AS LABORATÓRIO DE PROGRAMAÇÃO
APIs e Padrão Repository

O desenvolvimento de APIs RESTful tornaram-se um padrão muito utilizado no mercado nos dias atuais, isso se deve graças a sua simplicidade e escalabilidade, porém é necessário ter cuidado e fazer o uso de boas práticas para que tudo ocorra bem, melhorando a manutenção, segurança e experiência dos usuários.

1 - Uso correto dos métodos HTTP: Cada método deve ser utilizado para uma operação específica, correspondente ao método:

MÉTODO GET: Leitura de dados:

```
[HttpGet]
          public   async   Task<IActionResult>   GetAll()   =>   Ok(await
_repository.GetAllAsync());

    [HttpGet("{id}")]
    public async Task<IActionResult> GetById(int id)
    {
        var pedido = await _repository.GetByIdAsync(id);
        if (pedido == null) return NotFound();
        return Ok(pedido);
    }
```

Presente na hora de listar todos no GetAll, e no listar pelo ID em GetByIdAsync, tanto na PedidosController quanto na FornecedoresController.

MÉTODO POST: Para criar um novo dado:

```
[HttpPost]
    public async Task<IActionResult> Create(Pedido pedido)
    {
        await _repository.AddAsync(pedido);
         return CreatedAtAction(nameof(GetById), new { id = pedido.Id },
pedido);
    }
```

Presente no Create na PedidosController e na FornecedoresController.

MÉTODO PUT: Para atualizar um dado:

```
[HttpPut("{id}")]
    public async Task<IActionResult> Update(int id, Pedido pedido)
    {
        if (id != pedido.Id) return BadRequest();
        var existingPedido = await _repository.GetByIdAsync(id);
```

```
        if (existingPedido == null) return NotFound();

        await _repository.UpdateAsync(pedido);
        return NoContent();
    }
```

Presente nas duas Controllers, atualizando um dado pelo ID.

MÉTODO DELETE: Para excluir um dado:

```
[HttpDelete("{id}")]
    public async Task<IActionResult> Delete(int id)
    {
        var pedido = await _repository.GetByIdAsync(id);
        if (pedido == null) return NotFound();

        await _repository.DeleteAsync(id);
        return NoContent();
    }
```

Presente nas controllers, deletando pelo ID.

2 - Uso correto dos ENDPOINTS e Respostas HTTPS claras: Os endpoints devem ser claros e descritivos, na hora de testar no Postman por exemplo, utilizar:

GET/pedidos
POST/pedidos
GET/pedidos/{id}
PUT/pedidos/{id}
DELETE/pedidos{id}

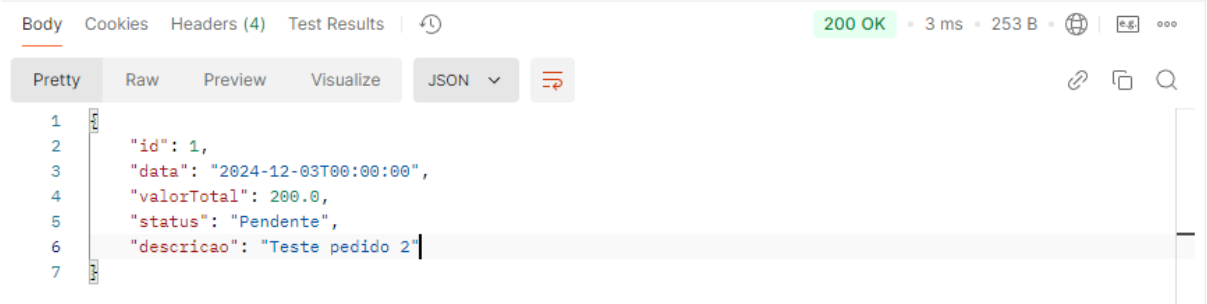
No postman, para listar o pedido com ID 1, eu utilizei dessa forma, passando um GET, o URL da minha api, depois um /api/pedidos/1, sendo o 1 o número do ID que queria listar.

Já as respostas HTTPs precisam ser claras, as principais são:

200 OK: Operação bem-sucedida.
201 Created: Recurso criado com sucesso.
404 Not Found: Recurso não encontrado.

400 Bad Request: Dados inválidos enviados pelo cliente.

Ao pressionar o botão SEND, o Postman retorna um 200, dando OK e mostrando o pedido que eu queria, esse pedido já havia sido cadastrado previamente, apenas para teste.



3 - Padrão Repository: O uso do padrão repository facilita a manutenção e realização de testes, já que ele abstrai o acesso ao banco de dados, implementando métodos assíncronos, aumentando a escalabilidade da aplicação.
Na nossa API implementamos o padrão repository nas duas classes, PedidoRepository e FornecedorRepository, mas vou exemplificar apenas a Pedido:

Aqui a interface **IPedidoRepository:**

```
public interface IPedidoRepository
{
        Task<IEnumerable<Pedido>> GetAllAsync();
        Task<Pedido> GetByIdAsync(int id);
        Task AddAsync(Pedido pedido);
        Task UpdateAsync(Pedido pedido);
        Task DeleteAsync(int id);
}
```

Agora a classe **PedidoRepository,** que está implementando a **IPedidoRepository:**

```
using Microsoft.EntityFrameworkCore;
public class PedidoRepository : IPedidoRepository {
    private readonly AppDbContext _context;

        public PedidoRepository(AppDbContext context)
        {
            _context = context;
```

```
        }
```

Por fim, adicionamos no **Program.cs:**

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
builder.Services.AddDbContext<AppDbContext>();
builder.Services.AddScoped<IFornecedorRepository,
FornecedorRepository>();
builder.Services.AddScoped<IPedidoRepository, PedidoRepository>();
```

O uso das boas práticas é essencial na criação de uma API, com uso correto dos métodos HTTP, uso correto dos endpoints, respostas HTTPs claras e padrão repository, uma API se torna escalável e de fácil manutenção. Nessa aplicação utilizei as boas práticas e por fim no Postman consegui realizar testes e ter certeza de sua funcionalidade.