Defesa do Princípio da Inversão de Dependência (DIP)

João Miguel Caires Fernandes
17 de abril de 2025

Introdução

Este princípio dita que código alto nível não deve depender de implementações concretas de baixo nível mas sim de classes abstratas ou interfaces, ou seja, código alto nível depende de métodos definidos mas não implementados ou então de métodos implementados mas com nível elevado de abstração, basicamente este princípio dita que algo pouco concreto não deve depender de algo mais concreto para funcionar, o que faz sentido, sendo que o propósito é facilitar o teste do código, tornar o código mais organizado e mais fácil de reutilizar. Por isso em vez de fazer todo o código depender de implementações concretas, segundo este princípio todo esse código deve depender de algo mais abstrato não de algo mais concreto.

Program.cs - Ponto de Entrada

O código seguinte começa por configurar os repositórios e os serviços, configura o EntityFrameworkCore para PostgreSQL, o ASP.NET Core para a gestão de utilizadores e roles, cria a base de dados caso não exista, tenta criar um utilizador com o role "Admin"assim que a aplicação arranca e configura a pipeline da aplicação para tratamento de erros, mostrar como se usa a aplicação com o Swagger trata do roteamento enviando o pedido para o sítio certo, verifica se o utilizador tem as permissões necessárias e trata do mapeamento.

```
using Microsoft.EntityFrameworkCore;
using ComparacaoPrecos.Data;
using ComparacaoPrecos.Service;
using ComparacaoPrecos.Repository;
using Microsoft.AspNetCore.Identity;


var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<ProdutoRepository>();
builder.Services.AddScoped<CategoriaRepository>();
builder.Services.AddScoped<ProdutoLojaRepository>();
builder.Services.AddScoped<LojaRepository>();

builder.Services.AddScoped<ProdutoService>();
builder.Services.AddScoped<CategoriaService>();
builder.Services.AddScoped<ProdutoLojaService>();
builder.Services.AddScoped<LojaService>();
```

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddControllers();

var connectionString = builder.Configuration.
   GetConnectionString("DefaultConnection") ?? throw new
   InvalidOperationException("Connection string '
   DefaultConnection' not found.");
builder.Services.AddDbContext<ApplicationDbContext>(options
   =>
    options.UseNpgsql(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<ApplicationUser>(options
   =>
{
    options.Password.RequireDigit = false;
    options.Password.RequireLowercase = false;
    options.Password.RequireUppercase = false;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequiredLength = 6;
})
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    try
    {
        await SeedAdminAsync(services);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Erro ao criar administrador: {ex.
           Message}");
    }
}


if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
```

```
        app.UseExceptionHandler("/Error");
        app.UseHsts();
        app.UseSwagger();
        app.UseSwaggerUI();
    }

    app.UseRouting();

    app.UseAuthorization();

    app.MapControllers();

    app.MapStaticAssets();
    app.MapRazorPages()
       .WithStaticAssets();

    app.Run();

    async Task SeedAdminAsync(IServiceProvider serviceProvider)
    {
        var userManager = serviceProvider.GetRequiredService<
           UserManager<ApplicationUser>>();
        var roleManager = serviceProvider.GetRequiredService<
           RoleManager<IdentityRole>>();

        string adminEmail = "admin@gmail.com";
        string adminPassword = "Admin@123";

        if (!await roleManager.RoleExistsAsync("Admin"))
        {
            await roleManager.CreateAsync(new IdentityRole("Admin
               "));
        }

        if (!await roleManager.RoleExistsAsync("Admin"))
        {
            var roleResult = await roleManager.CreateAsync(new
               IdentityRole("Admin"));
            if (roleResult.Succeeded)
            {
                Console.WriteLine("Role 'Admin' criada com
                   sucesso!");
            }
            else
            {
                Console.WriteLine("Erro ao criar role 'Admin': "
                   + string.Join(", ", roleResult.Errors));
                return;
            }
        }
```

```
        var adminUser = await userManager.FindByEmailAsync(
           adminEmail);
        if (adminUser == null)
        {
            adminUser = new ApplicationUser
            {
                UserName = adminEmail,
                Email = adminEmail,
                EmailConfirmed = true
            };

            var result = await userManager.CreateAsync(adminUser,
                adminPassword);
            if (result.Succeeded)
            {
                await userManager.AddToRoleAsync(adminUser, "
                   Admin");
                Console.WriteLine("Administrador criado com
                   sucesso!");
            }
            else
            {
                Console.WriteLine("Erro ao criar administrador: "
                    + string.Join(", ", result.Errors));
            }
        }
        else
        {
            Console.WriteLine("Administrador ja existe.");
        }
    }
```

Listing 1: Program.cs

LojaRepository.cs - Exemplo de repositório

Aqui temos um exemplo de algo de alto nível do qual depende o *LojaService.cs*, uma implementação baixo nível deste múdulo, expondo então um exemplo do Princípio da Inversão de Dependência. Todas as classes do diretório **Service** e do diretório **Repository** seguem esta lógica cumprindo assim com um dos princípios do **SOLID**.

```
    using Microsoft.EntityFrameworkCore;
    using ComparacaoPrecos.Data;

    namespace ComparacaoPrecos.Repository;

    public class LojaRepository {
        private readonly ApplicationDbContext _context;

        public LojaRepository(ApplicationDbContext context) {
```

```
            _context = context;
        }


        public async Task<Loja> AddLoja(Loja loja) {
            _context.Loja.Add(loja);
            await _context.SaveChangesAsync();
            return loja;
        }


        public async Task<List<Loja>> GetAllLojas() {
            return await _context.Loja.Where(l => !l.Deleted).
               ToListAsync();
        }


        public async Task<Loja> GetLojaById(int id) {
            var loja = await _context.Loja.FirstOrDefaultAsync(l
               => l.LojaID == id && !l.Deleted) ?? throw new
               KeyNotFoundException($"Loja with ID {id} not found
               or is deleted.");
            return loja;
        }
    }
```

Listing 2: LojaRepository.cs

LojaService.cs - Ficheiro que depende do *LojaRepository.cs*

Aqui temos todo o código que depende de repositório da classe *Loja*, e como se pode verificar todo o código é bastante simples, compacto e desacoplado, ou seja atinge-se o pretendido, assim código de baixo nível (relativamente) depende de código de alto nível, é mais fácil de testar de reutilizar e em geral o código fica mais organizado.

```
    using ComparacaoPrecos.Data;
    using ComparacaoPrecos.Repository;

    namespace ComparacaoPrecos.Service;

    public class LojaService
    {
        private readonly LojaRepository _lojaRepository;

        public LojaService(LojaRepository lojaRepository)
        {
            _lojaRepository = lojaRepository;
        }

        public async Task<Loja> AddLoja(Loja loja)
```

```
        {
            return await _lojaRepository.AddLoja(loja);
        }

        public async Task<List<Loja>> GetAllLojas()
        {
            return await _lojaRepository.GetAllLojas();
        }

        public async Task<Loja> GetLojaById(int id)
        {
            return await _lojaRepository.GetLojaById(id);
        }
    }
```

Listing 3: LojaService.cs

Conclusão

O Princípio da Inversão de Dependência (DIP) é essencial para garantir um código mais flexível, modular e fácil de manter. Como demonstrado nos exemplos apresentados, a separação entre serviços e repositórios permite uma estrutura em que componentes de alto nível não dependem diretamente de implementações concretas, mas sim de implementações mais abstratas. Assim promove-se uma arquitetura mais limpa e desacoplada, facilitando não só a reutilização de código como também a realização de testes unitários. Ao seguir este princípio, tornamos o sistema mais robusto e preparado para futuras mudanças ou expansões, cumprindo com um dos pilares fundamentais da programação orientada a objetos e dos princípios SOLID.