FMX Academy

INFORMAÇÕES		
MÓDULO	**AULA**	**INSTRUTOR**
05 ASP.NET Core Web API – Fundamentos	**01 - Introdução sobre APIs REST**	**Guilherme Paracatu**

1. Sumário

Objetivo: Introduzir o conceito de **APIs** e desmistificar o estilo arquitetural **REST**, focando nos princípios que o tornam o padrão de comunicação da web moderna.

Estrutura da Aula:

APIs: O Garçom Digital

- O que são APIs e por que são cruciais.
- Analogia do Garçom (API) e a Cozinha (Servidor).
- A importância da Separação Cliente-Servidor.

Introdução ao REST (Representational State Transfer)

- REST não é um protocolo, é um **Estilo Arquitetural**.
- Conceitos Centrais: **Recursos**, **Representações** e **Transferência de Estado**.
- O Protocolo HTTP como base do REST.

Os Pilares do Design RESTful

- **Stateless (Ausência de Estado):** O segredo da escalabilidade.
- **Cacheability (Cache):** O segredo da performance.
- **Interface Uniforme:** O segredo da padronização (Verbos HTTP e URIs).

Códigos HTTP e CRUD

- Mapeamento de **Verbos HTTP** para operações CRUD.
- A importância dos **Códigos de Status HTTP**.

Vantagens do REST e Contexto Moderno

- Escalabilidade, Flexibilidade e Simplicidade.
- REST vs. SOAP (Brevemente).
- Boas Práticas e o Futuro.

2. APIs e o Estilo Arquitetural REST

Objetivo: Desmistificar o conceito de API usando a analogia do garçom.

O que São APIs?

API significa **Interface de Programação de Aplicações** (Application Programming Interface). Ela é um **contrato** que define as regras de como dois softwares se comunicam. Pensem nela como a 'espinha dorsal' da web.

A Analogia do Garçom
Vamos usar a analogia mais clara para entender uma API. Pensem em um restaurante:

- **Cliente (Freguês):** O aplicativo que faz o pedido (um app móvel, um site).front-end
- **Servidor (Cozinha e Estoque):** A aplicação que tem os dados e a lógica (o backend C#, banco de dados).
- **API (Garçom):** A interface que recebe o pedido do cliente e o traduz para a cozinha.

A Abstração:
O cliente não entra na cozinha para pegar a comida, ele usa o garçom. Da mesma forma, sua aplicação cliente **não precisa saber** como o servidor armazena os dados (se é Oracle, SQL Server, etc.), apenas **como fazer o pedido** pela API.

Benefício: Separação de Interesses.

O cliente e o servidor podem evoluir de forma independente, contanto que o contrato (o garçom/API) não mude.

Termo	Descrição
API	**O contrato completo. É o sistema que lida com a autenticação, segurança, e a disponibilização geral de todos os serviços (todos os endpoints). É a biblioteca inteira que você usa.**
Endpoint	**Um ponto final de comunicação específico. É a URL específica que representa um recurso ou uma ação, e que o cliente acessa. Se a API é o restaurante inteiro, o Endpoint é a mesa (URL) e o prato (Recurso) que você pede.**

API (Restaurante): É o local completo com cardápio, garçons e cozinhas.

Endpoint (Prato Específico): É a URL exata que você chama. Exemplo: POST /api/pedidos ou GET /api/produtos/42.

Muitas pessoas usam API e Endpoint como sinônimos, mas a diferença é simples: a API é o sistema inteiro, o guarda-chuva que tem todas as regras e todos os serviços. O Endpoint é o endereço exato (a URL) de um recurso específico dentro daquela API. No ASP.NET Core, cada método dentro do nosso Controller (que lida com o GET, POST, etc.) criará um endpoint diferente.

3. Introdução ao REST (Representational State Transfer)

Objetivo: Definir REST, seus conceitos centrais (Recursos, Representações e Transferência de Estado) e sua relação com HTTP.

O Estilo Arquitetural REST:

REST não é um protocolo (como HTTP). É um **estilo arquitetural**, um conjunto de **princípios e restrições** para projetar APIs na web que sejam escaláveis, resilientes e evolutivas. Uma API que segue essas regras é chamada **RESTful.**

Gênese: Criado pelo Dr. Roy Fielding (um dos pais do HTTP e URIs) em 2000, com foco na longevidade e na escala da internet.

O Tripé do REST: Recurso, Representação e Estado

O REST gira em torno de três ideias interligadas, que são uma abstração de como a própria web funciona:

1. **Recursos: "Toda a informação é um Recurso.** É um conceito, uma funcionalidade ou um dado que pode ser nomeado. No nosso backend C#, um recurso é uma ContaBancaria, um Produto ou um Funcionario (nossas classes!)."
   - *Exemplo:* /api/usuarios, /api/produtos/42

2. **Representações: "O cliente nunca interage com o Recurso em si.** Ele interage com uma **Representação** do estado desse recurso, geralmente formatada em **JSON** (JavaScript Object Notation) ou, menos comum hoje, XML."
   - *Exemplo:* O cliente pede o produto 42, o servidor devolve um objeto JSON que é a **representação** atual do produto.

3. **Transferência de Estado: "Cada interação move o cliente para um novo estado.** O cliente envia uma requisição para o servidor. O servidor processa e transfere de volta a **representação** do novo estado (ou do estado solicitado) para o cliente. O cliente usa essa informação para fazer a próxima ação."

4. Os Pilares do Design RESTful

Objetivo: Detalhar as restrições que definem um sistema RESTful, focando nas mais críticas para a escalabilidade.

As Restrições Fundamentais (Os "Mandamentos") Sem um desses ele fica só REST

Para ser realmente RESTful, sua API deve seguir um conjunto de regras (restrições) que garantem longevidade e escalabilidade.

Restrição 1: Separação Cliente-Servidor

Cliente e Servidor devem ser totalmente independentes. A única coisa que os une é o contrato da API. A equipe de backend pode mudar o banco de dados de SQL para NoSQL, e o aplicativo móvel nem precisa saber, desde que a estrutura do JSON que ele recebe seja mantida.

Restrição 2: Stateless (Ausência de Estado)

Esta é a mais importante para a escalabilidade! **O servidor não armazena nenhuma informação de sessão do cliente entre as requisições.**

- "**O que isso significa?** Se um cliente faz a Requisição 1, o servidor processa e esquece quem ele era. Na Requisição 2, o cliente tem que reenviar toda a informação necessária (por exemplo, um token de autenticação, o carrinho de compras, etc.)."

- "**Vantagem Massiva:** Qualquer servidor pode processar qualquer requisição. Isso simplifica o balanceamento de carga: se um servidor cair, o cliente simplesmente envia a requisição para outro servidor sem perda de contexto. É a chave para a escala de internet."

Restrição 3: Cacheability (Capacidade de Cache)

As respostas do servidor devem ser marcadas como 'cacheáveis' ou não. Se o cliente sabe que a informação não muda (ex: a foto do perfil de um usuário), ele pode guardar (cachear) essa resposta localmente e usá-la em requisições futuras, em vez de pedir novamente ao servidor. Isso economiza recursos no servidor e acelera a performance para o usuário.

Restrição 4: Sistema em Camadas

O cliente não precisa saber se está falando direto com o servidor de dados, ou se há um balanceador de carga, um firewall ou um gateway de segurança no meio. A arquitetura em camadas permite que você introduza serviços intermediários (que tratam da segurança, do cache, etc.) sem quebrar o cliente.

Restrição 5: Interface Uniforme (Foco principal na próxima etapa)

Esta é a base da padronização e a mais definidora. Significa que a forma de interagir

com **qualquer** recurso deve ser consistente e padronizada. Isso é alcançado usando URIs e Verbos HTTP de forma clara.

No JSON de resposta(response) mostra todas as URIs que podem ser usadas (assim que vc chama um request)

5. Códigos HTTP e CRUD

Objetivo: Mapear a padronização REST/HTTP para operações de dados (CRUD).

ou métodos http

URI e Verbos HTTP: O Vocabulário da API

A padronização RESTful é construída em dois pilares do HTTP:

1. **URIs (Uniform Resource Identifiers):** Devem representar os **recursos (substantivos)**, não as ações (verbos).
   - *Boa Prática:* /api/produtos/42 (Identifica o recurso: Produto 42)
   - *Má Prática:* /api/deletarProduto?id=42
2. **Métodos HTTP (Verbos):** Definem a **ação (o verbo)** a ser executada no recurso.

Tabela: Mapeamento CRUD

O front usa esse código pra saber oq fazer

Método HTTP	Operação CRUD	Ação Pretendida (No Recurso)	Código de Sucesso Típico
GET	Read (Ler)	Obter dados de um recurso ou coleção.	**200 OK**
POST	Create (Criar)	Criar um novo recurso (geralmente em uma coleção).	**201 Created**

PUT	Update (Substituir)	**Substituir** o estado completo de um recurso.	**200 OK / 204 No Content**
DELETE	Delete (Excluir)	Remover um recurso.	**200 OK / 204 No Content**
PATCH	Update (Parcial)	Modificar parcialmente um recurso existente.	**200 OK / 204 No Content**

Códigos de Status HTTP: A Resposta do Servidor

"Códigos de Status: O Feedback da Cozinha"

O servidor deve retornar um código padronizado que informa ao cliente o resultado da operação. Classes de Códigos:

- **2xx (Sucesso):** Deu certo! (Ex: **200 OK**, **201 Created**).
- **4xx (Erro do Cliente):** A culpa é do cliente! (Ex: **400 Bad Request** - JSON inválido; **404 Not Found** - URI errado; **401 Unauthorized** - sem autenticação).
- **5xx (Erro do Servidor):** A culpa é do servidor! (Ex: **500 Internal Server Error** - erro de código inesperado).

front

Exemplo: "Se você tenta criar um usuário (POST /api/users) e o servidor aceita, ele retorna **201 Created** e o objeto do novo usuário. Se você tenta acessar um usuário que não existe (GET /api/users/999), ele retorna **404 Not Found.**"

Tipo	Regra de Ouro	Boa Prática
URI	Representar **Recursos (Substantivos).** no plural	/api/produtos/42
Endpoint	**URI + Verbo HTTP.** É a combinação única que define a operação. ao contrário verbo + uri	GET /api/produtos/42

Método HTTP	Operação CRUD	Ação Pretendida	**Endpoint**
GET	Read (Ler)	Obter dados de um recurso.	GET /api/produtos
POST	Create (Criar)	Criar um novo recurso.	POST /api/produtos
PUT	Update (Substituir)	Substituir o estado completo.	PUT /api/produtos/42
DELETE	Delete (Excluir)	Remover um recurso.	DELETE /api/produtos/42

Vantagens do REST e Contexto Moderno

Objetivo: Sintetizar os benefícios do REST e situá-lo no contexto das APIs.

Por Que REST Dominou o Jogo?

O sucesso do REST se deve aos benefícios que suas restrições (principalmente o **Stateless**) proporcionam:

- **Escalabilidade (Massiva):** A ausência de estado permite adicionar servidores dinamicamente, sem complexidade de sessão.
- **Flexibilidade/Portabilidade:** Agnosticismo de tecnologia. O backend C# fala com um frontend React e um app móvel Swift.
- **Simplicidade e Confiabilidade:** Construído sobre o padrão HTTP, o que o torna fácil de entender, implementar e depurar.
- **Desacoplamento Organizacional:** Permite que equipes (frontend, backend, mobile) trabalhem em paralelo, dependendo apenas do contrato da API.

REST vs. SOAP: Uma Nota Rápida

Antes do REST, o padrão era o **SOAP** (Simple Object Access Protocol). Embora ainda usado em sistemas corporativos legados (finanças, governo), o REST o superou devido à sua **simplicidade** e **leveza**.

- **SOAP:** Protocolo rígido, baseado em XML (mais verboso), mais complexo de configurar e lento.

- **REST:** Estilo leve, baseado em JSON (mais conciso), fácil de usar e mais rápido.

Conclusão:

O REST ganhou porque era mais simples, mais rápido e mais fácil de usar, aproveitando a estrutura já existente da internet (HTTP).

REST se estabeleceu como a 'língua franca' da web, e é com ele que iniciaremos nossa jornada em ASP.NET Core.

Próxima Aula: Na próxima aula, vamos abrir o VS Code e criar o nosso primeiro projeto ASP.NET Core, onde aplicaremos todos esses princípios na prática, construindo nosso primeiro recurso e endpoint!