

Respostas das Questões Técnicas do Processo Seletivo (16/07/2024)

Marco Antonio Viana de Souza (marco.antonio.viana.souza@gmail.com)

1) Fale sobre os princípios SOLID e forneça um exemplo prático de como cada princípio pode ser aplicado em uma aplicação .NET.

SOLID são princípios utilizando conceitos da orientação a objetivo que permite código de alta qualidade com fácil entendimento, manutenção, extensão e reaproveitamento.

É um acrônimo de **S**(RP), **O**(CP), **L**(SP), **I**(SP) e **D**(IP) significando:

<u>SRP</u>	Princípio da responsabilidade única (Single Responsibility Principle). O que diz: Uma classe deve ter um, e somente um, motivo para ser modificada.
<u>OCP</u>	Princípio Aberto-Fechado (The Open Closed Principle).
<u>LSP</u>	Princípio da Substituição de Liskov (The Liskov Substitution Principle).
<u>ISP</u>	Princípio da Segregação da Interface (The Interface Segregation Principle).
<u>DIP</u>	Princípio da inversão da dependência (The Dependency Inversion Principle).

Abaixo um exemplo sucinto de cada princípio

a) PRINCÍPIO: (S) → Single Responsibility Principle

O que diz: Uma classe deve ter um, e somente um, motivo para mudar.

Um exemplo que temos ainda de aplicações legadas são classes que possuem propriedades e comportamentos CRUD com acesso a banco de dados dentro. Desta forma a modificação de uma tecnologia de banco de dados levaria o sentido de modificar o código (SQL) escrito em um método como Listar, Encontrar. Na minha visão, este não é o motivo para alterar uma classe. O motivo para alteração seria a adição de um novo atributo por exemplo e para resolver isso devemos ter o acesso a banco de dados separado em outra classe, uma possível saída é aplicar o padrão Repository.

b) PRINCÍPIO: (O) → Open Closed Principle

Uso: Esse princípio enfatiza que as entidades de software (como módulos, classes e funções) devem estar **abertas para extensão** (ou seja, permitir adicionar novas funcionalidades) e **fechadas para modificação** (evitando alterações no código existente).

No exemplo abaixo, temos uma classe que calcula um desconto para clientes. Como podemos ver a taxa de desconto varia conforme o tipo de cliente. Neste caso, se novos tipos surgirem, é necessário modificar esta classe pois é somente ela que contém toda regra.

```
public class DescontoCliente
{
    public decimal CalcularDesconto(decimal valorTotal, string tipoCliente)
    {
        if (tipoCliente.ToUpper() == "REGULAR")
            return valorTotal * 0.1m; // Aplicaremos 10% de desconto
        else if (tipoCliente.ToUpper() == "BOM")
        {
            return valorTotal * 0.2m; // Aplicaremos 20% de desconto
            // Se houver novos tipos, as regras ficarão aqui, em cada Else IF, como no exemplo
            // abaixo.
        }
        else if (tipoCliente.ToUpper() == "ÓTIMO")
            return valorTotal * 0.25m; // Aplicaremos 25% de desconto
        Else
        {
            // Quando o tipo de cliente não for definido, nenhum
            // desconto será concedido.
            return 0m;
        }
    }
}
```

Exemplo da aplicação do princípio:

```
// Interface para cálculo de desconto
```

```
public interface ICalculadoraDesconto
{
    decimal CalcularDesconto(decimal valorTotal);
}
```

Criando classes que implementa a interface para implementar regras específicas conforme o tipo do desconto.

```
// Implementações específicas para cada tipo de cliente
```

```
public class DescontoRegular : ICalculadoraDesconto
{
    public decimal CalcularDesconto(decimal valorTotal) => valorTotal * 0.1m;
}
```

```
public class DescontonBom : ICalculadoraDesconto
```

```
{  
    public decimal CalcularDesconto(decimal valorTotal) => valorTotal * 0.2m;  
}
```

```
public class DescontonOtimo : ICalculadoraDesconto  
{  
    public decimal CalcularDesconto(decimal valorTotal) => valorTotal * 0.25m;  
}
```

```
// Classe que aplica o desconto  
public class ProcessadorPagamento  
{  
    private readonly ICalculadoraDesconto _calculadoraDesconto;  
  
    public ProcessadorPagamento(ICalculadoraDesconto calculadoraDesconto)  
    {  
        _calculadoraDesconto = calculadoraDesconto;  
    }  
  
    public decimal CalcularValorComDesconto(decimal valorTotal)  
    {  
        var desconto = _calculadoraDesconto.CalcularDesconto(valorTotal);  
        return valorTotal - desconto;  
    }  
}
```

c) **PRINCÍPIO: (L)** → Liskov Substitution Principle

Este princípio é ligado com o pilar da herança da Orientação a Objeto (OO).

Uma classe base deve poder ser substituída por sua classe derivada (sem alterar seu comportamento). Ou seja, caso existe uma função ou um método, em uma classe filha que tenha como herança uma classe base, então, se existir uma instância dessa classe filha, esta deve ter o comportamento igual ao da classe base.

```
// Classe base: Pessoa  
public class Pessoa  
{  
    public Guid Id { get; set; } = Guid.NewGuid();  
    public string? Nome { get; set; }  
}  
  
// Classe derivada: Funcionário  
public class Funcionario : Pessoa  
{  
    public decimal Salario { get; set; }  
  
    // Método específico para funcionários  
    public void Trabalhar()  
    {  
    }  
}
```

```

        Console.WriteLine($"{Nome} está trabalhando. Sua identificação: {Id}");
    }
}

// Classe derivada: Cliente
public class Cliente : Pessoa
{
    public string? NumeroCliente { get; set; }

    // Método específico para clientes
    public void FazerCompra()
    {
        Console.WriteLine($"{Nome} está fazendo uma compra. Seu código: {Id}");
    }
}

class Program
{
    static void Main()
    {
        // Exemplo de uso
        Pessoa pessoa1 = new Funcionario { Nome = "José", Salario = 10000 };
        Pessoa pessoa2 = new Cliente { Nome = "Pedro", NumeroCliente = "888" };

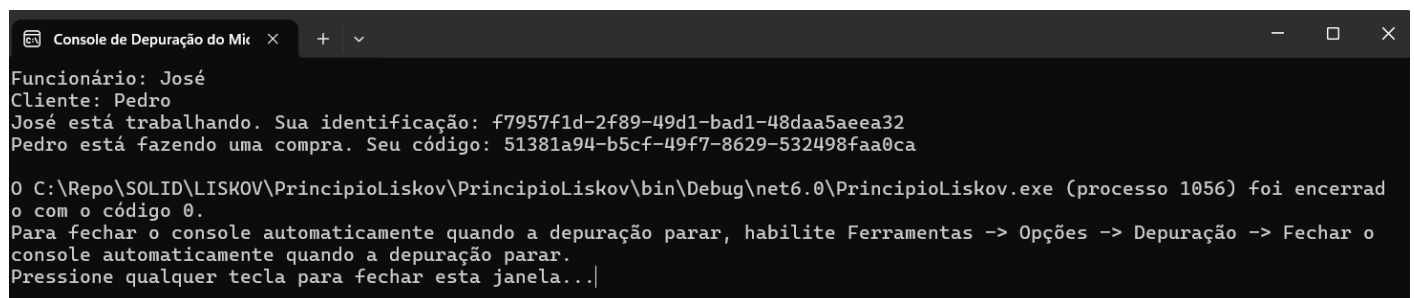
        Console.WriteLine($"Funcionário: {pessoa1.Nome}");
        Console.WriteLine($"Cliente: {pessoa2.Nome}");

        // Chamando métodos específicos
        if (pessoa1 is Funcionario func1)
            func1.Trabalhar();

        if (pessoa2 is Cliente cli1)
            cli1.FazerCompra();
    }
}

```

Resultado:



```

Console de Depuração do Mik
Funcionário: José
Cliente: Pedro
José está trabalhando. Sua identificação: f7957f1d-2f89-49d1-bad1-48daa5aeaa32
Pedro está fazendo uma compra. Seu código: 51381a94-b5cf-49f7-8629-532498faa0ca

O C:\Repo\SOLID\LISKOV\PrincipioLiskov\PrincipioLiskov\bin\Debug\net6.0\PrincipioLiskov.exe (processo 1056) foi encerrado com o código 0.
Para fechar o console automaticamente quando a depuração parar, habilite Ferramentas -> Opções -> Depuração -> Fechar o console automaticamente quando a depuração parar.
Pressione qualquer tecla para fechar esta janela...

```

d) PRINCÍPIO: (I) → Interface Segregation Principle

Este princípio, de uma forma sucinta refere-se que clientes (classes derivadas) não devem ser forçados a depender de métodos que não usam.

Exemplo que viola o princípio (ISP)

Suponha uma interface genérica chamada IPessoa com vários métodos para todas funcionalidades de uma pessoa (cliente e funcionário por exemplo).

```
public interface IPessoa
{
    void Cadastrar();
    void Atualizar();
    void Excluir();
    void EnviarEmail();
    void VerificarPeriodoDeFerias();
    // Mais métodos...
}
```

Criando a classe funcionário teremos que obrigatoriamente implementar “todos” os métodos de pessoa mesmo que algum não seja obrigatório para a classe.

```
Public class Funcionario : IPessoa
{
    public void Cadastrar()
    {
        // Código que implementa método cadastrar para funcionário
    }
    public void Atualizar()
    {
        // Código que implementa método atualizar para funcionário
    }
    public void Excluir()
    {
        // Código que implementa método excluir para funcionário
    }
    public void EnviarEmail()
    {
        // Código que implementa envio de e-mail
    }
    public void VerificarPeriodoDeFerias()
    {
        // Código que verifica se funcionário terá período de férias.
    }
}
```

O problema aqui é que a classe `Funcionario` precisa implementar todos os métodos da interface, incluindo aqueles que não fazem sentido para um funcionário (como `EnviarEmail`). Assim como implementar o método `VerificarPeriodoDeFerias` não faz sentido ser implementado numa classe `Cliente`, pois é específico para funcionário. Desta forma estamos forçando a classe a depender de métodos que ela não utilizará.

Uma das possíveis soluções para não violar este princípio seria:

```
public interface IEnvioEmail
{
    void EnviarEmail();
}
public interface IVerificarFerias
{
    void VerificarPeriodoDeFerias();
}
public interface ICadastro
{
    void Cadastrar();
}
public interface IAtualizacao
{
    void Atualizar();
}
public interface IExclusao
{
    void Excluir();
}
public interface IEnvioEmail
{
    void EnviarEmail();
}

public class Funcionario : ICadastro, IAtualizacao, IExclusao, IVerificarFerias
{
    public void Cadastrar()
    {
        // Implementação do cadastro de funcionário
    }
    public void Atualizar()
    {
        // Implementação da atualização de funcionário
    }
    public void Excluir()
    {
        // Implementação da exclusão de funcionário
    }
    public void VerificarPeriodoDeFerias()
    {

```

```

        // Implementação da validação de férias
    }
}

```

Desta forma, a classe funcionário está implementando somente os métodos necessários conforme a interface e seus contratos.

e) PRINCÍPIO: (D) → Dependency Inversion Principle

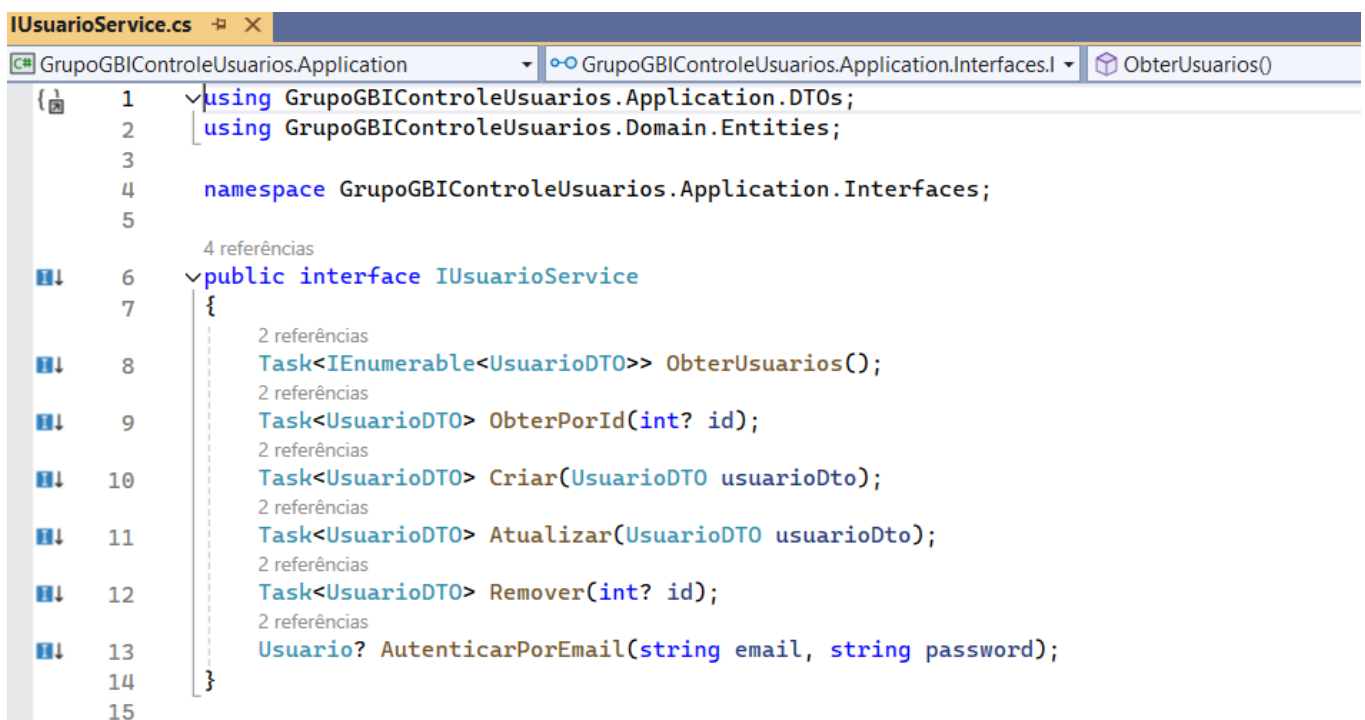
Devemos depender de “abstração” de não de implementação.

Esse princípio tem como objetivo desacoplar as dependências do projeto, induzindo que módulos de alto e baixo nível dependam de uma mesma abstração.

No .NET 6, este princípio é obtido através da criação de Interfaces e suas respectivas implementações (classe concreta). Geralmente são utilizadas em construtores “injetando a sua dependência” como parâmetro. Um container DI (container de injeção de dependência) fica responsável pelo registro dessa interface e possui a referência da classe que a implementa. Desta forma, não é necessário qualquer instanciação da classe na classe que utilizará o serviço, pois está “requisitando” um serviço e um outro mecanismo é que irá “devolver” a classe pronta para ela poder consumir seus métodos.

Exemplo:

Interface: IUserarioService



```

IUsuarioService.cs
C# GrupoGBIControleUsuarios.Application GrupoGBIControleUsuarios.Application.Interfaces.I ObterUsuarios()
{
    1 using GrupoGBIControleUsuarios.Application.DTOS;
    2 using GrupoGBIControleUsuarios.Domain.Entities;
    3
    4 namespace GrupoGBIControleUsuarios.Application.Interfaces;
    5
    6 public interface IUserarioService
    7 {
    8     Task<IEnumerable<UsuarioDTO>> ObterUsuarios();
    9     Task<UsuarioDTO> ObterPorId(int? id);
    10    Task<UsuarioDTO> Criar(UsuarioDTO usuarioDto);
    11    Task<UsuarioDTO> Atualizar(UsuarioDTO usuarioDto);
    12    Task<UsuarioDTO> Remover(int? id);
    13    Usuario? AutenticarPorEmail(string email, string password);
    14 }
    15

```

Classe que implementa interface IUserService (Classe de Serviço)

```
UsuarioService.cs | UsuarioController.cs | IUserService.cs
C# GrupoGBIControleUsuarios.Application | GrupoGBIControleUsuarios.Application.Services | _usuarioRepositorio

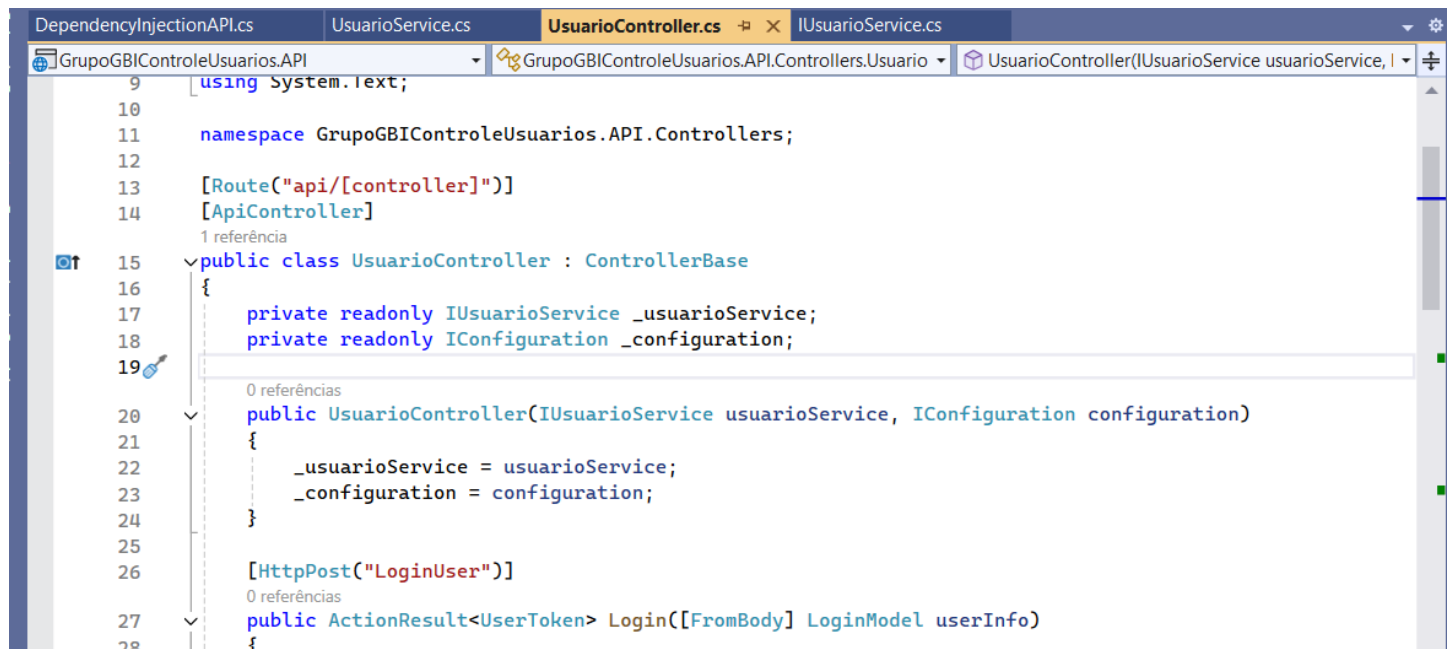
1  using AutoMapper;
2  using GrupoGBIControleUsuarios.Application.DTOs;
3  using GrupoGBIControleUsuarios.Application.Interfaces;
4  using GrupoGBIControleUsuarios.Domain.Entities;
5  using GrupoGBIControleUsuarios.Domain.Interfaces;
6
7  namespace GrupoGBIControleUsuarios.Application.Services;
8
9  2 referências
10 public class UsuarioService : IUserService
11 {
12     private IUserRepository _usuarioRepository;
13     private readonly IMapper _mapper;
14     0 referências
15     public UsuarioService(IUserRepository usuarioRepository,
16                           IMapper mapper)
17     {
18         _usuarioRepository = usuarioRepository;
19         _mapper = mapper;
20     }
21     2 referências
22     public async Task<IEnumerable<UsuarioDTO>> ObterUsuarios()
23     {
24         var usuarios = await _usuarioRepository.ObterUsuarios();
25         return _mapper.Map<IEnumerable<UsuarioDTO>>(usuarios);
26     }
27 }
```

Registro no Container de Injeção de Dependência (DI)

```
DependencyInjectionAPI.cs | UsuarioService.cs | UsuarioController.cs | IUserService.cs
C# GrupoGBIControleUsuarios.Infra.IoC | GrupoGBIControleUsuarios.Infra.IoC.Dependency | AddInfrastructureAPI(IServiceCo

34  //), b => b.MigrationsAssembly(typeof(ApplicationDbContext).Assembly.FullName));
35
36  services.AddScoped<IUsuarioService, UsuarioService>();
37  services.AddScoped<IUsuarioRepository, UsuarioRepository>();
38  services.AddAutoMapper(typeof(DomainToDTOMappingProfile));
39
40  return services;
41  }
42 }
```


Chamada no Controlador da WebAPI



```
DependencyInjectionAPI.cs | UsuarioService.cs | UsuarioController.cs | IUsuarioService.cs
GrupoGBIControleUsuarios.API | GrupoGBIControleUsuarios.API.Controllers.Usuario | UsuarioController(IUsuarioService usuarioService, I
9 | using System.Text;
10 |
11 | namespace GrupoGBIControleUsuarios.API.Controllers;
12 |
13 | [Route("api/[controller]")]
14 | [ApiController]
15 | public class UsuarioController : ControllerBase
16 | {
17 |     private readonly IUsuarioService _usuarioService;
18 |     private readonly IConfiguration _configuration;
19 |
20 |     public UsuarioController(IUsuarioService usuarioService, IConfiguration configuration)
21 |     {
22 |         _usuarioService = usuarioService;
23 |         _configuration = configuration;
24 |     }
25 |
26 |     [HttpPost("LoginUser")]
27 |     public ActionResult<UserToken> Login([FromBody] LoginModel userInfo)
28 |     {
```

2) O que são Delegates em C# e como o tipo genérico Func pode ser utilizado. Forneça um exemplo de código onde um Func é utilizado para encapsular uma função anônima que calcula a soma de dois números.

Resposta: Geralmente, o uso de Func é realizado para encapsular uma função anônima. Segue um exemplo simples do seu uso:

```
class Program
{
    // Obter dois números e retornar sua soma com Uso de Delegate e Func

    delegate int Calculadora(int a, int b);
    static void Main(string[] args)
    {
        // Criando uma função anônima que calcula a soma de dois números
        Calculadora soma = (x, y) => x + y;

        // Usando o Func para encapsular a função anônima
        Func<int, int, int> funcSoma = (x, y) => x + y;

        Console.WriteLine("Informe o primeiro número=>Numero1");
        int numero1 = Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("Informe o primeiro número=>Numero2");
        int numero2 = Convert.ToInt32(Console.ReadLine());

        int soma1 = soma(numero1, numero2);
        int soma2 = funcSoma(numero1, numero2);

        Console.WriteLine($"Resultado usando delegate: {soma1}");
        Console.WriteLine($"Resultado usando Func: {soma2}");
        Console.ReadLine();
    }
}
```

3) Explique a diferença entre as classes Task e Thread no .NET. Quando usar uma sobre a outra? Forneça um exemplo prático de uso de Task.

Resposta:

Na minha visão ambos são recursos ou técnicas utilizadas para garantir desempenho de uma aplicação e evitar que a mesma fique “travada” (isso numa forma bem resumida).

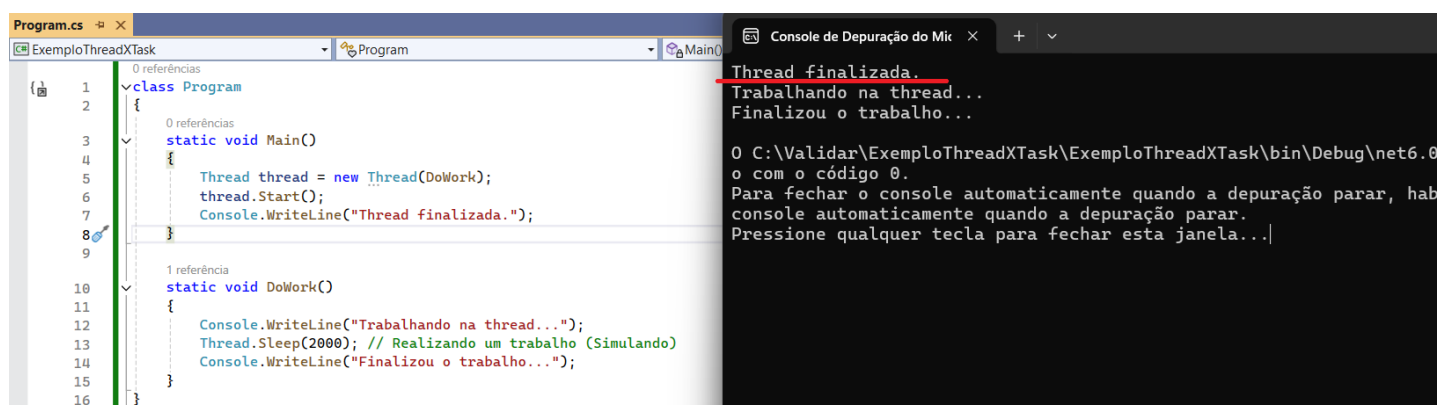
Em termos de uso, a Task representa uma tarefa ou uma operação que “sempre irá retornar um resultado”, mesmo usando o modificar `async` para explicitar uma task assíncrona ela sempre dá essa garantia de retorno.

Já uma Thread, temos o disparo de uma ação mas nem sempre a garantia do seu retorno ou sequenciamento. Ambos são técnicas de programação multitarefa.

Na Threads o processo de divide em várias tarefas executadas concorrentemente, enquanto uma Task corresponde a uma “Tarefa” que deverá ser realizada.

Exemplo de Threads

Exemplo de uso trabalhando com Thread (Sem encadeamento)

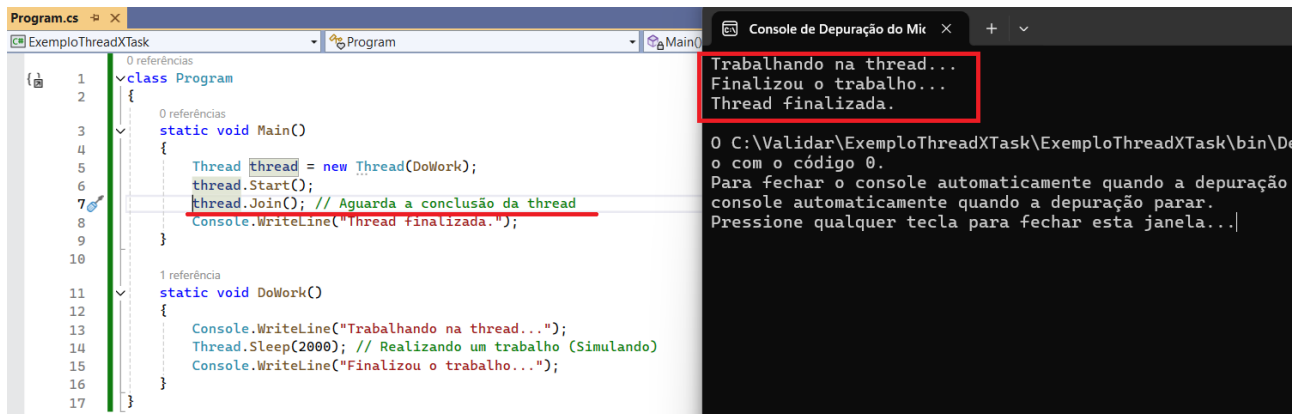


```
Program.cs
ExemploThreadXTask
0 referências
class Program
{
    0 referências
    static void Main()
    {
        Thread thread = new Thread(DoWork);
        thread.Start();
        Console.WriteLine("Thread finalizada.");
    }
    1 referência
    static void DoWork()
    {
        Console.WriteLine("Trabalhando na thread...");
        Thread.Sleep(2000); // Realizando um trabalho (Simulando)
        Console.WriteLine("Finalizou o trabalho...");
    }
}
```

```
Console de Depuração do Mic
Thread finalizada.
Trabalhando na thread...
Finalizou o trabalho...

O C:\Validar\ExemploThreadXTask\ExemploThreadXTask\bin\Debug\net6.0
o com o código 0.
Para fechar o console automaticamente quando a depuração parar, hab
console automaticamente quando a depuração parar.
Pressione qualquer tecla para fechar esta janela...|
```

Exemplo de uso trabalhando com Thread (Com encadeamento – (JOIN))



The screenshot shows a Visual Studio IDE with a C# file named `Program.cs` and a console window titled "Console de Depuração do Mik".

Program.cs:

```
1 class Program
2 {
3     static void Main()
4     {
5         Thread thread = new Thread(DoWork);
6         thread.Start();
7         thread.Join(); // Aguarda a conclusão da thread
8         Console.WriteLine("Thread finalizada.");
9     }
10
11     static void DoWork()
12     {
13         Console.WriteLine("Trabalhando na thread...");
14         Thread.Sleep(2000); // Realizando um trabalho (Simulando)
15         Console.WriteLine("Finalizou o trabalho...");
16     }
17 }
```

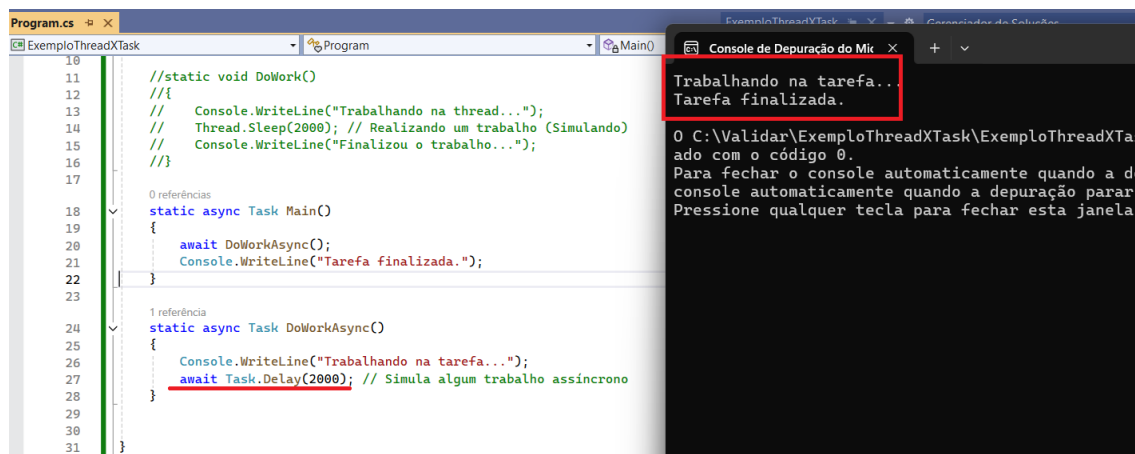
Console de Depuração do Mik:

```
Trabalhando na thread...
Finalizou o trabalho...
Thread finalizada.

O C:\Validar\ExemploThreadXTask\ExemploThreadXTask\bin\De
o com o código 0.
Para fechar o console automaticamente quando a depuração
console automaticamente quando a depuração parar.
Pressione qualquer tecla para fechar esta janela...
```

No exemplo acima, com o encadeamento da thread através da instrução `Join`, a sequência de ações foi respeitada.

Exemplo de Task



The screenshot shows a Visual Studio IDE with a C# file named `Program.cs` and a console window titled "Console de Depuração do Mik".

Program.cs:

```
10 //static void DoWork()
11 //{
12 //    Console.WriteLine("Trabalhando na thread...");
13 //    Thread.Sleep(2000); // Realizando um trabalho (Simulando)
14 //    Console.WriteLine("Finalizou o trabalho...");
15 //}
16
17
18 static async Task Main()
19 {
20     await DoWorkAsync();
21     Console.WriteLine("Tarefa finalizada.");
22 }
23
24 static async Task DoWorkAsync()
25 {
26     Console.WriteLine("Trabalhando na tarefa...");
27     await Task.Delay(2000); // Simula algum trabalho assíncrono
28 }
29
30
31 }
```

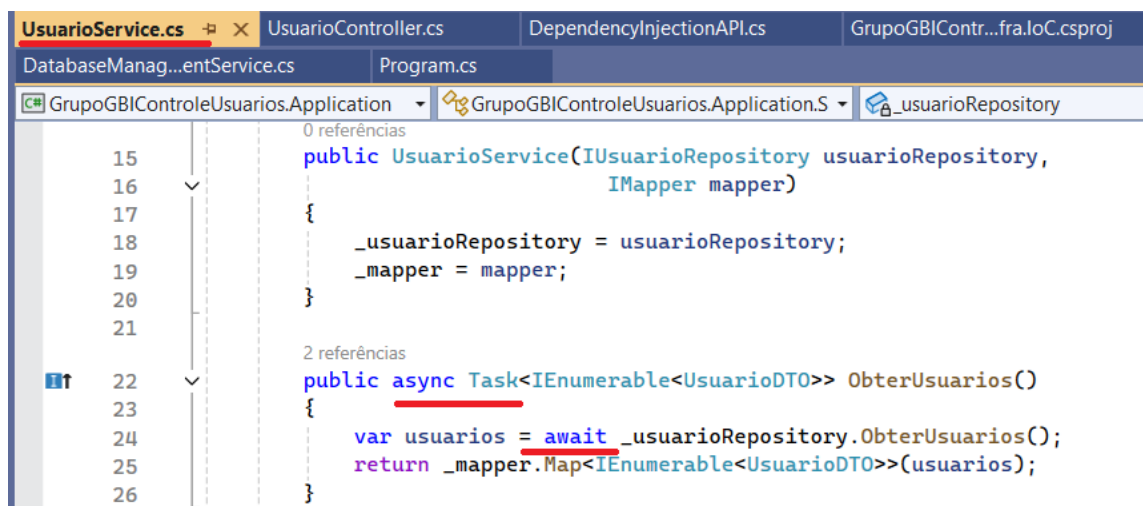
Console de Depuração do Mik:

```
Trabalhando na tarefa...
Tarefa finalizada.

O C:\Validar\ExemploThreadXTask\ExemploThreadXTask
ado com o código 0.
Para fechar o console automaticamente quando a de
console automaticamente quando a depuração parar.
Pressione qualquer tecla para fechar esta janela.
```

A instrução `await` assegura que a task terá o retorno.

Um outro exemplo comum de Task são operações envolvendo banco de dados:



The screenshot shows a Visual Studio IDE with a C# file named `UsuarioService.cs` and a console window titled "Console de Depuração do Mik".

UsuarioService.cs:

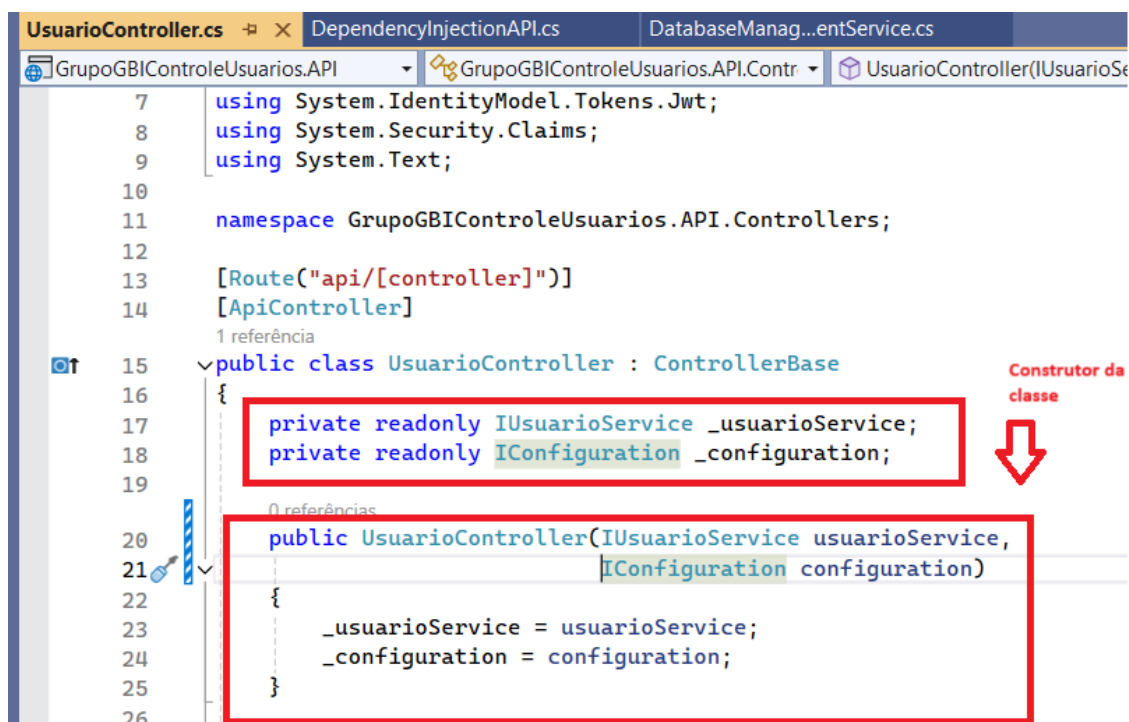
```
15 public UsuarioService(IUsuarioRepository usuarioRepository,
16                       IMapper mapper)
17 {
18     _usuarioRepository = usuarioRepository;
19     _mapper = mapper;
20 }
21
22 public async Task<IEnumerable<UsuarioDTO>> ObterUsuarios()
23 {
24     var usuarios = await _usuarioRepository.ObterUsuarios();
25     return _mapper.Map<IEnumerable<UsuarioDTO>>(usuarios);
26 }
```

Na minha opinião, o uso de Task assíncronas e pelo fato de poder interceptar exceções encadeadas (da filha para a pai) faz com que seu uso se torne mais comum.

4) O que é Dependency Injection? Explique como o .NET Core/6 implementa o padrão de injeção de dependência e forneça um exemplo de código.

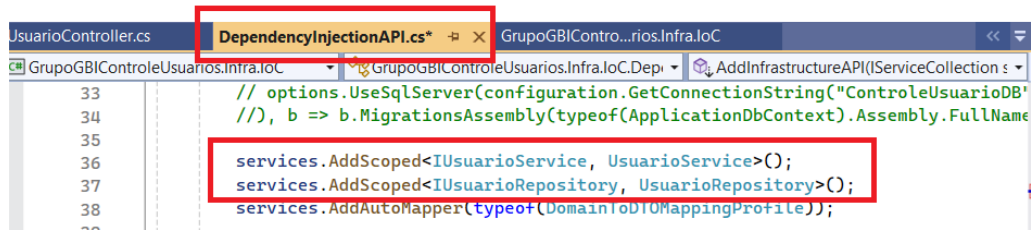
Resposta:

A injeção de dependência é um padrão de projeto que tem como objetivo remover as dependências desnecessárias entre as classes. No .NET Core /6 existe o container de injeção de dependência onde é possível registrar os contratos (Interfaces) e as classes concretas que as implementam. Seu uso geralmente é realizado injetando num construtor de uma classe. Sem esse recurso teríamos que sempre criar um objeto que precisamos dentro de outra classe e, desta forma teríamos um forte acoplamento.



```
7 using System.IdentityModel.Tokens.Jwt;
8 using System.Security.Claims;
9 using System.Text;
10
11 namespace GrupoGBIControleUsuarios.API.Controllers;
12
13 [Route("api/[controller]")]
14 [ApiController]
15 public class UsuarioController : ControllerBase
16 {
17     private readonly IUserarioService _usuarioService;
18     private readonly IConfiguration _configuration;
19
20     public UsuarioController(IUserarioService usuarioService,
21                             IConfiguration configuration)
22     {
23         _usuarioService = usuarioService;
24         _configuration = configuration;
25     }
26 }
```

No exemplo acima temos o construtor do controlador “Usuario”. Nele estamos utilizando a injeção de dependência passando duas Interfaces (IUsuarioService e IConfiguration). A definição das classes que implementam estas interfaces estão definidas no Container de Injeção de Dependência (para centralizar as interfaces e as respectivas classes que as implementam). No caso acima temos:



5) Descreva o funcionamento do Entity Framework e explique as diferenças entre Code First, Database First e Model First. Qual a abordagem que você prefere e por quê?

Resposta:

Ao desenvolver software nos deparamos com várias situações.

1)Data Base First

Em resumo podemos ter que trabalhar com cenários onde já exista um banco de dados existente (legado) e neste caso há regras de negócios, procedures, views, triggers tudo já mapeado. Neste cenário a abordagem Database First pode ser a mais indicada pois é possível gerar classes (através de ferramentas). No dotnet 6 temos o recurso do comando abaixo para gerar as classes automaticamente.

```
Scaffold-DbContext "Data Source=[servidor];Initial Catalog = NomeBD;Integrated Security=True" Microsoft.EntityFrameworkCore.SqlServer.
```

Embora exista este recurso, muitas vezes os desenvolvedores fazem uma leitura da tabela, recriando nomes para objetos e mantendo a correspondência com a tabela e campo original, geralmente utilizando data annotation.

Exemplo: Tabela: tblCliente, com campo cli_codigo

```

[Table("tblCliente")] // Este é o nome real da tabela.
Public class Cliente // Este é o nome interno para EF.
{
    [key] // Reforça que é uma chave primária.
    [Column("cli_codigo")] // Este é o campo real da tabela.
    Int clienteId {get;set;}
}

```

2) Model First

Embora não tão comum, é possível criar as classes diretamente de forma gráfica no diagrama do visual studio e a partir daí termos as propriedades das classes e seus relacionamentos. Podemos desta forma utilizar as classes criadas pelo diagrama realizar a configuração para conexão com o banco de dados. Apesar de aparentar ser ágil esta forma de desenvolvimento, o código é gerado pelo Visual Studio e, poderá conter códigos desnecessários, por isso sempre é indicado uma revisão caso use esta abordagem. Talvez para protótipos rápidos seja indicado. Eu particularmente nunca atuei desta forma.

3) Code First

De todas, esta tem sido a principal forma de trabalho para implementação de novas funcionalidades.

De uma forma geral ela simplifica e agiliza a produtividade de código pois criamos as classes e posteriormente os scripts são gerados de forma automática para serem aplicados ao banco de dados.

Outra vantagem desta abordagem consiste em que ao tratar com provedores de bancos de dados diferentes (Oracle, SqlServer, Postgree, MySql) não precisamos nos preocupar com a forma que o SQL será gerado pois esta é responsabilidade do EF. Além do mais é possível com o recurso de Migration manter todo histórico de instruções de modificação ou cargas de dados no banco de dados.

Minha opinião e escolha:

Tendo em vista a agilidade, padronização para geração de código e histórico, o uso do *Code First* é hoje a opção que escolheria para desenvolvimento e correções em projetos utilizando .NET 6.0