FMX Academy

INFORMAÇÕES		
MÓDULO	**AULA**	**INSTRUTOR**
05 ASP.NET Core Web API – Fundamentos	**03 - Arquitetura Limpa com Injeção de Dependência e Interfaces**	**Guilherme Paracatu**

1. Injeção de Dependência

OBJETIVO: Refatorar o projeto LojaApi para introduzir a **Injeção de Dependência (DI)** em todas as camadas, utilizando **Interfaces** para desacoplar completamente o Controller, o Service e o Repository, e isolar as **Regras de Negócio.**

FERRAMENTAS: VS Code, Projeto LojaApi da aula anterior.

1. O "Porquê": Aprofundando em Arquitetura de Software

Antes de codificar, vamos entender a filosofia por trás da Injeção de Dependência (DI) e da Inversão de Controle (IoC).

1.1. O Problema do Acoplamento Forte (O "Jeito Antigo")

No início, nossas classes criavam suas próprias dependências. Isso é chamado de **Acoplamento Forte.**

Analogia do Eletricista: É como comprar um abajur (Controller) com uma lâmpada específica já soldada dentro (Repository). Se a lâmpada queimar, você precisa jogar o abajur fora ou fazer uma cirurgia complexa. É inflexível e difícil de manter.

Problemas desta abordagem:

- **Acoplamento Forte:** As classes estão "amarradas" umas às outras.
- **Dificuldade de Troca:** Trocar uma peça (como nosso repositório mockado por um de banco de dados) exige alterar o código em vários lugares.
- **Dificuldade de Teste:** É impossível testar uma peça isoladamente.

1.2. O Conceito: Inversão de Controle (IoC) - O "Princípio de Hollywood"

A **Inversão de Controle (IoC)** é um princípio que inverte o fluxo de controle. Em vez de uma classe criar suas dependências, um **container** (o ASP.NET Core) é responsável por criar e "entregar" essas dependências para ela.

A melhor analogia para IoC é o **"Princípio de Hollywood":**

> **"Não nos ligue, nós ligaremos para você."**

Nosso Controller não vai mais "ligar" (criar) o Service. Ele apenas "espera" que o framework o entregue pronto.

1.3. O Padrão: Injeção de Dependência (DI) e o Princípio SOLID (DIP)

A **Injeção de Dependência (DI)** é o padrão que usamos para implementar a IoC, geralmente via **injeção no construtor**. Isso está diretamente ligado ao **Princípio da Inversão de Dependência (DIP)** do SOLID, que nos diz:

> **"Dependa de abstrações (interfaces), não de implementações (classes concretas)."**

Conceito	Papel no Projeto	Analogia da Lâmpada
Interface (IClienteService)	O **Contrato** (o que precisa ser feito).	O **Padrão do Bocal da Lâmpada** (universal).
Implementação (ClienteService)	O **Como** (a lógica da lâmpada).	A **Lâmpada Concreta** (LED, Incandescente).
DI (ASP.NET Core)	O **Entregador** que encaixa a implementação no contrato.	O **Vendedor da Loja** que te entrega a lâmpada.

1.4. O Poder da Troca de Implementação

Com DI e Interfaces, podemos trocar a "lâmpada" (a implementação) sem precisar trocar o "abajur" (as classes que a consomem). Para mudar de um repositório de teste para um de banco de dados, só precisamos mudar **uma linha de código** no Program.cs.

Cenário de Teste: Adicionando um serviço
```
builder.Services.AddScoped<IClienteRepository, ClienteRepositoryMock>();
```

Cenário de Produção: Adicionando o banco
```
builder.Services.AddScoped<IClienteRepository, ClienteRepositoryDB>();
```
O Controller e o Service nem percebem a troca! Isso é **desacoplamento e flexibilidade.**

2. Refatorando a Aplicação (Passo a Passo)

Vamos aplicar esses conceitos para desacoplar todas as camadas da nossa LojaApi.

Passo 1: Criar a Camada de Serviço (Services)

A camada de Services é responsável por orquestrar as operações e conter as **Regras de Negócio.**

1. Crie a pasta **Services.**
2. Crie a interface **IClienteService.cs** e a implementação **ClienteService.cs.**

Services/IClienteService.cs (O Contrato do Serviço)

```
// Services/IClienteService.cs
using LojaApi.Entities;

namespace LojaApi.Services
{
    public interface IClienteService
    {
        List<Cliente> ObterTodos();
        Cliente? ObterPorId(int id);
        Cliente Adicionar(Cliente novoCliente);
        Cliente? Atualizar(int id, Cliente clienteAtualizado);
        bool Remover(int id);
    }
}
```

Services/ClienteService.cs (A Implementação com Regras de Negócio)

```
// Services/ClienteService.cs
using LojaApi.Entities;
using LojaApi.Repositories;

namespace LojaApi.Services
{
    public class ClienteService : IClienteService
    {
        // ATENÇÃO: Esta classe ainda está acoplada ao repositório estático.
        // Vamos corrigir isso nos próximos passos!

        public List<Cliente> ObterTodos()
        {
            // Regra de Negócio: Não exibir clientes inativos na listagem geral.
            return ClienteRepository.GetAll().Where(c => c.Ativo).ToList();
        }

        public Cliente? ObterPorId(int id)
        {
            return ClienteRepository.GetById(id);
        }

        public Cliente Adicionar(Cliente novoCliente)
        {
            // Regra de Negócio: Todo cliente novo é cadastrado com nome em maiúsculas
e como ativo.
            novoCliente.Nome = novoCliente.Nome.ToUpper();
            novoCliente.Ativo = true;
```

```
            return ClienteRepository.Add(novoCliente);
        }

        public Cliente? Atualizar(int id, Cliente clienteAtualizado)
        {
            // Regra de Negócio: O ID do cliente não pode ser alterado durante a
atualização.
            if (id != clienteAtualizado.Id)
            {
                // Em um projeto real, uma exceção seria lançada aqui.
                return null;
            }

            return ClienteRepository.Update(id, clienteAtualizado);
        }

        public bool Remover(int id)
        {
            // Regra de Negócio: Em vez de deletar fisicamente, fazemos uma "remoção
lógica".
            var cliente = ClienteRepository.GetById(id);
            if (cliente != null)
            {
                cliente.Ativo = false;
                return ClienteRepository.Update(id, cliente) != null;
            }
            return false;
        }
    }
}
```

Passo 2: Refatorar a Camada de Repositório (Repositories)

Agora, vamos desacoplar o Service do Repository.

1. Crie a pasta **Repositories/Interfaces**.
2. Crie a interface **IClienteRepository.cs**.
3. Refatore a classe **ClienteRepository.cs** para não ser mais estática e implementar a interface.

Repositories/Interfaces/IClienteRepository.cs (O Contrato do Repositório)

```
// Repositories/Interfaces/IClienteRepository.cs
using LojaApi.Entities;

namespace LojaApi.Repositories.Interfaces
{
    public interface IClienteRepository
    {
        List<Cliente> ObterTodos();
        Cliente? ObterPorId(int id);
        Cliente Adicionar(Cliente novoCliente);
```

```
        Cliente? Atualizar(int id, Cliente clienteAtualizado);
        bool Remover(int id);
    }
}
```

Repositories/ClienteRepository.cs (A Implementação Mockada)

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

namespace LojaApi.Repositories
{
    public class ClienteRepository : IClienteRepository
    {
        private readonly 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 int _nextId = 4;

        public List<Cliente> ObterTodos() => _clientes;

        public Cliente? ObterPorId(int id) => _clientes.FirstOrDefault(c => c.Id ==
id);

        public Cliente Adicionar(Cliente novoCliente)
        {
            novoCliente.Id = _nextId++;
            _clientes.Add(novoCliente);
            return novoCliente;
        }

        public Cliente? Atualizar(int id, Cliente clienteAtualizado)
        {
            var clienteExistente = ObterPorId(id);
            if (clienteExistente == null) return null;

            clienteExistente.Nome = clienteAtualizado.Nome;
            clienteExistente.Email = clienteAtualizado.Email;
            clienteExistente.Ativo = clienteAtualizado.Ativo;
            return clienteExistente;
        }

        public bool Remover(int id)
        {
            var clienteParaDeletar = ObterPorId(id);
            if (clienteParaDeletar == null) return false;
```

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

Passo 3: Conectar Tudo com Injeção de Dependência

Agora que temos nossos contratos e implementações, vamos refatorar o Service para receber sua dependência e configurar o "entregador" no Program.cs.

Services/ClienteService.cs (Refatorado com DI)

```
// Services/ClienteService.cs
using LojaApi.Entities;
using LojaApi.Repositories.Interfaces;
using LojaApi.Services;

namespace LojaApi.Services
{
    public class ClienteService : IClienteService
    {
        private readonly IClienteRepository _clienteRepository;

        // O Service agora recebe sua dependência (o contrato do repositório) via construtor.
        public ClienteService(IClienteRepository clienteRepository)
        {
            _clienteRepository = clienteRepository;
        }

        // Os métodos agora usam a dependência injetada (_clienteRepository)
        public List<Cliente> ObterTodos()
        {
            // Regra: Não exibir clientes inativos.
            return _clienteRepository.ObterTodos().Where(c => c.Ativo).ToList();
        }

        public Cliente? ObterPorId(int id)
        {
            return _clienteRepository.ObterPorId(id);
        }

        public Cliente Adicionar(Cliente novoCliente)
        {
            novoCliente.Nome = novoCliente.Nome.ToUpper();
            novoCliente.Ativo = true;
            return _clienteRepository.Adicionar(novoCliente);
        }
```

```
        public Cliente? Atualizar(int id, Cliente clienteAtualizado)
        {
            if (id != clienteAtualizado.Id) return null;
            return _clienteRepository.Atualizar(id, clienteAtualizado);
        }

        public bool Remover(int id)
        {
            var cliente = _clienteRepository.ObterPorId(id);
            if (cliente != null)
            {
                cliente.Ativo = false;
                return _clienteRepository.Atualizar(id, cliente) != null;
            }
            return false;
        }
    }
}
```

Program.cs (Configurando o "Entregador" de DI)

```
// Program.cs
using LojaApi.Repositories;
using LojaApi.Repositories.Interfaces;
using LojaApi.Services;

var builder = WebApplication.CreateBuilder(args);

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

// >>>>> CONFIGURAÇÃO DA INJEÇÃO DE DEPENDÊNCIA (DI) <<<<<

// 1. Registro do Serviço
builder.Services.AddScoped<IClienteService, ClienteService>();

// 2. Registro do Repositório
//    Sempre que alguém (como o ClienteService) pedir a Interface IClienteRepository,
//    entregue a implementação (mockada) ClienteRepository.
builder.Services.AddSingleton<IClienteRepository, ClienteRepository>();

// Configuração do Swagger
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
```

```
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

Passo 4: Refatorar o Controller para Usar DI

Por fim, o Controller. Dependendo da interface IClienteService.

Controllers/ClientesController.cs (Versão Final)

```
// Controllers/ClientesController.cs
using LojaApi.Entities;
using LojaApi.Services;
using Microsoft.AspNetCore.Mvc;

namespace LojaApi.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ClientesController : ControllerBase
    {
        private readonly IClienteService _clienteService;

        public ClientesController(IClienteService clienteService)
        {
            _clienteService = clienteService;
        }

        [HttpGet]
        public ActionResult<List<Cliente>> GetAll()
        {
            return Ok(_clienteService.ObterTodos());
        }

        [HttpGet("{id}")]
        public ActionResult<Cliente> GetById(int id)
        {
            var cliente = _clienteService.ObterPorId(id);
            if (cliente == null) return NotFound();
            return Ok(cliente);
        }

        [HttpPost]
        public ActionResult<Cliente> Add([FromBody] Cliente novoCliente)
        {
```

```
            if (string.IsNullOrWhiteSpace(novoCliente.Nome))
            {
                return BadRequest("O nome do cliente é obrigatório.");
            }
            var clienteCriado = _clienteService.Adicionar(novoCliente);
            return CreatedAtAction(nameof(GetById), new { id = clienteCriado.Id },
clienteCriado);
        }

        [HttpPut("{id}")]
        public ActionResult<Cliente> Update(int id, [FromBody] Cliente
clienteAtualizado)
        {
            if (string.IsNullOrWhiteSpace(clienteAtualizado.Nome))
            {
                 return BadRequest("O nome do cliente é obrigatório.");
            }
            var cliente = _clienteService.Atualizar(id, clienteAtualizado);
            if (cliente == null) return NotFound();
            return Ok(cliente);
        }

        [HttpDelete("{id}")]
        public IActionResult Delete(int id)
        {
            var sucesso = _clienteService.Remover(id);
            if (!sucesso) return NotFound();
            return NoContent();
        }
    }
}
```

---

3. Conclusão da Aula

O que fizemos hoje:

- **Aprofundamos em DI e IoC:** Entendemos a teoria por trás de uma arquitetura limpa.
- **Criamos Contratos (Interfaces):** Para o Service e o Repository.
- **Isolamos Regras de Negócio:** Na camada de Services.
- **Desacoplamos Todas as Camadas:** O Controller depende do IService, que por sua vez depende do IRepository.
- **Configuramos o DI:** Ensinamos o ASP.NET Core a conectar as interfaces às suas implementações concretas.

Resultado: Nossa API tem a mesma funcionalidade externa, mas sua **arquitetura interna** é agora profissional, flexível e preparada para o futuro, seguindo os princípios de **Inversão de Dependência (DIP)** e **Injeção de Dependência (DI).**