

INFORMAÇÕES

MÓDULO	AULA	INSTRUTOR
05 ASP.NET Core Web API - Fundamentos	02 - Criando Sua Primeira Web API	Guilherme Paracatu

1. Configuração do Projeto e Desvendando a Estrutura

OBJETIVO: Criar um projeto ASP.NET Core Web API, entender sua estrutura básica, e implementar um Controller completo (GET, POST, PUT, DELETE) para o recurso Cliente, utilizando dados *mockados* e testando com o Swagger.

FERRAMENTAS: Terminal (VS Code) e C# .NET 8.

1.1. Criando o Projeto via Terminal

O primeiro passo para um desenvolvedor C# moderno é dominar a linha de comando (dotnet CLI).

Instruções (Para Digitar no Terminal do VS Code):

1. Abra o terminal integrado do VS Code (Ctrl + `).
2. Execute o comando para criar a Web API no padrão .NET 8:

```
dotnet new webapi -n LojaApi -f net8.0 --use-controllers
```

Comando	Função	Explicação
dotnet new webapi	Comando base do .NET CLI.	Cria um novo projeto baseado no template de Web API.
-n LojaApi	Nome do Projeto.	Define o nome da pasta e do projeto principal como LojaApi .
-f net8.0	Framework de Destino.	Especifica que o projeto deve usar o .NET 8.0 (a versão mais recente com suporte de longo prazo).
--use-controllers	Estrutura Clássica.	<i>Crucial!</i> Diz ao .NET para usar o modelo MVC/Controller (o padrão que usaremos), em vez do novo modelo Minimal APIs.

3. Abra a pasta do projeto no VS Code:

```
code .
```

2. Entendendo a Estrutura Base

Ao abrir a pasta LojaApi, você verá alguns arquivos e pastas essenciais. Vamos entender o papel de cada um:

Arquivo/Pasta	Função	Analogia
Program.cs	Ponto de Entrada e Configuração	O coração do aplicativo. Define quais serviços serão usados (banco de dados, autenticação, Swagger, etc.) e roda a aplicação .
appsettings.json	Configurações Globais	O arquivo de "Configurações". Usado para armazenar dados que mudam entre ambientes (chaves de conexão com banco de dados, chaves de API, etc.).
LojaApi.csproj	Definição do Projeto	O "Manifesto" do projeto. Lista os pacotes NuGet que o projeto usa, a versão do .NET e as configurações de compilação.
Controllers/	Controladores	O "Garçom Principal". Contém as classes que recebem as requisições HTTP e decidem qual lógica de negócio executar.
obj/, bin/	Arquivos de Compilação	Arquivos temporários e executáveis. Não devem ser versionados (ignorados pelo Git) .

O Papel do Program.cs e o Swagger

No Program.cs, as primeiras linhas já configuram o ambiente e o **Swagger**, uma ferramenta essencial para APIs.

```
// Program.cs
// 1. O Builder cria, configura e agrupa todos os serviços da aplicação.
var builder = WebApplication.CreateBuilder(args);

// Adiciona os serviços ao contêiner de injeção de dependência.
builder.Services.AddControllers(); // Adiciona suporte para Controllers (MVC)

// Adiciona o Swagger: Serviço de Geração de Documentação
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// ... (Restante do código)
```

Swagger/OpenAPI: É um framework de documentação interativa. Quando você rodar a API, o Swagger gera uma interface web onde você pode ver todos os seus Endpoints e **testá-los** diretamente do navegador. Isso é a documentação viva e essencial para um desenvolvedor.

3. Organização de Pastas (Clean Architecture)

Para desenvolver uma API escalável e limpa, precisamos separar as responsabilidades.

Instruções (Para Criar Pastas no VS Code):

Crie as seguintes pastas na raiz do projeto LojaApi e apague o arquivo WeatherForecast.cs e o controller WeatherForecastController.cs (que são templates de exemplo):

1. Entities (Entidades)
2. Repositories (Repositórios)
3. Controllers (Controladores) - *Já existe, vamos usar.*

A estrutura de responsabilidades (que será o foco da nossa próxima aula de Arquitetura) é:

Pasta	Responsabilidade	Analogia
Entities	O Quê são os dados. Define a estrutura (classe) do objeto (Cliente, Produto, etc.).	O Modelo do recurso.
Repositories	Onde estão os dados. Lida com a lógica de acesso e manipulação de dados (banco de dados, arquivos, ou, neste caso, dados mockados).	Acesso ao "Banco de Dados".
Controllers	Como as requisições são recebidas. Recebe o HTTP, valida, chama o Repository e formata a resposta.	O Ponto de Comunicação.

4. Implementação: O Recurso Cliente

Implementação: O Recurso Cliente

Não usaremos banco de dados, então vamos simular a persistência de dados.

Entities/Cliente.cs (A Estrutura do Recurso)

Crie o arquivo Cliente.cs e defina a estrutura do nosso recurso.

```
// Entities/Cliente.cs
namespace LojaApi.Entities
{
    // A classe que representa o nosso recurso
    public class Cliente
    {
        public int Id { get; set; }
        public string Nome { get; set; } = string.Empty;
        public string Email { get; set; } = string.Empty;
        public bool Ativo { get; set; }
    }
}
```

Repositories/ClienteRepository.cs (O Banco de Dados Mockado)

Crie o arquivo ClienteRepository.cs. Este será nosso "banco de dados" em memória (static list).

```
// Repositories/ClienteRepository.cs
using LojaApi.Entities;

namespace LojaApi.Repositories
{
    public static class ClienteRepository // Usando 'static' para simplificar o acesso
    sem Injeção de Dependência
    {
        // Lista estática para SIMULAR o Banco de Dados
        private static List<Cliente> _clientes = new List<Cliente>
        {
            new Cliente { Id = 1, Nome = "Alice Silva", Email = "alice@mail.com", Ativo
= true },
            new Cliente { Id = 2, Nome = "Bruno Costa", Email = "bruno@mail.com", Ativo
= true },
            new Cliente { Id = 3, Nome = "Carlos Santos", Email = "carlos@mail.com",
Ativo = false }
        };

        private static int _nextId = 4; // Variável para gerenciar o próximo ID

        // Implementação dos métodos CRUD

        // 1. Read (Ler Todos)
        public static List<Cliente> GetAll()
        {
            return _clientes;
        }
    }
}
```

```

// 2. Read (Ler por ID)
public static Cliente? GetById(int id)
{
    // Retorna o primeiro cliente com o ID, ou null se não encontrar
    return _clientes.FirstOrDefault(c => c.Id == id);
}

// 3. Create (Criar)
public static Cliente Add(Cliente novoCliente)
{
    novoCliente.Id = _nextId++; // Atribui o próximo ID
    _clientes.Add(novoCliente);
    return novoCliente;
}

// 4. Update (Substituir/Completo)
public static Cliente? Update(int id, Cliente clienteAtualizado)
{
    var clienteExistente = _clientes.FirstOrDefault(c => c.Id == id);

    if (clienteExistente == null)
    {
        return null; // Não encontrou para atualizar
    }

    // O PUT (Update) substitui os campos
    clienteExistente.Nome = clienteAtualizado.Nome;
    clienteExistente.Email = clienteAtualizado.Email;
    clienteExistente.Ativo = clienteAtualizado.Ativo; // Assume-se que todos os
campos vieram
    return clienteExistente;
}

// 5. Delete (Excluir)
public static bool Delete(int id)
{
    var clienteParaDeletar = _clientes.FirstOrDefault(c => c.Id == id);

    if (clienteParaDeletar == null)
    {
        return false; // Não encontrou para deletar
    }

    _clientes.Remove(clienteParaDeletar);
    return true;
}
}

```

Controllers/CientesController.cs (O Garçon Principal - CRUD Completo)

Crie o arquivo CientesController.cs. Este controlador receberá as requisições e fará a ponte com o Repository.

```
// Controllers/CientesController.cs
using LojaApi.Entities;
using LojaApi.Repositories;
using Microsoft.AspNetCore.Mvc;

namespace LojaApi.Controllers
{
    [ApiController] // Indica que esta classe é um Controlador de API (sem Views)
    [Route("api/[controller]")] // Define a rota base: 'api/Cientes'
    public class CientesController : ControllerBase // Herda de ControllerBase (padrão para APIs)
    {
        // -----
        // 1. GET - Ler Recursos
        // -----

        // Endpoint: GET api/Cientes
        [HttpGet]
        public ActionResult<List<Cliente>> GetAll()
        {
            var clientes = ClienteRepository.GetAll();
            // 200 OK - Sucesso
            return Ok(clientes);
        }

        // Endpoint: GET api/Cientes/{id}
        // O {id} é um parâmetro de rota
        [HttpGet("{id}")]
        public ActionResult<Cliente> GetById(int id)
        {
            var cliente = ClienteRepository.GetById(id);

            if (cliente == null)
            {
                // 404 Not Found - Recurso não encontrado
                return NotFound();
            }

            // 200 OK - Sucesso
            return Ok(cliente);
        }

        // -----
        // 2. POST - Criar Recurso
        // -----
    }
}
```

```

// Endpoint: POST api/Clientes
[HttpPost]
public ActionResult<Cliente> Add(Cliente novoCliente)
{
    // Validação simples (o ideal é fazer validações mais complexas)
    if (string.IsNullOrEmpty(novoCliente.Nome))
    {
        // 400 Bad Request - Erro no cliente (dados inválidos)
        return BadRequest("O nome do cliente é obrigatório.");
    }

    var clienteCriado = ClienteRepository.Add(novoCliente);

    // 201 Created - Novo recurso criado com sucesso
    // Retorna o recurso criado e a URL para acessá-lo (boa prática REST)
    return CreatedAtAction(nameof(GetById), new { id = clienteCriado.Id },
clienteCriado);
}

// -----
// 3. PUT - Atualizar/Substituir Recurso (Completo)
// -----

// Endpoint: PUT api/Clientes/{id}
[HttpPut("{id}")]
public ActionResult<Cliente> Update(int id, Cliente clienteAtualizado)
{
    // Validação de entrada
    if (string.IsNullOrEmpty(clienteAtualizado.Nome))
    {
        return BadRequest("O nome do cliente é obrigatório para
atualização.");
    }

    var cliente = ClienteRepository.Update(id, clienteAtualizado);

    if (cliente == null)
    {
        // 404 Not Found - Tentou atualizar algo que não existe
        return NotFound();
    }

    // 200 OK - Sucesso (Recurso substituído)
    return Ok(cliente);
}

// -----
// 4. DELETE - Excluir Recurso
// -----

// Endpoint: DELETE api/Clientes/{id}
[HttpDelete("{id}")]
public IActionResult Delete(int id) // Usamos IActionResult pois não

```

```

retornaremos um objeto Cliente
{
    var sucesso = ClienteRepository.Delete(id);

    if (!sucesso)
    {
        // 404 Not Found - Tentou deletar algo que não existe
        return NotFound();
    }

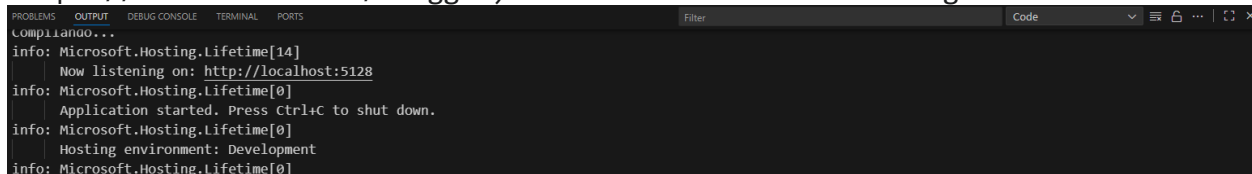
    // 204 No Content - Sucesso, mas não há corpo de resposta (padrão REST para DELETE)
    return NoContent();
}
}
}

```

5. Testes e Execução com Swagger

Instruções (Para Executar e Testar):

1. **Rodar a API:** No terminal, dentro da pasta LojaApi, execute:
`dotnet run`
2. **Acessar o Swagger:** O terminal exibirá uma URL (geralmente `https://localhost:5128/swagger`). Abra essa URL no seu navegador.



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
compilando...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5128
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]

```

3. **Testar o GET (Leitura):**
 - Clique no Endpoint GET `/api/Clientes`.
 - Clique em "Try it out" e depois em "Execute".
 - **Resultado Esperado:** Código **200 OK** e o corpo da resposta (Response body) listando os 3 clientes mockados.

4. Testar o POST (Criação):

- Clique no Endpoint POST /api/Clientes.
- Clique em "Try it out". No campo Request body, insira um novo cliente (o Id não é necessário, o Repository o cria):

```
JSON
{
  "nome": "Daniela Nova",
  "email": "daniela@mail.com",
  "ativo": true
}
```

- Clique em "Execute".
- **Resultado Esperado:** Código **201 Created** e o novo cliente (com Id: 4) no corpo da resposta.

5. Testar o DELETE (Exclusão):

- Clique no Endpoint DELETE /api/Clientes/{id}.
- Clique em "Try it out". No campo id, digite **4** (o cliente que você acabou de criar).
- Clique em "Execute".
- **Resultado Esperado:** Código **204 No Content** (sucesso).

Verificação: Execute novamente o GET /api/Clientes para confirmar que o cliente 4 foi removido.

Resumo da Aula:

- **Comando:** Criamos o projeto com `dotnet new webapi --use-controllers`.
- **Organização:** Adotamos a estrutura profissional Controllers, Entities, Repositories.
- **Encapsulamento/POO:** Usamos classes (Cliente) e um repositório (ClienteRepository) para organizar a lógica de acesso aos dados, separada do controlador.
- **REST/HTTP:** Implementamos todos os verbos **GET**, **POST**, **PUT**, e **DELETE** de forma padronizada.
- **Testes:** Utilizamos o **Swagger** para testar cada Endpoint em tempo real, verificando os códigos de status (200, 201, 404, 204).

Na próxima aula, vamos aprofundar essa estrutura, utilizando Injeção de Dependência e aplicando Interfaces para tornar o Repositório ainda mais flexível e desacoplado.