### **Introdução**

As APIs RESTful (Representational State Transfer) têm se consolidado como um padrão amplamente utilizado para integração de sistemas devido à sua simplicidade, flexibilidade e eficiência. Seguindo o paradigma REST, estas APIs permitem a comunicação entre sistemas heterogêneos de forma padronizada e orientada a recursos. Este texto explora as boas práticas para o desenvolvimento de APIs RESTful, enfatizando a importância de princípios como design limpo, consistência e segurança. Além disso, exemplifica a aplicação dessas práticas no desenvolvimento de uma API para gerenciamento de pedidos e fornecedores.

### **Desenvolvimento**

#### **1. Design Baseado em Recursos**

Uma API RESTful deve ser projetada para expor recursos bem definidos, que representem entidades do domínio do problema. No projeto, as entidades **Pedido** e **Fornecedor** foram definidas como recursos centrais, acessíveis por meio de URLs descritivas:

* **Pedidos**: /api/pedidos
* **Fornecedores**: /api/fornecedores

Cada recurso é manipulado utilizando os métodos HTTP apropriados:

* **GET**: Recupera informações.
* **POST**: Cria novos registros.
* **PUT**: Atualiza registros existentes.
* **DELETE**: Remove registros.

Exemplo de endpoint:

[HttpGet("{id}")]

public async Task<IActionResult> GetById(int id)

{

var pedido = await \_pedidoRepository.GetByIdAsync(id);

if (pedido == null)

{

return NotFound(new { Message = "Pedido não encontrado" });

}

return Ok(pedido);

}

Esse código exemplifica o uso do verbo HTTP **GET** para buscar um pedido específico pelo ID, retornando um status **404 Not Found** caso o recurso não exista.

#### **2. Uso de Convenções HTTP**

A adoção de códigos de status HTTP adequados é essencial para fornecer feedback claro ao cliente:

* **200 OK**: Operações bem-sucedidas (ex.: consulta de recursos).
* **201 Created**: Indica criação de recursos com sucesso.
* **204 No Content**: Operações de atualização ou exclusão sem retorno de corpo.
* **400 Bad Request**: Solicitações malformadas.
* **404 Not Found**: Recurso inexistente.

No projeto, essas convenções foram respeitadas. Por exemplo, após a criação de um pedido, retorna-se **201 Created** junto com a URL do novo recurso:

[HttpPost]

public async Task<IActionResult> Create(Pedido pedido)

{

if (pedido == null)

{

return BadRequest(new { Message = "Dados inválidos" });

}

await \_pedidoRepository.AddAsync(pedido);

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

}

#### **3. Estrutura de Dados e Validação**

É fundamental que os dados trafegados sejam bem estruturados e validados antes do processamento. No projeto, a API valida os dados de entrada, garantindo a integridade e a consistência das informações.  
Por exemplo, ao atualizar um pedido:

[HttpPut("{id}")]

public async Task<IActionResult> Update(int id, Pedido pedido)

{

if (id != pedido.Id)

{

return BadRequest(new { Message = "ID do pedido não corresponde" });

}

var pedidoExistente = await \_pedidoRepository.GetByIdAsync(id);

if (pedidoExistente == null)

{

return NotFound(new { Message = "Pedido não encontrado" });

}

await \_pedidoRepository.UpdateAsync(pedido);

return NoContent();

}

Essa abordagem evita inconsistências, como atualizar um recurso inexistente.

#### **4. Segurança e Boas Práticas**

Embora o projeto não implemente autenticação, recomenda-se o uso de **tokens JWT** para proteger a API contra acessos não autorizados. Além disso, o uso de **HTTPS** garante a encriptação dos dados transmitidos.

#### **5. Manutenibilidade e Padrão Repository**

O uso do padrão **Repository** facilita a manutenção do código ao desacoplar a lógica de acesso a dados da lógica de controle. No projeto, foram criadas interfaces como IPedidoRepository e IFornecedorRepository:

public interface IPedidoRepository

{

Task<IEnumerable<Pedido>> GetAllAsync();

Task<Pedido> GetByIdAsync(int id);

Task AddAsync(Pedido pedido);

Task UpdateAsync(Pedido pedido);

Task DeleteAsync(int id);

}

Esse padrão permite substituir a implementação do repositório sem impactar a lógica de negócio.

#### **6. Documentação e Testes**

A integração com o Swagger garante a documentação automatizada da API, facilitando seu uso por desenvolvedores. O Swagger foi configurado no projeto por meio do arquivo Program.cs:

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

Além disso, o uso de ferramentas como Postman e Curl foi empregado para validar as operações da API.

### **Conclusão**

O desenvolvimento de uma API RESTful eficaz requer atenção às boas práticas, como o design baseado em recursos, o uso adequado de métodos HTTP e códigos de status, além de validação de dados e segurança. No projeto apresentado, essas práticas foram aplicadas para criar uma solução robusta e escalável. O uso de padrões de projeto, como o Repository, e a documentação com Swagger, garantem a manutenibilidade e usabilidade da API, demonstrando a relevância da teoria para a prática no desenvolvimento de sistemas modernos.