DEPENDENCY SOULS

Do caos, nasce a ordem injetada

GABRIEL HENRIQUE AMAZONAS

POR QUE USAR INJEÇÃO DE DEPENDÊNCIA?

BENEFÍCIOS PRÁTICOS PARA MANTER SEU CÓDIGO ORGANIZADO E PROTEGIDO

A injeção de dependência é uma forma de escrever código onde, em vez de uma classe criar sozinha os objetos que precisa para funcionar, ela recebe esses objetos prontos de fora. Isso deixa o código mais limpo, organizado e fácil de modificar, já que cada parte do sistema fica com uma responsabilidade bem definida.

Além de ajudar na organização, a injeção de dependência também contribui para a segurança do código. Como os objetos são criados e configurados em um único lugar, é mais fácil aplicar validações, regras de acesso e boas práticas de segurança. Isso reduz o risco de erros e vulnerabilidades, tornando o sistema mais confiável.

01

O QUE É INJEÇÃO DE DEPENDÊNCIA?

Explicando o que é injeção de dependência

A injeção de dependência (Dependency Injection - DI) é um padrão de projeto que tem como objetivo principal desacoplar componentes dentro de uma aplicação.

Em vez de uma classe criar diretamente suas dependências, essas dependências são fornecidas a ela externamente, geralmente pelo construtor ou por meio de um container de injeção.

Esse desacoplamento facilita testes, manutenção e evolução do sistema, além de promover boas práticas de arquitetura como a inversão de controle (IoC).

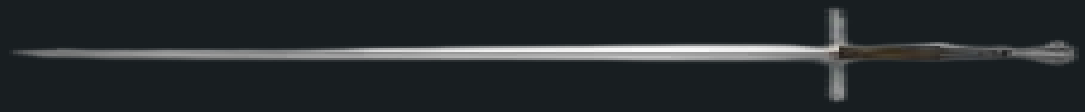
Em aplicações C#, o uso de injeção de dependência é amplamente suportado pelo framework .NET, especialmente através do pacote Microsoft.Extensions.DependencyInjection.

Exemplo:

Imagine uma aplicação que precisa enviar e-mails. Sem DI, sua classe PedidoService criaria diretamente uma instância de EmailService. Com DI, você injeta uma interface IEmailService, permitindo trocar facilmente a implementação sem alterar o código da classe principal.

02

POR QUE USAR INJEÇÃO DE DEPENDÊNCIA?



MOTIVOS PARA UTILIZAR

- Ao aplicar DI, você torna seu código mais modular, reutilizável e fácil de testar. Dependências podem ser simuladas em testes unitários (usando mocks), o que reduz a complexidade e melhora a confiabilidade do software.
- Ademais, a centralização da configuração das dependências torna o sistema mais seguro. Pode-se, por exemplo, aplicar camadas de validação ou criptografia em um único ponto de entrada, minimizando o risco de falhas distribuídas em diferentes partes do sistema.
- Além disso, ao utilizar interfaces como contratos para suas dependências, torna-se mais simples substituir uma implementação por outra sem alterar a lógica principal da aplicação. Isso permite maior flexibilidade e facilidade na manutenção ou evolução do código.

Untitled-1

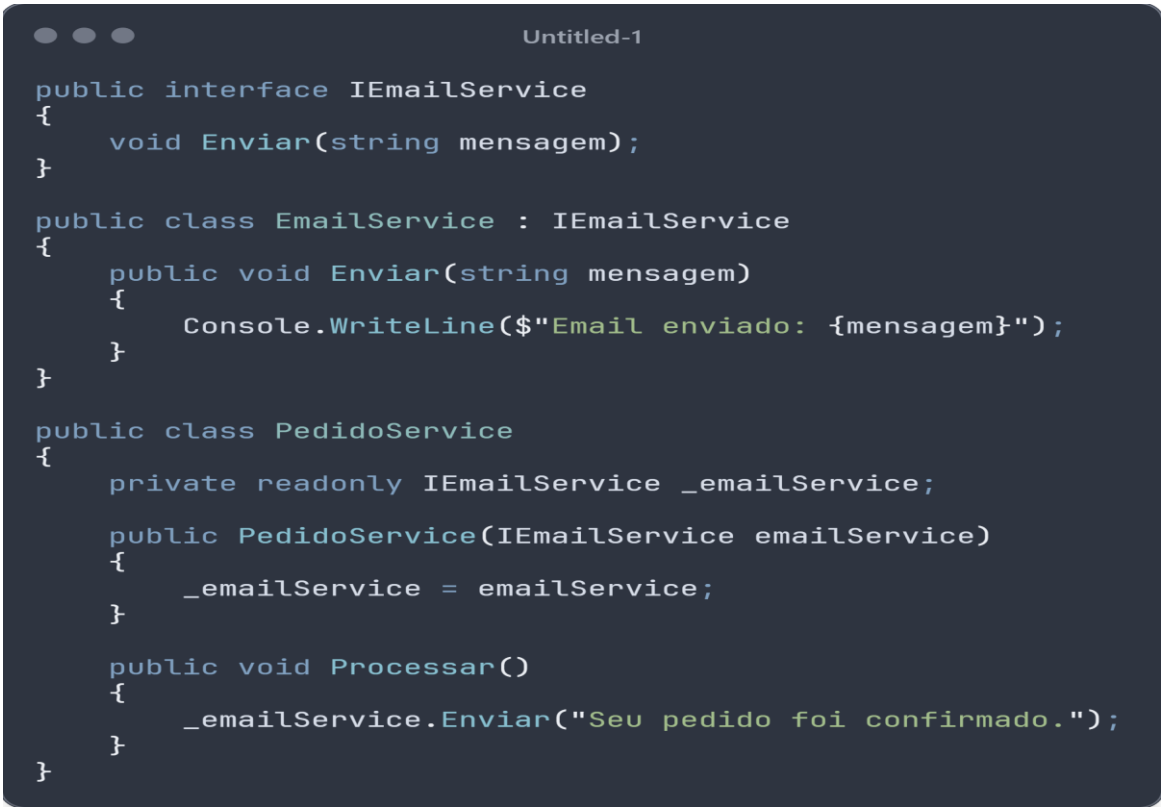
```
public interface IEmailService
{
    void Enviar(string mensagem);
}

public class EmailService : IEmailService
{
    public void Enviar(string mensagem)
    {
        Console.WriteLine($"Email enviado: {mensagem}");
    }
}

public class PedidoService
{
    private readonly IEmailService _emailService;

    public PedidoService(IEmailService emailService)
    {
        _emailService = emailService;
    }

    public void Processar()
    {
        _emailService.Enviar("Seu pedido foi confirmado.");
    }
}
```



Esse exemplo mostra como a classe PedidoService depende apenas da interface IEmailService, e não da implementação direta. Assim, qualquer classe que implemente IEmailService pode ser usada, facilitando testes e trocas de implementação.

03

FORMAS DE INJETAR DEPENDÊNCIAS EM C#

MEIOS DE INJETAR

INJEÇÃO VIA CONSTRUTOR

A forma mais comum e recomendada. As dependências são passadas como parâmetros no construtor da classe.

Untitled-1

```
public class PedidoService
{
    private readonly IPedidoRepository _pedidoRepository;

    public PedidoService(IPedidoRepository pedidoRepository)
    {
        _pedidoRepository = pedidoRepository;
    }
}
```

Caso de uso:

Em uma API, você injeta um repositório no construtor do controlador para garantir que todos os métodos da API possam usá-lo com segurança desde o início da requisição.

MEIOS DE INJETAR

INJEÇÃO VIA PROPRIEDADE

A injeção de dependência (DI) via propriedade pública (também chamada de Property Injection) em C# é uma técnica válida, porém usada em contextos bem específicos.

Untitled-1

```
public class PedidoService
{
    public IEmailService EmailService { get; set; }

    public void Processar()
    {
        EmailService?.Enviar("Pedido processado com sucesso (via propriedade).");
    }
}
```

Caso de uso:

- Quando a dependência é opcional. Se o seu componente pode funcionar sem uma dependência, mas se ela for fornecida, usará um comportamento adicional ou mais eficiente, a injeção via propriedade é uma boa escolha.
- Quando a injeção ocorre após a construção do objeto. Há casos em que o container precisa construir o objeto primeiro, e só depois injetar dependências adicionais. Isso acontece, por exemplo, com:
  - Ciclos de dependência
  - Inicialização condicional
  - Ferramentas de testes que usam mocks/dummies

- Para facilitar testes unitários (sem mudar o construtor)Se uma classe já tem muitos parâmetros no construtor, ou o construtor não pode ser alterado (por herança, por exemplo), usar injeção por propriedade permite injetar mocks diretamente nos testes:

Untitled-1

```
var service = new RelatorioService();
service.Logger = new Mock<ILogger>().Object;
```

Boas práticas:

Prefira usar public apenas se for realmente necessário; internal ou protected set pode ser mais seguro.

Documente claramente que a propriedade é de injeção, especialmente se for opcional. Se for usar DI via propriedade com um framework como ASP.NET Core, lembre-se que o Microsoft.Extensions.DependencyInjection não realiza automaticamente property injection — será necessário configuração manual ou um container mais completo, como Autofac ou Unity.

Untitled-1

```
builder.RegisterType<RelatorioService>()
       .PropertiesAutowired(); // Autofac faz injeção nas propriedades públicas
```

MEIOS DE INJETAR

INJEÇÃO VIA MÉTODO

A injeção de dependência via método (ou Method Injection) em C# é uma forma menos comum de injeção, mas pode ser extremamente útil em alguns contextos específicos. Ela consiste em passar as dependências como parâmetros de um método em vez de injetá-las via construtor ou propriedades.

Untitled-1

```
public class PedidoService
{
    public void ProcessarPedido(IEmailService emailService)
    {
        emailService.Enviar();
    }
}
```

Caso de uso:

- Quando a dependência é usada apenas em um único método. Se uma dependência não é necessária durante o ciclo de vida da classe inteira, mas sim pontualmente em um método, faz mais sentido passá-la como parâmetro.
- Para manter a classe mais enxuta. Em vez de injetar dezenas de serviços no construtor, você pode optar por passar as dependências somente quando necessário
- Facilita testes e reutilização. A classe fica mais genérica e testável, já que não depende de serviços fixos para funcionar. Exemplo:

Untitled-1

```
[Test]
public void DeveChamarExportador()
{
    var mockExportador = new Mock<IExportador>();
    var relatorio = new Relatorio();
    var exportador = new ExportadorRelatorio();

    exportador.ExportarRelatorio(relatorio, mockExportador.Object);

    mockExportador.Verify(e ⇒ e.Exportar(relatorio), Times.Once);
}
```

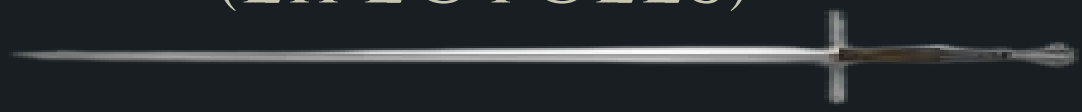
- Quando a dependência muda entre chamadas. Se cada chamada do método usa uma implementação diferente da dependência, a injeção via método é mais apropriada que via construtor ou propriedade.

OBSERVAÇÕES FINAIS

Embora menos comum, a injeção via propriedade ou método pode ser útil em cenários específicos. Como existem diversas formas de injeção de dependência, é essencial avaliar qual melhor se adapta ao seu cenário. Não existe uma forma certa ou errada — todas são viáveis conforme o contexto da aplicação.

04

CICLO DE VIDA DAS DEPENDÊNCIAS (LIFECYCLES)



CICLO DE VIDA

SINGLETON

Uma única instância é criada e compartilhada por toda a aplicação.

Ideal para serviços sem estado compartilhado, como caches ou loggers. Porém, como a instância é compartilhada, deve-se ter cuidado com recursos que mantêm estado interno, pois podem causar efeitos colaterais indesejados entre requisições.

Untitled-1

```
services.AddSingleton<IEmailService, EmailService>();
```

Exemplo:

Um serviço de log que precisa ser acessado globalmente por toda a aplicação.

CICLO DE VIDA

SCOPED

Uma nova instância é criada por requisição HTTP.

Esse ciclo de vida é indicado para serviços que devem ter um contexto próprio durante o ciclo de vida de uma requisição, como acesso a dados, autenticação ou regras de negócio específicas de um request. Fora de aplicações web, o escopo pode ser manualmente definido ao criar escopos customizados.

Untitled-1

```
services.AddScoped<IPedidoService, PedidoService>();
```

Exemplo:

Um serviço de autenticação de usuário, onde cada requisição tem seu próprio contexto.

CICLO DE VIDA

TRANSIENT

Uma nova instância é criada sempre que for solicitada.

É o mais leve dos ciclos de vida. Indicado para serviços simples e sem estado que não compartilham informações entre usos. Deve ser evitado em casos de dependências custosas para se criar, pois são recriadas constantemente.

Untitled-1

```
services.AddTransient<INotificacaoService, NotificacaoService>();
```

Exemplo:

Um serviço de envio de notificações onde cada uso pode ser independente e descartável.

CICLO DE VIDA

OBSERVAÇÕES FINAIS

O ciclo de vida determina quantas vezes e quando uma instância de uma dependência será criada. Configurar isso corretamente é essencial para:

- Garantir performance e eficiência.
- Evitar concorrência indevida (como dados compartilhados entre requisições).Evitar vazamentos de memória.
- Manter consistência de estado.

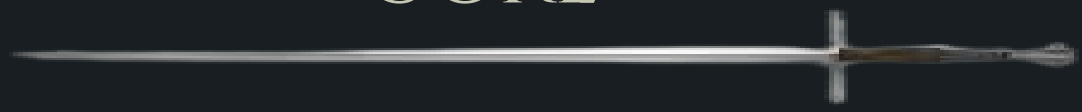
Evite misturar lifecycles de forma incorreta. Se um serviço de lifecycle maior (ex: Singleton) depende de um serviço de lifecycle menor (ex: Scoped ou Transient), isso pode causar exceções ou comportamentos imprevisíveis.

- Regra de ouro: nunca injete um Scoped ou Transient dentro de um Singleton, a menos que use um IServiceProvider ou Factory.

Em testes, o uso de Transient ou Scoped facilita isolar comportamentos. O uso de Singleton em testes pode introduzir estado compartilhado e tornar os testes não determinísticos.

05

APLICANDO DI NA PRÁTICA COM ASP.NET CORE



APLICANDO DI NA PRATICA

Imagine uma API de pedidos onde cada requisição precisa acessar o banco, enviar notificações e registrar logs. Com DI, cada serviço é injetado automaticamente com o ciclo de vida apropriado, sem que você precise instanciá-los manualmente.

Em projetos ASP.NET Core, a injeção de dependência é configurada no método ConfigureServices da classe Startup ou no builder.Services no Program.cs:

Untitled-1

```
builder.Services.AddScoped<IPedidoService, PedidoService>();
builder.Services.AddSingleton<IEmailService, EmailService>();
```

APLICANDO DI NA PRATICA

O framework injeta automaticamente as dependências nos controladores e em outras partes do sistema. Por exemplo:

Untitled-1

```
public class PedidoController : ControllerBase
{
    private readonly IPedidoService _pedidoService;

    public PedidoController(IPedidoService pedidoService)
    {
        _pedidoService = pedidoService;
    }

    [HttpPost]
    public IActionResult CriarPedido()
    {
        _pedidoService.Processar();
        return Ok("Pedido criado com sucesso!");
    }
}
```