

O diagrama a seguir mostra o design da aplicação:

Figura 51 - Diagrama representando a aplicação.



Adicionar um contexto de Banco de Dados

Um contexto de banco de dados é uma representação do banco de dados propriamente dito, no caso do Entity Framework, esse contexto é uma classe que coordena o modelo de dados baseado em objetos e que são mapeados para as tabelas dos bancos de dados. O Entity Framework está fora do escopo deste curso e exige um material totalmente dedicado. Mas é importante saber que para criarmos uma classe que representa um contexto, esta deve herdar a classe **Microsoft.EntityFrameworkCore.DbContext**. Este contexto pode ser mapeado para diferentes tipos de provedores de banco de dados, mas para critério de estudo utilizaremos um provedor em memória (**InMemory**) provido pelo **Entity Framework Core** através de um pacote chamado **Microsoft.EntityFrameworkCore.InMemory** para criar um "banco de dados" funcional na memória.

Classe do contexto DbContext e Geração de dados

Criaremos como passo inicial uma classe que herde de DbContext (classe essa que nos permitirá trabalhar com o nosso banco de dados em memória) e um gerador de dados que inicializará com alguns dados de amostra. O nome para essa classe será **LivrosDbContext**.

Figura 52 - Classe DbContext representando o contexto.

```
using BookStore.Models;
using Microsoft.EntityFrameworkCore;

namespace BookStore.Data.Context
{
    2 references
    public class LivrosDbContext : DbContext
    {
        #region Construtor

        0 references
        public LivrosDbContext(DbContextOptions<LivrosDbContext> options)
            : base(options) { }

        #endregion

        #region Propriedades

        0 references
        public DbSet<Livro> Livros { get; set; }

        #endregion
    }
}
```

- No Gerenciador de soluções adicionaremos uma nova pasta chamada **Models** pela ação de clicar com o botão direito do mouse no projeto e Selecionando a opção **Adicionar > Nova Pasta**.
- Selecione a pasta **Models** e adicione uma nova classe, para isto clique com o botão direito e selecione a opção **Adicionar > Classe**. Dê à classe o nome **Livro** e selecione **Adicionar**.

- Substitua o código do modelo pelo seguinte:

Figura 53 - Livro model (modelo).

```
namespace BookStore.Models
{
    0 references
    public class Livro
    {
        0 references
        public string Id { get; set; }
        0 references
        public string Titulo { get; set; }
        0 references
        public string Autor { get; set; }
        0 references
        public int Edicao { get; set; }
        0 references
        public string Editora { get; set; }
        0 references
        public string ISBN { get; set; }
    }
}
```

Como o banco de dados que estamos trabalhando é representado em memória, é importante que alguns dados de amostra sejam inseridos no momento da inicialização.

A forma mais simples de criarmos esses dados será utilizar o contexto e tabela fictícia representada pelo tipo **DbSet<Livro>** e adicionar elementos de forma natural, lembrando que **DbSet** é uma coleção de dados que representa a tabela **Livros**. Para fazer isso, vamos criar uma nova classe (que chamei de **GeradorDados**) com um método estático **Inicializar** que recebe como parâmetro uma interface **IServiceProvider** que será responsável por resolver um serviço para obter uma instância do contexto.

Figura 54 - Gerador de dados.

```
namespace BookStore.Data.Gerador
{
    0 references
    public class GeradorDados
    {
        0 references
        public static void Inicializar(IServiceProvider serviceProvider)
        {
            using (var contexto = new LivrosDbContext(serviceProvider.GetRequiredService<DbContextOptions<LivrosDbContext>>()))
            {
                // Procurando por livros.
                if (contexto.Livros.Any())
                {
                    return; // Dados já criados.
                }

                contexto.Livros.AddRange(
                    new Livro
                    {
                        Id = 1,
                        Autor = "J.J.Tolkien Books",
                        Edicao = 12,
                        Editora = "J.J.Tolkien Books",
                        ISBN = Guid.NewGuid().ToString(),
                        Titulo = "Senhor dos Aneis"
                    });

                contexto.SaveChanges();
            }
        }
    }
}
```

Agora que temos um contexto e um método inicializador com dados, vamos adicioná-lo ao arquivo `Startup.cs` e `Program.cs` onde fará mais sentido para o processo de inicialização. Esses arquivos são parte da aplicação ASP.NET Core para que nossos dados sejam preenchidos.

Classes `Startup.cs` e `Program.cs`

Na classe de inicialização (**`Startup.cs`**), adicionaremos **`LivrosDbContext`** na configuração dos serviços, sendo:

Figura 55 - Métodos `ConfigureServices` da classe `Startup.cs`.

```
0 references
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddDbContext<LivrosDbContext>(options => options.UseInMemoryDatabase(databaseName: "LivrosStore"));
}
```

Após o procedimento anterior, precisamos fazer algumas modificações no arquivo Program.cs para carregar nossos dados na memória na inicialização da aplicação. Nós precisamos:

- Obter a instância de **IHost** responsável pela execução da aplicação.
- Utilizar o provedor de serviços do host através de um escopo.
- Solicitar uma instância do contexto **LivrosDbContext**.
- Chamar **GeradorDados** para inicializar o banco com os dados de amostra.

Figura 56 - Inicializando Gerador de dados.

```

0 references
public class Program
{
    0 references
    public static void Main(string[] args)
    {
        //1 - RECUPERE O IHOST NO QUAL SUPORTARA A APLICACAO.
        var host = CreateHostBuilder(args).Build();

        //2 - ENCONTRE A CAMADA DE SERVICO DENTRO DO ESCOPO
        using (var escopo = host.Services.CreateScope())
        {
            //3 - RECUPERE A INSTANCIA DE LIVROSDBCONTEXT EM NOSSA CAMADA DE SERVICO
            var servicos = escopo.ServiceProvider;

            var contexto = servicos.GetRequiredService<LivrosDbContext>();

            //4 - CHAME O GERADOR DE DADOS PARA CRIAR OS DADOS EXEMPLO
            GeradorDados.Inicializar(servicos);
        }

        //CONTINUE EXECUTAR A APLICACAO
        host.Run();
    }

    1 reference
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}

```

Dessa forma o contexto de banco de dados fará parte da aplicação.

Controller LivroController.cs

Adicione uma nova classe chamada *LivroController.cs*, marque a classe com o atributo **[ApiController]**. Esse é o atributo que vai fazer com o que a controller responda às solicitações da Web Api e herde a classe **ControllerBase**.

Através do construtor vamos injetar uma instância do contexto (**LivrosDbContext**) na controller. Essa será a instância utilizada por todas as ações da controller.

Método POST

Figura 57 - Método Post.

```
[HttpPost]
0 references
public async Task<ActionResult<Livro>> Post([FromBody]Livro livro)
{
    contexto.Livros.Add(livro);

    await contexto.SaveChangesAsync();

    return CreatedAtAction(nameof(Get), new { id = livro.Id }, livro);
}
```

O código acima representa um método HTTP POST, conforme indicado pelo atributo **[HttpPost]**. O método obtém o valor do item (livro) no corpo da solicitação HTTP.

O método CreatedAtAction:

- Retorna uma ação bem-sucedida com o código de status **HTTP 201**. Convencionalmente, o 201 é a resposta padrão para um método HTTP POST quando se cria um novo recurso no servidor/banco de dados.
- Cria no cabeçalho de resposta a localização especificada pelo atributo **Location** com a URI do livro recém-criado.
- O primeiro parâmetro faz referência à action **GET** para criar atributo de **Location** no cabeçalho com o valor da URI. A palavra-chave **nameof** do C# é utilizada preferencialmente nas situações em que precisamos inferir uma string. Ao invés de digitar o nome do membro, essa instrução recebe o nome do membro definido e dessa forma evita erros de digitação, pois digitação manual é factível de erro e devido a linguagem ser case sensitive pode falhar ao especificar o membro referido.

Métodos GET

Dois pontos de extremidade GET são implementados:

- Get /api/livros
- Get /api/livros/{id}

Figura 58 - Métodos Get.

```
[HttpGet("{id}")]
1 reference
public async Task<ActionResult<Livro>> Get([FromRoute]int id)
{
    var livro = await contexto.Livros.FindAsync(id);

    if(livro == null)
    {
        return NotFound();
    }

    return livro;
}

[HttpGet]
1 reference
public async Task<ActionResult<List<Livro>>> Get()
{
    return await contexto.Livros.ToListAsync();
}
```

O tipo de retorno dos **GET** métodos é **ActionResult<T>**. O ASP.NET Core no processo de resposta automaticamente serializa o objeto e os grava no corpo da mensagem de resposta no formato JSON. Quando retornado com sucesso, o tipo de retorno é caracterizado por **200 OK**, supondo que não haja falhas/exceções. Quando situações de falha/exceções acontecem e não há tratativa, são convertidas em erros **5xx**.

Os retornos oriundos do tipo **ActionResult** podem em sua escala representar uma variedade de códigos de status HTTP. Por exemplo, o método **GET** parametrizado pode retornar dois valores de status diferentes:

- Se o identificador referenciado (Id) não corresponde a nenhum elemento existente, ele retorna um código de erro de status **404 NotFound** representando que o elemento não foi encontrado.
- Se o elemento for encontrado, o método retornará com sucesso com o código **200** no corpo de resposta JSON, além de retornar o elemento solicitado.

Método PUT

Figura 59 - Método Put.

```
[HttpPut("{id}")]
0 references
public async Task<ActionResult<Livro>> Put([FromRoute] int id, Livro livro)
{
    if (id != livro.Id)
    {
        return BadRequest();
    }

    contexto.Entry(livro).State = EntityState.Modified;

    try
    {
        await contexto.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!VerificarSeLivroExiste(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

PUT é semelhante ao **POST**, exceto pelo uso do atributo **HttpPut**. A resposta para esse método é 204 (Sem Conteúdo). Na especificação de HTTP, uma solicitação utilizando o verbo PUT requer que o cliente envie o modelo/entidade atualizado, não somente apenas as informações alteradas, pois para atualizações parciais utilizamos o atributo **HttpPatch**. Se você ver um erro ao chamar **PUT**, considere chamar o método **GET** para certificar de que o elemento existe no banco de dados.

Métodos DELETE

Examine o método **Delete**, o atributo **HttpDelete** é aplicado e o identificador do elemento será inserido na rota, no primeiro instante é verificado a existência do elemento no banco de dados e validado em casos de não ser encontrado, seguindo pela exclusão propriamente dita com o retorno **204** (Sem Conteúdo) baseado na convenção.

Figura 60 - Método Delete.

```
[HttpDelete("{id}")]
0 references
public async Task<IActionResult> Delete(int id)
{
    var livro = await contexto.Livros.FindAsync(id);

    if (livro == null)
    {
        return NotFound();
    }

    contexto.Livros.Remove(livro);

    await contexto.SaveChangesAsync();

    return NoContent();
}
```

Para excluir um item, utilize o **Postman**, selecione o método como **Delete** e envie a requisição para acompanhar o processo.

Roteamento

O atributo **[HttpGet]** decora um método que responde a uma solicitação HTTP do tipo Get. Na construção das Urls para cada método devemos considerar o seguinte:

- No atributo Route especificamos **api**, não é obrigatório, mas é importante por representar o propósito, em seguida temos [controller], que será substituído pelo nome **Livros** do **LivrosController**, sendo o prefixo Livros, mas se atribuirmos o valor diretamente, esta será a rota.

Figura 61 – Controlador.

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class LivrosController : ControllerBase
{
    #region Campos

    private LivrosDbContext contexto;

    #endregion
}
```

O sufixo "Controller" é uma convenção pela Microsoft que é mantida por todas as controllers. O roteamento do ASP.NET Core faz distinção entre maiúsculas e minúsculas.

Podemos especificar outras rotas, como na action com o atributo [**HttpGet**] (por exemplo, [**HttpGet** ("estados")]), devemos acrescentar ao final da url/caminho principal.

Quando o método **GET**, que exige um identificador por exemplo **id**, essa marcação "{id}" representa um parâmetro que vai identificar o item a ser procurado. Quando há uma requisição para esse método, o valor "{id}" é referenciado como parte da URL.

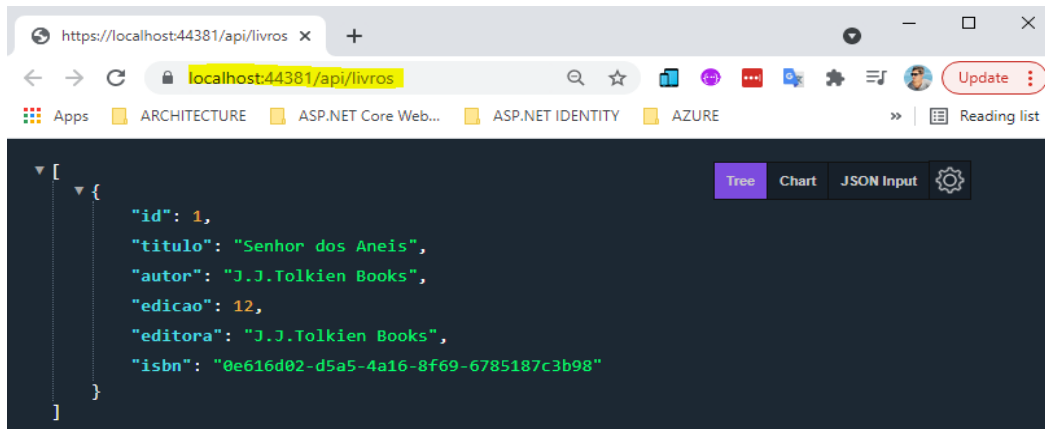
Instale o Postman

Este tutorial usa o Postman para testar a API exemplo.

- Instale através do link (<https://www.postman.com/downloads/>).

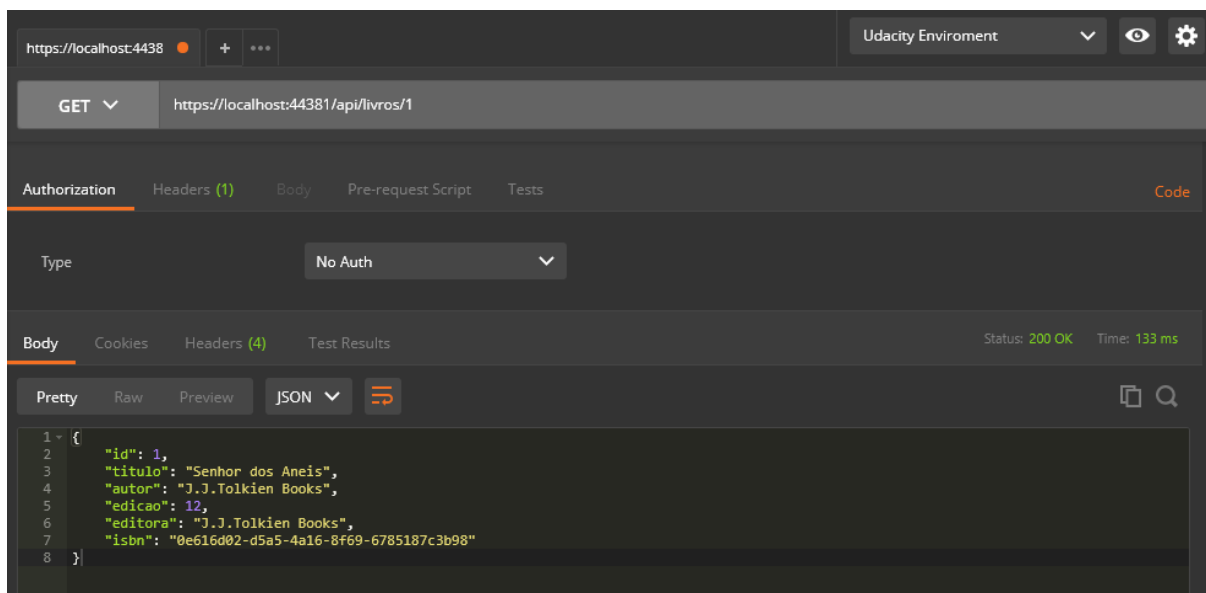
- Execute a aplicação.

Figura 62 - Aplicação em Execução.



- Inicie o Postman.
- Execute os testes nos métodos correspondentes.

Figura 63 - Postman com o método Get.



Capítulo 12. Segurança

A segurança de aplicações sempre foi, ou deveria ser, uma das principais preocupações dos tempos de desenvolvimento, e nos últimos anos a autenticação de APIs vem sendo bastante discutida nas comunidades técnicas em todo o mundo.

Existem diversas maneiras de resolver a autenticação em APIs, uma delas é fazendo uso de tokens de acesso, sendo o JWT (Json Web Token) um dos padrões de tokens de acesso mais famosos e seguros da atualidade.

Autenticação

A autenticação é o processo de confirmação da identidade de um usuário. Geralmente, essa é a etapa inicial do processo de segurança. Para confirmar a identidade do usuário, ele deve apresentar evidências físicas ou não físicas (informações) para a plataforma de autenticação. Os usuários podem ser agrupados da seguinte forma:

- **O que possuem:** a posse de um objeto físico, como uma chave, cartão-chave, chaveiro ou cartão magnético.
- **O que sabem:** informações confidenciais que inclui uma senha, código de acesso, número de identificação pessoal (PIN), data de nascimento, número do Seguro Social ou outras informações de identificação pessoal (PII).
- **O que utilizam:** biometria ou o uso de um dedo indicador, polegar, mão, voz, retina, rosto ou outro identificador físico exclusivo para obter acesso a um recurso. O atributo físico deve corresponder ao utilizado no momento da inscrição do usuário no sistema.

As senhas são geralmente o fator de autenticação mais comum e mais antigo. Se a senha corresponder exatamente à senha criada pelo usuário ou pelo sistema, o sistema assume a validade e concede acesso.

Outros processos de autenticação baseados em informações também estão ganhando popularidade. Um deles é o **PIN** único ou a senha temporária gerada pelo sistema. Ele permite que um usuário acesse uma sessão única ou temporária que expira após um determinado período de tempo **OTP** (one time password). Os usuários de banco móvel normalmente encontram esse procedimento para transações de transferência de dinheiro, especificamente quando um novo destinatário, a princípio não reconhecido pelo sistema, é adicionado.

Outra forma de confirmar a identidade do usuário é por meio de um aplicativo de autenticação, geralmente no dispositivo móvel do usuário, que gera códigos de segurança temporários que concedem acesso a outro site ou serviço.

A autenticação de dois fatores (2FA) e a autenticação múltiplos fatores (MFA) também estão cada vez mais sendo empregadas para aumentar a segurança além do nível fornecido apenas pelas senhas. Esses processos requerem a verificação bem-sucedida de uma ou mais modalidades antes de conceder acesso a um sistema. Por exemplo, o MFA pode pedir a um usuário que forneça uma senha e o PIN temporário enviado ao dispositivo móvel do usuário.

Autorização

Autorização é o processo de dar a um usuário permissão para acessar um local físico ou recurso baseado em informações (por exemplo, um documento, banco de dados, aplicativo ou site).

Autorização infelizmente é usada como sinônimo de autenticação, mas isso é um erro. A autenticação ocorre primeiro, seguida pela autorização. Os usuários precisam provar suas identidades antes que um sistema possa conceder-lhes permissão para entrar.

No entanto, permissão é um termo amplo. Um usuário pode passar pelos procedimentos de autenticação e receber acesso a um sistema, mas isso não significa que ele pode acessar todos os componentes de uma aplicação ou serviço on-line,

porque permissões específicas podem ser definidas pela organização que permitiu o acesso.

Permissões são o que um usuário pode ver ou fazer em um site ou dentro de uma aplicação. Sem essas permissões específicas, todos os usuários teriam acesso às mesmas informações ou recursos.

Dessa forma, as permissões e restrições e sua administração adequada são essenciais para a segurança de uma organização por vários motivos. Isso ocorre porque eles:

Impedir que um usuário acesse a conta de outro cliente:

Essa é talvez a razão mais importante pela qual as permissões são necessárias. Por exemplo, um cliente pode fazer login em sua conta bancária por meio do site do banco ou de um aplicativo móvel. Embora o banco tenha permitido que o usuário entre no sistema, o banco também precisa autorizar as permissões do usuário. Caso contrário, o usuário teria acesso não apenas à sua própria conta, mas também a todas as outras contas no sistema. As permissões garantem que os usuários possam acessar apenas as informações de que precisam.

Impedir que contas gratuitas recebam os benefícios de recursos premium:

Os níveis de permissão restringem os usuários gratuitos de um site Software-as-a-Service (SaaS), como um jornal com conteúdo fechado ou uma plataforma de colaboração on-line, a fim de obter acesso a recursos premium. As permissões precisam ser implementadas para que os usuários tenham acesso apenas aos recursos pelos quais pagaram. Sem restrições, haveria perda de receita para a organização.

Garantir cruzamento zero entre contas de clientes externos e contas internas:

As permissões também separam os usuários internos dos externos. Embora funcionários e clientes possam ter permissão para usar o site da empresa, os funcionários devem ter acesso a dados e sistemas que os clientes não deveriam ter.

Na mesma linha, alguns funcionários não devem ter acesso a informações importantes do cliente. Como tal, a organização deve criar diferentes níveis de autorização para cada funcionário.

Definir os níveis de permissão corretos é tão importante quanto selecionar a combinação certa de fatores de autenticação. Na verdade, a autorização adequada pode reduzir os efeitos negativos de uma violação de dados. Por exemplo, se um hacker obtiver acesso à conta de um funcionário e esse funcionário não estiver autorizado a acessar as informações bancárias ou de cartão de crédito do cliente, os efeitos nocivos da violação podem ser reduzidos.

Além disso, as autorizações tornam os funcionários mais produtivos. Se eles tiverem o nível correto de acesso aos arquivos e programas de que precisam para realizar seu trabalho, não precisam pedir acesso constantemente a seus gerentes ou TI. Eles também não serão distraídos ou sobrecarregados por arquivos e programas de que não precisam.

Autenticação vs Autorização

Para reiterar, autenticação e autorização são etapas separadas no processo de provisionamento de acesso do usuário. Podemos fazer uma analogia para demonstrar as diferenças.

Considere um agente cuidador de animais (cães) que precisa entrar na casa de uma família que está viajando de férias. E antes de viajar eles contratam uma empresa que cuida de cães para que durante o período de ausência o agente fornecido pela empresa possa ir até a residência e cuidar do animal. Então para acessar a residência o agente precisa de:

1. **Autenticação**, como uma chave, cartão-chave ou código de segurança para entrar na casa. Então se o agente tiver a peça correta de hardware para destravar a porta, ela poderá entrar na casa.

2. **Autorização**, como as permissões e restrições são estabelecidas pela família. O agente foi autorizado a acessar a sala de estar (onde fica a coleira do animal) e a cozinha (onde fica guardada a comida do animal). Uma vez acessadas as áreas em que foram concedidas as permissões, o agente não pode acessar outros lugares.

Neste exemplo, autenticação e autorização trabalham juntas. O agente tem o direito de entrar na casa (autenticação) e, uma vez lá, tem acesso apenas a determinadas áreas (autorização).

Tabela 4 - Comparação entre Autenticação e Autorização.

	Autenticação	Autorização
O que faz?	Verifica a identidade com credenciais.	Concede (ou nega permissão de acesso).
Como funciona?	Principalmente por meio de senhas e organização biométrica.	Por meio das configurações da equipe de segurança de uma organização.
É visível para o usuário?	Sim	Não
O usuário pode alterá-lo?	Possivelmente	Não
Como os dados se movem?	Por meio de tokens de identificação.	Por meio de tokens de acesso.

Token

Um token é uma chave criptografada, contendo informações do usuário, que só pode ser descriptografada com outra chave (privada) que fica no servidor.

Pode parecer estranho dizer que temos informações do usuário (não sensíveis) em uma chave, mas isto garante que não será necessário armazenar esses tokens, tornando possível o que chamamos de aplicações sem estado.

Isso é fundamental para o escalonamento de qualquer aplicação. Afinal, sempre que uma máquina nova da sua aplicação é provisionada, sua memória não é copiada.

JWT - JSON Web Token

JSON Web Token (JWT) é um padrão aberto (RFC 7519) que define uma forma compacta e independente para transmitir informações com segurança entre as partes como um objeto JSON. Essas informações podem ser verificadas e confiáveis porque são assinadas digitalmente. Os JWTs podem ser assinados usando um segredo (com o algoritmo HMAC) ou um par de chaves pública / privada usando RSA ou ECDSA.

Embora os JWTs possam ser criptografados para fornecer sigilo entre as partes, nos concentraremos nos tokens assinados. Os tokens assinados podem verificar a integridade das declarações contidas nele, enquanto os tokens criptografados ocultam essas declarações de outras partes. Quando os tokens são assinados usando pares de chave pública / privada, a assinatura também certifica que apenas a parte que possui a chave privada é quem a assinou.

Quando você deve usar JSON Web Tokens?

Aqui estão alguns cenários em que JSON Web Tokens são úteis:

Autorização: este é o cenário mais comum para o uso do JWT. Depois que o usuário estiver conectado, cada solicitação subsequente incluirá o JWT, permitindo que o usuário acesse rotas, serviços e recursos permitidos com esse token. O **Single Sign On** (SSO) é um recurso que usa amplamente o JWT hoje em dia, devido à sua pequena sobrecarga e sua capacidade de ser facilmente usado em diferentes domínios.

Troca de informações: JSON Web Tokens são uma boa maneira de transmitir informações entre as partes com segurança. Como os JWTs podem ser assinados - por exemplo, usando pares de chaves pública / privada - você pode ter certeza de que os remetentes são quem dizem ser. Além disso, como a assinatura é calculada usando o cabeçalho e a carga útil, você também pode verificar se o conteúdo não foi adulterado.

Qual é a estrutura do JSON Web Token?

Em sua forma compacta, JSON Web Tokens consistem em três partes separadas por pontos (.), que são:

- Cabeçalho (**Header**)
- Carga útil (**Payload**)
- Assinatura (**Signature**)

Portanto, um JWT normalmente se parece com o seguinte.

xxxxx

.yyyyy.zzzzz

Vamos analisar as diferentes partes.

Cabeçalho (Header)

O cabeçalho normalmente consiste em duas partes: o tipo do token, que é JWT, e o algoritmo de assinatura que está sendo usado, como HMAC SHA256 ou RSA.

Por exemplo:

Figura 64 - Header JWT.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Fonte: site Jwt.io.

Então, esse JSON é codificado em **Base64** para formar a primeira parte do JWT.

Carga útil (Payload)

A segunda parte do token é a carga útil, que contém as declarações. As declarações são sobre uma entidade (normalmente, o usuário) e dados adicionais. Existem três tipos de declarações: registradas, públicas e privadas.

- **Declarações registradas (Registered claims):** trata-se de um conjunto de declarações predefinidas que não são obrigatórias, mas recomendadas, para fornecer um conjunto de declarações úteis e interoperáveis. Alguns deles são: **iss** (emissor), **exp** (tempo de expiração), **sub** (assunto), **aud** (público) e outros.
- **Declarações públicas (Public claims):** podem ser definidas à vontade por aqueles que usam JWTs. Mas para evitar colisões, eles devem ser definidos no IANA JSON Web Token Registry ou como um URI que contém um namespace resistente a colisões.
- **Declarações privadas (Private claims):** são as declarações personalizadas criadas para compartilhar informações entre as partes que concordam em usá-las e não são declarações registradas ou públicas.

Um exemplo de carga útil poderia ser:

Figura 65 - Payload JWT.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Fonte: site Jwt.io.

A carga útil é então codificada em **Base64** para formar a segunda parte do JSON Web Token.

Assinatura (Signature)

Para criar a parte da assinatura, você deve pegar o cabeçalho codificado, a carga útil codificada, um segredo, o algoritmo especificado no cabeçalho e assiná-lo.

Por exemplo, se você deseja usar o algoritmo HMAC SHA256, a assinatura será criada da seguinte maneira:

Figura 66 - Signature JWT.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Fonte: site Jwt.io.

A assinatura é usada para verificar se a mensagem não foi alterada ao longo do caminho e, no caso de tokens assinados com uma chave privada, também pode verificar se o remetente do JWT é quem diz ser.

Juntando tudo

A saída são três strings Base64-URL separadas por pontos que podem ser facilmente passadas em ambientes HTML e HTTP, embora sejam mais compactas quando comparadas aos padrões baseados em XML, como SAML.

A imagem seguinte mostra um JWT que tem o cabeçalho anterior e a carga útil codificados e é assinado com um segredo.

Figura 67 – JWT.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Fonte: site Jwt.io.

Capítulo 13. Documentando APIs

Os desenvolvedores que consomem APIs podem estar tentando resolver problemas de negócios importantes com ela. Portanto, é muito importante que eles entendam como usar nossa API de maneira eficaz. É aqui que a documentação de APIs entra em cena.

A documentação da API é o processo de fornecer instruções sobre como usar e integrar efetivamente uma API. Portanto, pode ser considerado um manual de referência conciso contendo todas as informações necessárias para utilizá-la com detalhes sobre as funções, classes, tipos de retorno, argumentos e muito mais, apoiado por tutoriais e exemplos.

Portanto, ter a documentação adequada permite que os consumidores se integrem às APIs o mais rápido possível e sigam em frente com seu desenvolvimento, facilitando também a manutenção e suporte.

Swagger / Open API

Swagger é uma especificação independente de linguagem para descrever APIs REST. O Swagger também é conhecido como OpenAPI e nos permite entender os recursos de um serviço sem olhar para o código de implementação real, além de minimizar a quantidade de trabalho necessária ao integrar uma API. Da mesma forma, também ajuda os desenvolvedores de API a documentar suas APIs com rapidez e precisão.

Especificação Swagger

A especificação do Swagger é uma parte importante do fluxo do Swagger. Por padrão, um documento chamado swagger.json é gerado pela ferramenta Swagger, que é baseada em nossa API. Ele descreve os recursos de nossa API e como acessá-la via HTTP.

Swagger UI

O Swagger oferece uma UI baseada na web que fornece informações sobre o serviço. Isso é construído usando a especificação Swagger e incorporado dentro ao pacote **Swashbuckle** e, portanto, pode ser hospedado em nossa aplicação ASP.NET Core usando middlewares.

Integrado a UI Swagger

Podemos usar o pacote Swashbuckle para integrar facilmente o Swagger em nossos projetos .NET Core Web Api, gerando a especificação Swagger para o projeto. Além disso, a UI do Swagger também está contida no **Swashbuckle**.

Existem três componentes principais no pacote Swashbuckle:

Swashbuckle.AspNetCore.Swagger: Contém o modelo de objeto Swagger e o middleware para expor objetos **SwaggerDocument** como JSON.

Swashbuckle.AspNetCore.SwaggerGen: Um gerador Swagger que cria objetos **SwaggerDocument** diretamente das rotas, controllers e modelos.

Swashbuckle.AspNetCore.SwaggerUI: Uma versão incorporada da ferramenta Swagger UI, no qual interpreta JSON Swagger para construir uma experiência rica e personalizável para descrever a funcionalidade da Web API.

Instalando o pacote:

- O primeiro passo é instalar o pacote **Swashbuckle**.
- Podemos executar o seguinte comando na janela do console do gerenciador de pacotes
 - ***Install-Package Swashbuckle.AspNetCore***
- Isso instalará o pacote Swashbuckle em nossa aplicação.

Configurando o Swagger Middleware:

A próxima etapa é configurar o Swagger Middleware.

Vamos fazer as seguintes alterações no método `ConfigureServices()` da classe `Startup.cs`:

Figura 68 - Adicionar SwaggerGen.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<LivrosDbContext>(options => options.UseInMemoryDatabase(databaseName: "LivrosStore"));

    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1.0", new OpenApiInfo { Title = "Book Store API", Version = "v1.0" });
    });

    services.AddControllers();
}
```

Isso adiciona o gerador Swagger à coleção de serviços.

No método **`Configure()`**, vamos habilitar o middleware para servir o documento JSON gerado e a UI Swagger:

Figura 69 - Adicionar SwaggerEndpoint.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseSwagger();

    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("v1/swagger.json", "My API V1");
    });

    app.UseHttpsRedirection();

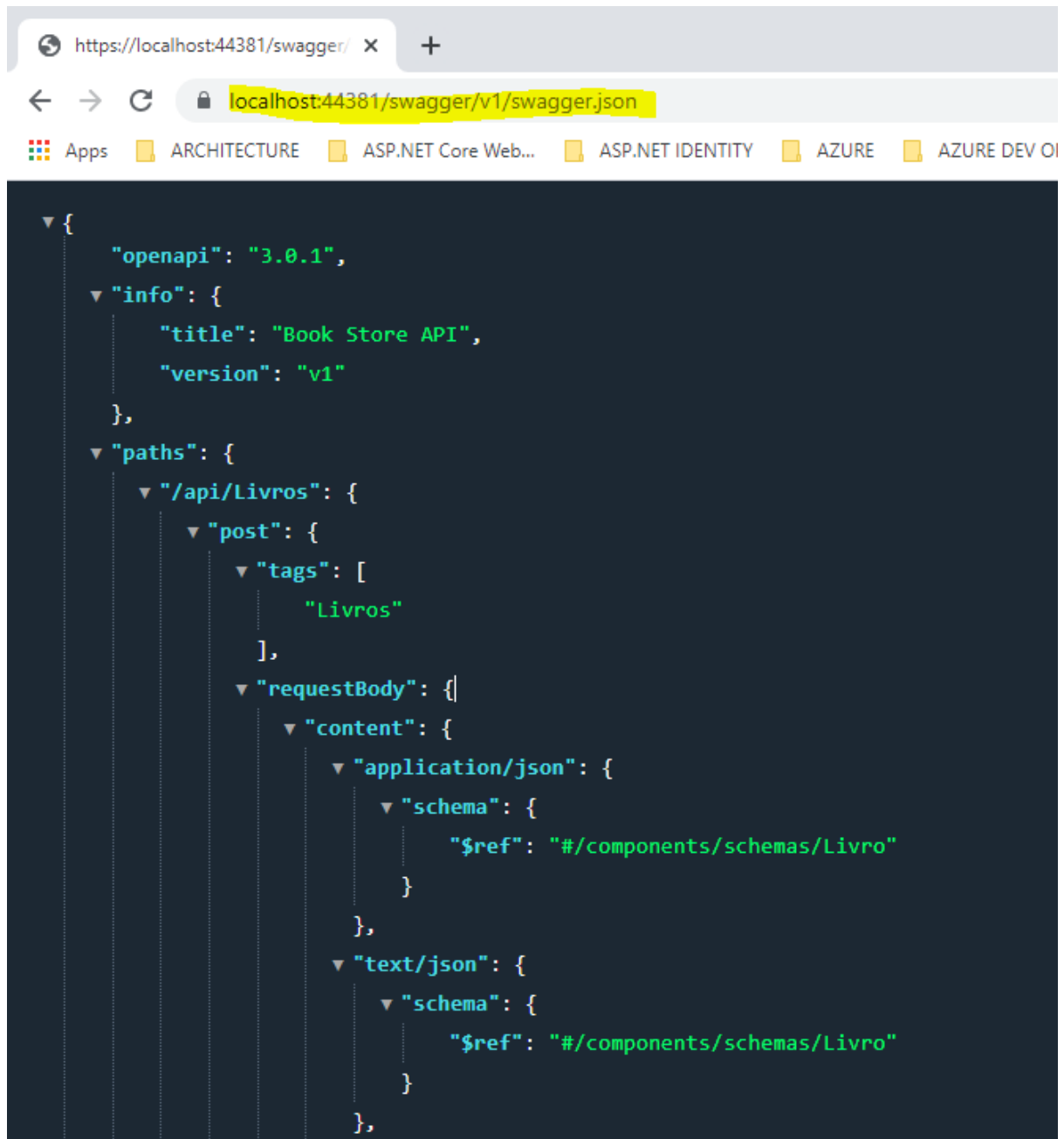
    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

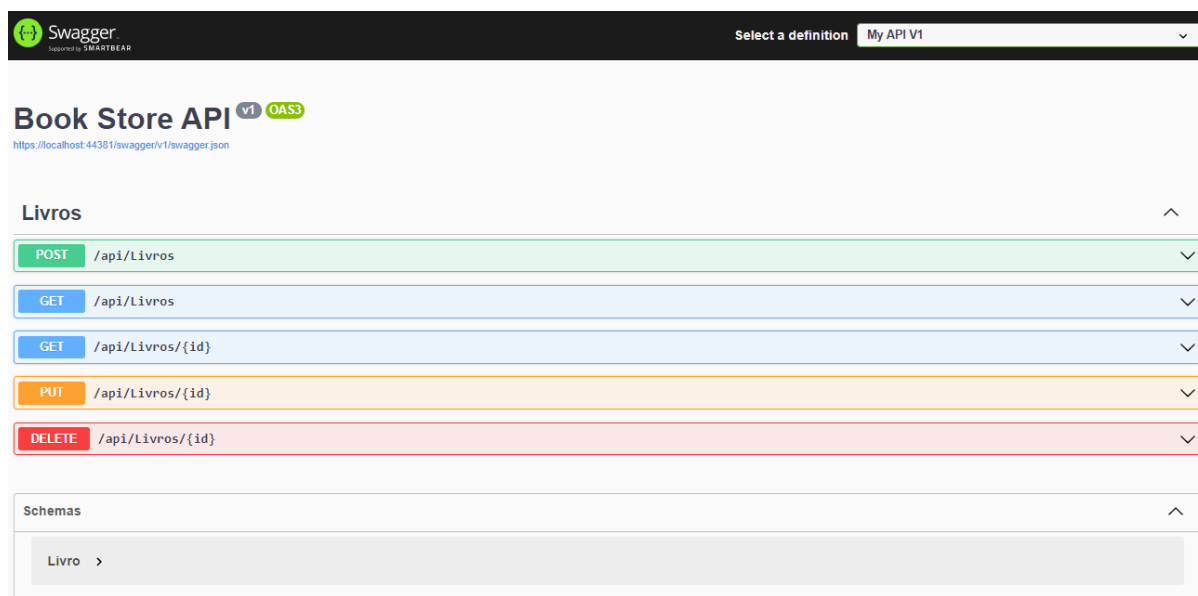
Agora, vamos executar a aplicação e navegar para **https://localhost:<port>/swagger/v1/swagger.json**. Podemos ver que um documento que descreve os endpoints é gerado:

Figura 70 - Endpoints gerados.



A UI do Swagger pode ser encontrada em <https://localhost:<port>/swagger>.

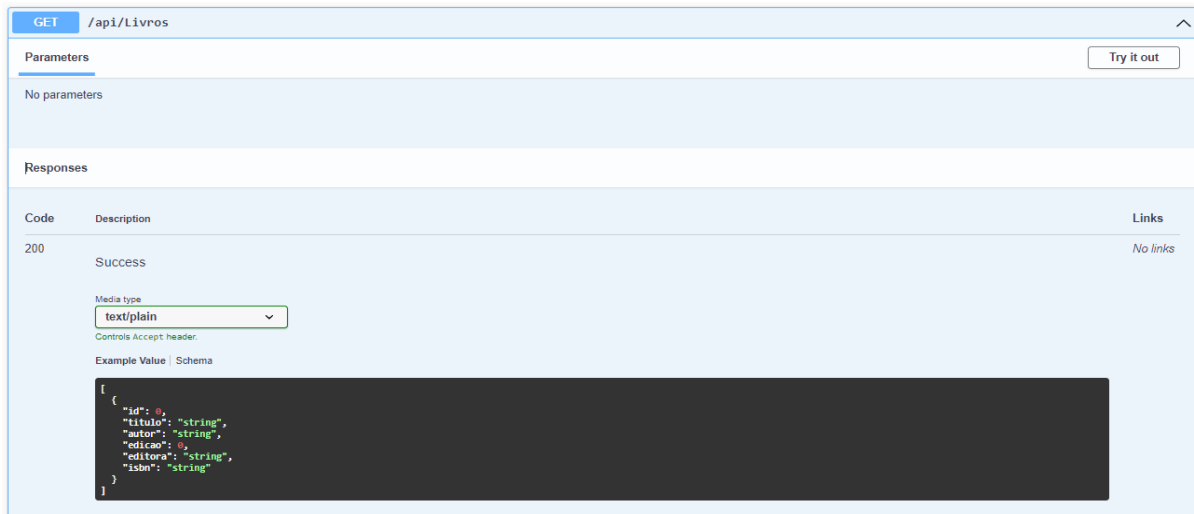
Figura 71 – Endpoints.



Agora podemos explorar a API por meio da UI do Swagger e será mais fácil incorporá-la a outras aplicações. Podemos ver cada controlador e seus métodos (actions) listados.

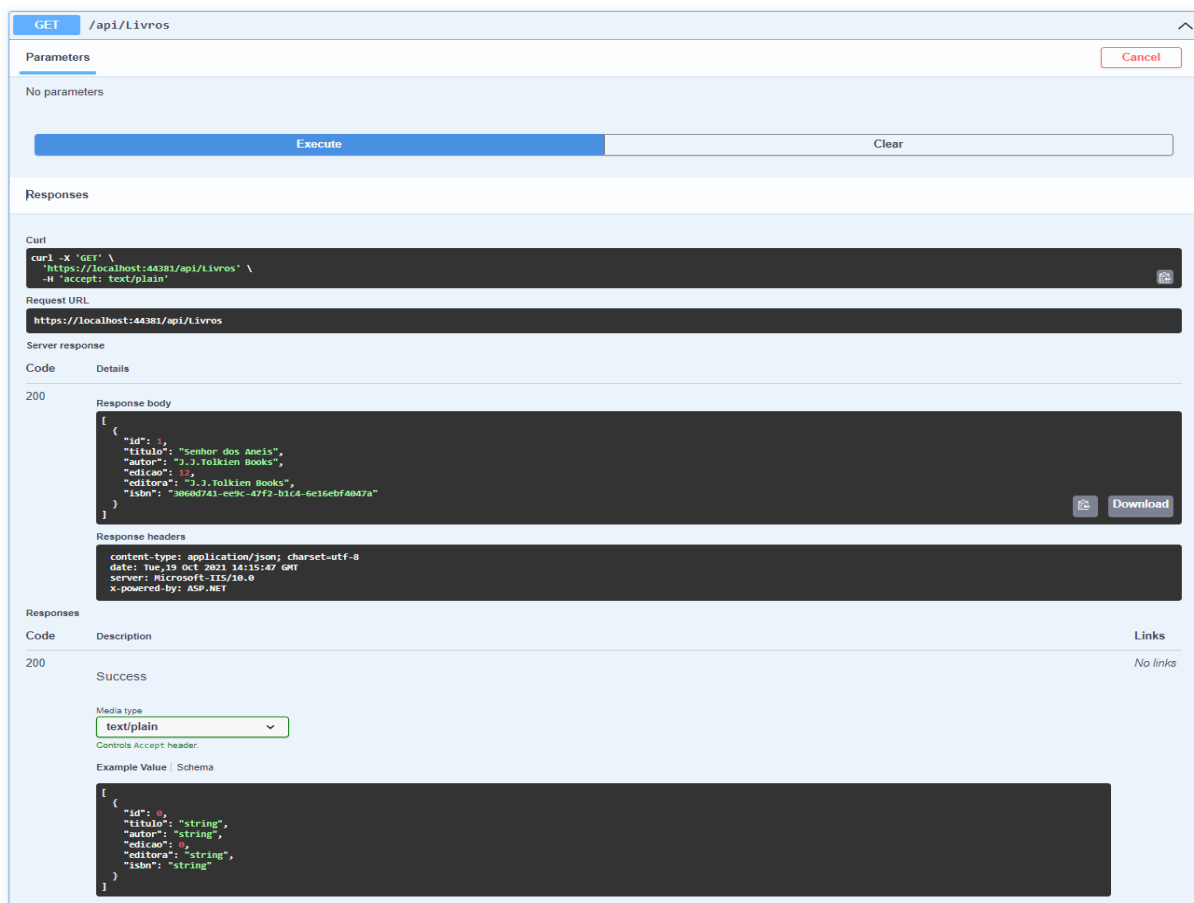
Depois de clicar em uma action (método de ação), podemos ver informações detalhadas como parâmetros, resposta e valores de exemplo. Também existe a opção de testar (**Try it out**) para cada uma das actions:

Figura 72 - Endpoint GET detalhes.



Ao clicar no botão **Try it out**, podemos testar o endpoint e ver a resposta:

Figura 73 - Endpoint GET detalhes.



Referências

ADDIE, Scott. Controller action return types in ASP.NET Core web API. **Docs Microsoft**, 14 set. 2021. Disponível em: <<https://docs.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-5.0>>. Acesso em: 04 nov. 2021.

ARQUITETURA REST: Saiba o que é e seus diferenciais. **Totvs**, 23 mar. 2020. Disponível em: <<https://www.totvs.com/blog/developers/rest/>>. Acesso em: 04 nov. 2021.

AUTHENTICATION vs. Authorization. **Auth0 docs**, c2013-2021. Disponível em: <<https://auth0.com/docs/get-started/authentication-and-authorization>>. Acesso em: 04 nov. 2021.

CHAI, Wesley. HTTP (Hypertext Transfer Protocol). **Whatis.com**, mar. 2021. Disponível em: <<https://whatis.techtarget.com/definition/HTTP-Hypertext-Transfer-Protocol>>. Acesso em: 04 nov. 2021.

CONFIGURING and Using Swagger UI in ASP.NET Core Web API. **CodeMaze**, 17 jun. 2021. Disponível em: <<https://code-maze.com/swagger-ui-asp-net-core-web-api/>>. Acesso em: 04 nov. 2021.

GUEDES, Marylene. O que é uma API?. **Treinaweb**, 2019. Disponível em: <<https://www.treinaweb.com.br/blog/o-que-e-uma-api>>. Acesso em: 04 nov. 2021.

LARKIN, Kirk; ANDERSON, Rick. Tutorial: criar uma API Web com o ASP.NET Core. **Docs Microsoft**, 07 out. 2021. Disponível em: <<https://docs.microsoft.com/pt-br/aspnet/core/tutorials/first-web-api?view=aspnetcore-5.0&tabs=visual-studio>>. Acesso em: 04 nov. 2021.

ROUTING differences between ASP.NET MVC and ASP.NET Core. **Docs Microsoft**, 15 set. 2021. Disponível em: <<https://docs.microsoft.com/en-us/dotnet/architecture/porting-existing-aspnet-apps/routing-differences>>. Acesso em: 04 nov. 2021.

ROUTING in ASP.Net Core Web API. **Dot Net Tutorials**, 2021. Disponível em: <<https://dotnettutorials.net/lesson/routing-in-asp-net-core-web-api/>>. Acesso em: 04 nov. 2021.

TEIXEIRA, Pedro Henrique Faria. **Uma API REST para a contratação de profissionais na aplicação Severino**. 2021. 61 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Universidade Federal de Uberlândia, Uberlândia, 2021. Disponível em: <<https://repositorio.ufu.br/handle/123456789/32435>>. Acesso em: 04 nov. 2021.

WHAT is Authentication? **Fortinet**, c2021. Disponível em: <<https://www.fortinet.com/resources/cyberglossary/authentication-vs-authorization>>. Acesso em: 04 nov. 2021.

WHAT is HTTP? **Cloudflare**, c2021. Disponível em: <<https://www.cloudflare.com/en-au/learning/ddos/glossary/hypertext-transfer-protocol-http/>>. Acesso em: 04 nov. 2021.