UNIVERSIDADE LUTERANA DO BRASIL (ULBRA)
ANDRÉ RAMOS KARNAS

DESENVOLVIMENTO DE APIs RESTful

TORRES
2024

INTRODUÇÃO

As APIs RESTful seguem um conjunto de princípios e boas práticas que garantem que a comunicação entre sistemas seja eficiente, escalável e de fácil manutenção. Este trabalho acadêmico discute as boas práticas de desenvolvimento de APIs RESTful, explora como essas práticas foram aplicadas no desenvolvimento de uma Web API para gerenciamento de pedidos e fornecedores, e utiliza exemplos de código para ilustrar a implementação dessas práticas.

Boas Práticas no Desenvolvimento de APIs RESTful

Ao desenvolver uma API RESTful, existem várias boas práticas que ajudam a garantir a qualidade, a manutenção e a escalabilidade da aplicação. A seguir, são discutidas algumas dessas práticas:

1. Definir Endpoints Claros e Consistentes

Uma das melhores práticas ao desenvolver APIs RESTful é definir endpoints claros e intuitivos. Isso significa que os endpoints devem ser descritos de forma que qualquer desenvolvedor consiga entender o que a API faz apenas pelo nome da URL.

Por exemplo, para gerenciar pedidos e fornecedores, os endpoints poderiam ser:

GET /pedidos: Para listar todos os pedidos.

GET /pedidos/{id}: Para obter informações detalhadas sobre um pedido específico.

POST /pedidos: Para criar um novo pedido.

PUT /pedidos/{id}: Para atualizar um pedido existente.

DELETE /pedidos/{id}: Para excluir um pedido.

Na implementação do controlador PedidosController, por exemplo, temos os seguintes métodos para lidar com as operações CRUD:

```
[HttpGet]
public ActionResult<List<Pedido>> GetAll()
{
```

```
    var pedidos = _repository.GetAll();
    return Ok(pedidos);
}

[HttpPost]
public ActionResult Post(Pedido pedido)
{
    _repository.Post(pedido);
    return CreatedAtAction(nameof(Get), new { id = pedido.Id }, pedido);
}
```

Esses métodos seguem as convenções padrão de nomenclatura para endpoints RESTful e garantem que as operações CRUD sejam realizadas de forma simples e direta.

2. Utilizar o Status Code HTTP Adequado

Os status codes HTTP são uma parte fundamental das APIs RESTful, pois indicam o resultado de uma operação. É importante retornar os códigos de status adequados para as diferentes situações. Os códigos mais comuns incluem:

200 OK: Quando a operação foi bem-sucedida.

201 Created: Quando um recurso foi criado com sucesso (por exemplo, após um POST).

204 No Content: Quando uma operação foi bem-sucedida, mas não há conteúdo a ser retornado (por exemplo, após um DELETE).

400 Bad Request: Quando a requisição é inválida ou malformada.

404 Not Found: Quando o recurso solicitado não foi encontrado.

Na implementação do controlador FornecedoresController, o método Post retorna o código 201 Created quando um fornecedor é adicionado com sucesso:

```
[HttpPost]
public ActionResult Post(Fornecedor fornecedor)
{
    _repository.Post(fornecedor);
    return CreatedAtAction(nameof(Get), new { id = fornecedor.Id },
fornecedor);
}
```

Esse código garante que a resposta seja adequada ao sucesso da criação de um novo fornecedor.

3. Implementação de Validação de Dados

Uma prática essencial no desenvolvimento de APIs é validar os dados recebidos antes de processá-los. A validação garante que os dados estejam no formato correto e atendam às exigências da aplicação. Isso ajuda a evitar erros e inconsistências no banco de dados.

No caso do pedido e fornecedor, podemos validar se os campos obrigatórios estão presentes e se os dados estão em um formato válido antes de realizar operações como a criação ou atualização.

Por exemplo, ao adicionar um fornecedor, podemos verificar se o campo CNPJ está no formato adequado:

```
[HttpPost]
public ActionResult Post(Fornecedor fornecedor)
{
```

```
    if (string.IsNullOrEmpty(fornecedor.Cnpj) || fornecedor.Cnpj.Length != 18)
    {
        return BadRequest("CNPJ inválido");
    }

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

A validação evita que dados inválidos sejam persistidos no banco de dados e melhora a confiabilidade da API.

4. Utilizar a Injeção de Dependência

A injeção de dependência (DI) é um padrão que permite que as dependências de uma classe sejam fornecidas a ela em vez de serem criadas dentro da própria classe. Esse padrão promove a modularidade, facilita os testes unitários e melhora a manutenção do código.

No código da aplicação, a injeção de dependência foi usada para passar o repositório de fornecedores ou pedidos para os controladores, garantindo que o código seja desacoplado e fácil de testar.

Exemplo de injeção de dependência no controlador de fornecedores:

```
public class FornecedoresController : ControllerBase
{
    private readonly IFornecedorRepository _repository;

    public FornecedoresController(IFornecedorRepository repository)
    {
```

```
        _repository = repository;
    }

    // Métodos GET, POST, PUT, DELETE
}
```

Essa abordagem permite que o repositório seja facilmente substituído por um mock durante os testes, facilitando o processo de testes unitários.

5. Utilizar o Padrão Repository para Persistência de Dados

O padrão Repository é utilizado para separar a lógica de acesso ao banco de dados da lógica de negócios da aplicação. Isso garante que as operações de leitura e escrita no banco de dados sejam encapsuladas em uma camada separada e facilita a manutenção do código.

No exemplo abaixo, temos a implementação do repositório para o Fornecedor:

```
public class FornecedorRepository : IFornecedorRepository
{
    private readonly AppDbContext _context;

    public FornecedorRepository(AppDbContext context)
    {
        _context = context;
    }

    public void Post(Fornecedor fornecedor)
    {
        _context.Fornecedores.Add(fornecedor);
        _context.SaveChanges();
```

```
    }

    // Outros métodos CRUD
}
```

Essa camada de repositório proporciona uma separação clara entre as operações do banco de dados e a lógica de controle de fluxo da aplicação.

CONCLUSÃO

O desenvolvimento de APIs RESTful segue um conjunto de boas práticas que ajudam a garantir a eficiência, a escalabilidade e a manutenção do sistema. A utilização de métodos claros e consistentes para manipulação de dados, a aplicação correta de códigos de status HTTP, a validação de dados, a injeção de dependência e o padrão Repository são apenas algumas das boas práticas que devem ser adotadas. No desenvolvimento da Web API para gerenciamento de pedidos e fornecedores, essas práticas foram seguidas rigorosamente para criar uma API robusta, fácil de manter e de alta qualidade. O uso de exemplos de código e a aplicação dessas práticas são essenciais para garantir que o sistema seja eficiente e seguro, atendendo aos requisitos do projeto e proporcionando uma boa experiência de uso para os desenvolvedores e usuários finais.