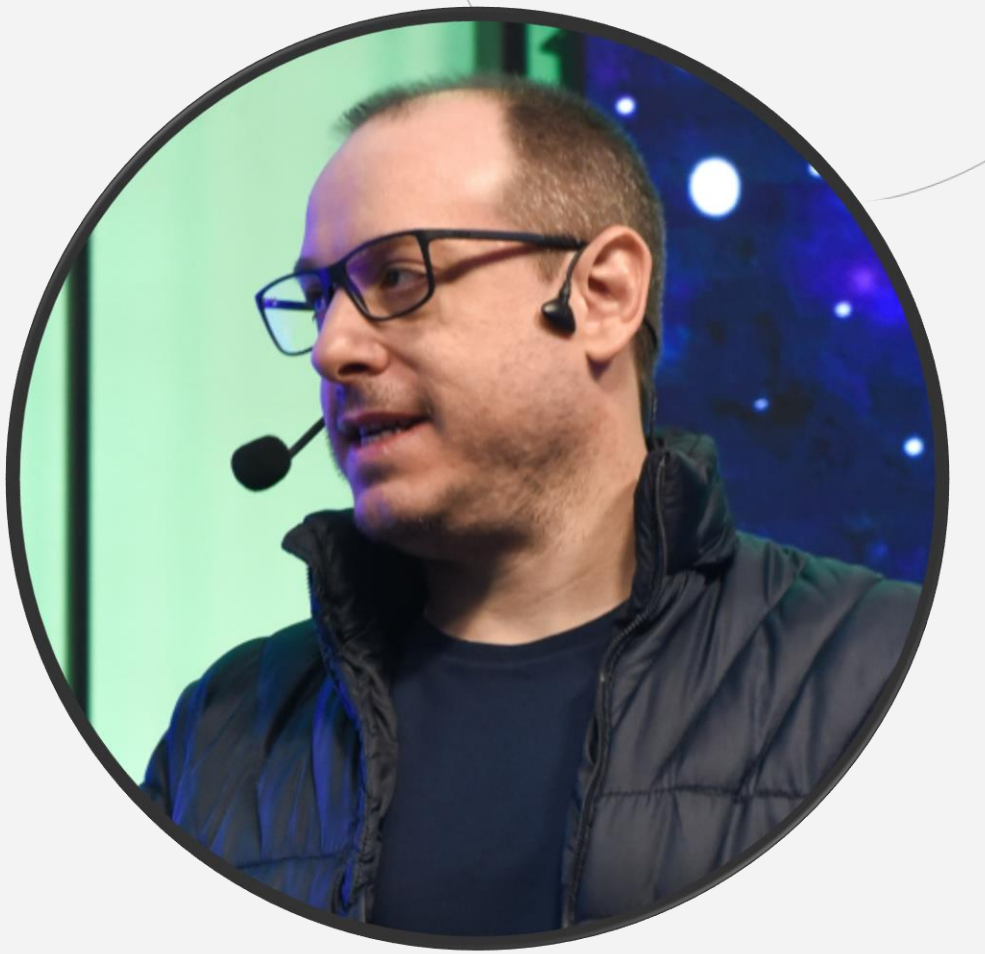
Injeção de Dependência

Imersão ASP.NET



André Baltieri

10x Microsoft MVP

Agenda

- O que é DI?
- O que é IoC?
- O que é DIP?
- Como os itens acima se relacionam
- DI no ASP.NET

Sobre este curso

- Devs ASP.NET/.NET
- Buscam aprimorar a teoria
- Querem conhecer mais DI

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Dependency Injection!");

app.Run();
```

Dependency Injection

Um termo bem confuso 💉

Dependency Injection

- **Não é um padrão** (Design Pattern)
- Técnica que **implementa** o **IoC**
  - Inversion of Controle (Inversão de Controle)
  - DIP
- Ajuda no baixo acoplamento
- Provê uma melhor divisão de responsabilidades
- O que eu preciso para trabalhar?
  - Quem vai me prover? Não importa

Baixo acoplamento

- Imagina um sistema **grande**
- Cada pedacinho tem que **focar em uma coisa**
  - **Não dá** pra abraçar o mundo
- Tem que funcionar de forma **independente**
  - Fácil de **entender**
  - Fácil de dar **manutenção**
  - Se precisar **jogar fora e criar outro** é fácil

Bad, bad, bad

- Vamos tomar como base um pedido
- Recebe os parâmetros
- Processa o pedido

```
public class OrderController : Controller
{
    [Route("v1/orders")]
    [HttpPost]
    public async Task<string> Place(
        string customerId,
        string zipCode,
        string promoCode,
        int[] products)
    {
        ...
    }
}
```

Bad, bad, bad

- Vamos tomar como base um pedido
- Recebe os parâmetros
- Processa o pedido

```
public class OrderController : Controller
{
    [Route("v1/orders")]
    [HttpPost]
    public async Task<string> Place(
        string customerId,
        string zipCode,
        string promoCode,
        int[] products)
    {
        // #1 - Recupera o cliente
        // #2 - Calcula o frete
        // #3 - Calcula o total dos produtos
        // #4 - Aplica o cupom de desconto
        // #5 - Gera o pedido
        // #6 - Calcula o total
        // #7 - Retorna
    }
}
```

Bad, bad, bad

```
// #1 - Recupera o cliente
Customer customer = null;
using (var conn = new SqlConnection("CONN_STRING"))
{
    customer = conn.Query<Customer>
        ("SELECT * FROM CUSTOMER WHERE ID=" + customerId)
        .FirstOrDefault();
}
```

Bad, bad, bad

```
// #2 - Calcula o frete
decimal deliveryFee = 0;
var request = new HttpRequestMessage(HttpMethod.Get, "URL/" + zipCode);
request.Headers.Add("Accept", "application/json");
request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

using (HttpClient client = new HttpClient())
{
    var response = await client.SendAsync(request);
    if (response.IsSuccessStatusCode)
    {
        deliveryFee = await response.Content.ReadAsAsync<decimal>();
    }
    else
    {
        // Caso não consiga obter a taxa de entrega o valor padrão é 5
        deliveryFee = 5;
    }
}
```

Bad, bad, bad

```
// #3 - Calcula o total dos produtos
decimal subTotal = 0;
for (int p = 0; p < products.Length; p++)
{
    var product = new Product();
    using (var conn = new SqlConnection("CONN_STRING"))
    {
        product = conn.Query<Product>
            ("SELECT * FROM PRODUCT WHERE ID=" + products[p])
            .FirstOrDefault();
    }
    subTotal += product.Price;
}
```

Bad, bad, bad

```
// #4 - Aplica o cupom de desconto
decimal discount = 0;
using (var conn = new SqlConnection("CONN_STRING"))
{

    var promo = conn.Query<PromoCode>
        ("SELECT * FROM PROMO_CODES WHERE CODE=" + promoCode)
        .FirstOrDefault();
    if (promo.ExpireDate > DateTime.Now)
    {
        discount = promo.Value;
    }
}
```

Bad, bad, bad

```
// #5 - Gera o pedido
var order = new Order();
order.Code = Guid.NewGuid().ToString().ToUpper().Substring(0, 8);
order.Date = DateTime.Now;
order.DeliveryFee = deliveryFee;
order.Discount = discount;
order.Products = products;
order.SubTotal = subTotal;

// #6 - Calcula o total
order.Total = subTotal - discount + deliveryFee;

// #7 - Retorna
return $"Pedido {order.Code} gerado com sucesso!";
```

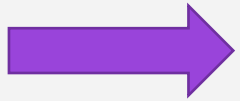
O problema

- **Dificil** de ler
- **Difícil** de mudar
- Código **não é reusável**
- **Alto acoplamento**
- **Testes**? Pra quê?

```
public class OrderController : Controller
{
    [Route("v1/orders")]
    [HttpPost]
    public async Task<string> Place(
        string customerId,
        string zipCode,
        string promoCode,
        int[] products)
    {
        // #1 - Recupera o cliente
        Customer customer = null;
        using (var conn = new SqlConnection("CONN_STRING"))
        {
            customer = conn.Query<Customer>
                ("SELECT * FROM CUSTOMER WHERE ID=" + customerId)
                .FirstOrDefault();
        }

        // #2 - Calcula o frete
        decimal deliveryFee = 0;
        var request = new HttpRequestMessage(HttpMethod.Get, "URL/" + zipCode);
        request.Headers.Add("Accept", "application/json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        using (HttpClient client = new HttpClient())
        {
            var response = await client.SendAsync(request);
            if (response.IsSuccessStatusCode)
            {
                deliveryFee = await response.Content.ReadAsAsync<decimal>();
            }
            else
            {
                // Caso não consiga obter a taxa de entrega o valor padrão é 5
                deliveryFee = 5;
            }
        }

        // #3 - Calcula o total dos produtos
        decimal subTotal = 0;
        for (int p = 0; p < products.Length; p++)
        {
            var product = new Product();
            using (var conn = new SqlConnection("CONN_STRING"))
            {
                product = conn.Query<Product>
                    ("SELECT * FROM PRODUCT WHERE ID=" + products[p])
                    .FirstOrDefault();
            }
            subTotal += product.Price;
        }

        // #4 - Aplica o cupom de desconto
        decimal discount = 0;
        using (var conn = new SqlConnection("CONN_STRING"))
        {

            var promo = conn.Query<PromoCode>
                ("SELECT * FROM PROMO_CODES WHERE CODE=" + promoCode)
                .FirstOrDefault();
            if (promo.ExpireDate > DateTime.Now)
            {
                discount = promo.Value;
            }
        }

        // #5 - Gera o pedido
        var order = new Order();
        order.Code = Guid.NewGuid().ToString().ToUpper().Substring(0, 8);
        order.Date = DateTime.Now;
        order.DeliveryFee = deliveryFee;
        order.Discount = discount;
        order.Products = products;
        order.SubTotal = subTotal;

        // #6 - Calcula o total
        order.Total = subTotal - discount + deliveryFee;

        // #7 - Retorna
        return $"Pedido {order.Code} gerado com sucesso!";
    }
}
```

Como resolvemos isto?

```
// #2 - Calcula o frete
decimal deliveryFee = 0;
var request = new HttpRequestMessage(HttpMethod.Get, "URL/" + zipCode);
request.Headers.Add("Accept", "application/json");
request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

using (HttpClient client = new HttpClient())
{
    var response = await client.SendAsync(request);
    if (response.IsSuccessStatusCode)
    {
        deliveryFee = await response.Content.ReadAsAsync<decimal>();
    }
    else
    {
        // Caso não consiga obter a taxa de entrega o valor padrão é 5
        deliveryFee = 5;
    }
}
```



```
public class DeliveryService {
    public decimal GetDeliveryFee(string zipCode) {
        var request = new HttpRequestMessage(HttpMethod.Get, "URL/" + zipCode);
        request.Headers.Add("Accept", "application/json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        using (HttpClient client = new HttpClient())
        {
            var response = await client.SendAsync(request);
            if (response.IsSuccessStatusCode)
            {
                deliveryFee = await response.Content.ReadAsAsync<decimal>();
            }
            else
            {
                deliveryFee = 5;
            }
        }
    }
}
```

Como resolvemos isto?

- **Orientação a Objetos**
  - **Abstração, encapsulamento**
    - **Simples** e direto
  - Pedaços **pequenos**
  - **Reusáveis**
  - **Testáveis**
  - **Legíveis**
  - **Fácil** manutenção

```
public class DeliveryService {
    public decimal GetDeliveryFee(string zipCode) {
        var request = new HttpRequestMessage(HttpMethod.Get, "URL/" + zipCode);
        request.Headers.Add("Accept", "application/json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        using (HttpClient client = new HttpClient())
        {
            var response = await client.SendAsync(request);
            if (response.IsSuccessStatusCode)
            {
                deliveryFee = await response.Content.ReadAsAsync<decimal>();
            }
            else
            {
                deliveryFee = 5;
            }
        }
    }
}
```

Como resolvemos isto?

- **Orientação a Objetos**
  - **Simples** e direto
  - Pedaços **pequenos**
  - **Reusáveis**
  - **Testáveis**
  - **Legíveis**
  - **Fácil** manutenção

```
public class OrderController : Controller
{
    [Route("v1/orders")]
    [HttpPost]
    public async Task<string> Place(
        string customerId,
        string zipCode,
        string promoCode,
        int[] products)
    {
        ...
        var deliveryService = new DeliveryService();
        decimal deliveryFee = deliveryService.GetDeliveryFee(zipCode);
        ...
    }
}
```

Cobre o pé... descobre a cabeça

- Está **bem melhor**, mas...
- A **dependência** ainda existe
  - Só mudou de lugar
- Depende da **implementação**
  - Depender da **abstração**

```
public class OrderController : Controller
{
    [Route("v1/orders")]
    [HttpPost]
    public async Task<string> Place(
        string customerId,
        string zipCode,
        string promoCode,
        int[] products)
    {
        ...
        var deliveryService = new DeliveryService();
        decimal deliveryFee = deliveryService.GetDeliveryFee(zipCode);
        ...
    }
}
```

Inversion of Control

- **Inversão de Controle**
- **Externaliza** as responsabilidades
  - **Delega**
- **Cria uma dependência** externa
  - O controller não é mais **responsável** pelo cáculo do frete, agora ele **depende de um serviço**

```
public class OrderController : Controller
{
    private readonly DeliveryService _deliveryService;

    OrderController(DeliveryService deliveryService) {
        _deliveryService = deliveryService;
    }

    [Route("v1/orders")]
    [HttpPost]
    public async Task<string> Place(
        string customerId,
        string zipCode,
        string promoCode,
        int[] products)
    {
        ...
        decimal deliveryFee = _deliveryService.GetDeliveryFee(zipCode);
        ...
    }
}
```

Inversion of Control

- **Inversão de Controle**
- **Externaliza** as responsabilidades
  - **Delega**
- **Cria uma dependência** externa
  - O controller não é mais **responsável** pelo cáculo do frete, agora ele **depende de um serviço**

```
public class OrderController : Controller
{
    [Route("v1/orders")]
    [HttpPost]
    public async Task<string> Place(
        string customerId,
        string zipCode,
        string promoCode,
        int[] products,
        [FromServices]DeliveryService deliveryService)
    {
        ...
        decimal deliveryFee = deliveryService.GetDeliveryFee(zipCode);
        ...
    }
}
```

Inversion of Control

- **Inversão de Controle**
- **Externaliza** as responsabilidades
  - **Delega**
- **Cria uma dependência** externa
  - O controller não é mais **responsável** pelo cáculo do frete, agora ele **depende de um serviço**

```
[TestMethod]
public void ShouldPlaceAnOrder() {
    var service = new DeliveryService();
    var controller = new OrderController(service);
    ...
}
```

Cobre o pé... descobre a cabeça

- Implementação
  - **Concreto**
  - **Materialização**
  - É o **“Como”**
- Abstração
  - **Contrato**
  - Só as **definições**
  - É o **“O que”**

```
public class DeliveryService {
    public decimal GetDeliveryFee(string zipCode) {
        var request = new HttpRequestMessage(HttpMethod.Get, "URL/" + zipCode);
        request.Headers.Add("Accept", "application/json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        using (HttpClient client = new HttpClient())
        {
            var response = await client.SendAsync(request);
            if (response.IsSuccessStatusCode)
            {
                deliveryFee = await response.Content.ReadAsAsync<decimal>();
            }
            else
            {
                deliveryFee = 5;
            }
        }
    }
}
```

Cobre o pé... descobre a cabeça

- Implementação
  - **Não varia**
  - **É uma visão, uma versão**
  - **Mais acoplado**
- Abstração
  - **Tem várias implementações**
  - **Menos acoplado**

```
public class DeliveryService {
    public decimal GetDeliveryFee(string zipCode) {
        var request = new HttpRequestMessage(HttpMethod.Get, "URL/" + zipCode);
        request.Headers.Add("Accept", "application/json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        using (HttpClient client = new HttpClient())
        {
            var response = await client.SendAsync(request);
            if (response.IsSuccessStatusCode)
            {
                deliveryFee = await response.Content.ReadAsAsync<decimal>();
            }
            else
            {
                deliveryFee = 5;
            }
        }
    }
}
```

Mas por que abstrair?

- **Facilita** as mudanças
  - Imagina um cenário crítico como a troca de um banco de dados..
- **Testes de Unidade**
  - Não podem depender de banco, rede ou qualquer outra coisa externa
- Se você depende da abstração, a **implementação não importa**...

```
public class OrderController : Controller
{
    [Route("v1/orders")]
    [HttpPost]
    public async Task<string> Place(
        string customerId,
        string zipCode,
        string promoCode,
        int[] products,
        [FromServices]DeliveryService deliveryService)
    {
        ...
        decimal deliveryFee = deliveryService.GetDeliveryFee(zipCode);
        ...
    }
}
```

Dependency Inversion Principle

- Princípio da **inversão de dependência**
- Depender de **abstrações** e não de **implementações**

```
public interface IDeliveryService {
    decimal GetDeliveryFee(string zipCode);
}
```

Dependency Inversion Principle

- Princípio da **inversão de dependência**
- Depender de **abstrações** e não de **implementações**

```
public class DeliveryService : IDeliveryService {
    public decimal GetDeliveryFee(string zipCode)
{
        ...
    }
}
```

Dependency Inversion Principle

- Princípio da **inversão de dependência**
- Depender de **abstrações** e não de **implementações**

```
public class OrderController : Controller
{
    private readonly IDeliveryService _deliveryService;

    OrderController(IDeliveryService deliveryService) {
        _deliveryService = deliveryService;
    }
    ...
}
```

Dependency Inversion Principle

- Princípio da **inversão de dependência**
- Depender de **abstrações** e não de **implementações**

```
public FakeDeliveryService : IDeliveryService {
    public decimal GetDeliveryFee(string zipCode) {
        return 10;
    }
}

[TestMethod]
public void ShouldPlaceAnOrder() {
    IDeliveryService service = new FakeDeliveryService();
    var controller = new OrderController(service);
    ...
}
```

Service Locator e DI no ASP.NET

- SL diz **como resolver** as dependências criadas
  - Funciona como um dê-para
- Já temos um pronto no **ASP.NET**
  - Podemos utilizar outros

```
// Assim
builder.Services.AddTransient<IDeliveryFeeService, DeliveryFeeService>();

// ou
builder.Services.AddScoped<IDeliveryFeeService, DeliveryFeeService>();

// ou
builder.Services.AddSingleton<IDeliveryFeeService, DeliveryFeeService>();
```

AddTransient

- Sempre cria uma **nova instância** do objeto
- Ideal para cenários onde queremos sempre **um novo objeto**

```
// Assim
builder.Services.AddTransient<IDeliveryFeeService, DeliveryFeeService>();

// ou
builder.Services.AddScoped<IDeliveryFeeService, DeliveryFeeService>();

// ou
builder.Services.AddSingleton<IDeliveryFeeService, DeliveryFeeService>();
```

AddScoped

- Cria **um objeto** por transação
- Se você chamar 2 ou mais serviços que dependem do **mesmo objeto**, a mesma instância será utilizada
- Ideal para cenários onde queremos **apenas um objeto** por requisição (Banco)

```
// Assim
builder.Services.AddTransient<IDeliveryFeeService, DeliveryFeeService>();

// ou
builder.Services.AddScoped<IDeliveryFeeService, DeliveryFeeService>();

// ou
builder.Services.AddSingleton<IDeliveryFeeService, DeliveryFeeService>();
```

Singleton

- Padrão que visa garantir **apenas um instância** de um objeto para **aplicação toda**
- Um bom exemplo são as **configurações**
  - Uma vez carregadas, **ficam até a aplicação reiniciar**

```
// Assim
builder.Services.AddTransient<IDeliveryFeeService, DeliveryFeeService>();

// ou
builder.Services.AddScoped<IDeliveryFeeService, DeliveryFeeService>();

// ou
builder.Services.AddSingleton<IDeliveryFeeService, DeliveryFeeService>();
```

AddSingleton

- Cria **um objeto** quando a aplicação inicia
- **Mantém este objeto** na memória até a aplicação para ou reiniciar
- Sempre devolve a **mesma instância** deste objeto, com os mesmos valores
- **CUIDADO**

```
// Assim
builder.Services.AddTransient<IDeliveryFeeService, DeliveryFeeService>();

// ou
builder.Services.AddScoped<IDeliveryFeeService, DeliveryFeeService>();

// ou
builder.Services.AddSingleton<IDeliveryFeeService, DeliveryFeeService>();
```

AddDbContext

- Item **especial** do tipo **Scoped**
- Utilizado exclusivamente com **Entity Framework**
- Garante que a conexão só dura **até o fim da requisição**

```
builder
    .Services
    .AddDbContext<BlogDataContext>(x ⇒ x.UseSqlServer(connStr));
```

Resumo

- DI
- IoC
- DIP
- Service Locator

DEMO #01

DIP, IOC e DI na prática 

Resolvendo a bagunça

```
builder.Services.AddScoped(new SqlConnection());
builder.Services.AddTransient<IProductRepository, ProductRepositoy>();
builder.Services.AddTransient<ICustomerRepository, CustomerRepository>();
builder.Services.AddTransient<IDiscountRepository, DiscountRepository>();
builder.Services.AddTransient<IOrderRepository, OrderRepository>();
builder.Services.AddTransient<IRoleRepository, RoleRepository>();
builder.Services.AddTransient<ICartRepository, CartRepository>();
```

Extension Methods

- Permitem **adicionar comportamentos** as classes built-in do .NET
- Como por exemplo o **WebApplicationBuilder.cs**
  - Mesmo se a classe for selada

```
public sealed class WebApplicationBuilder
{
    ...
    public IServiceCollection Services { get; }
    ...
}
```

Extension Methods

```
public static class DependenciesExtension
{
    public static void AddRepositories(this IServiceCollection services)
    {
        services.AddTransient<ICustomerRepository, CustomerRepository>();
        services.AddTransient<IPromoCodeRepository, PromoCodeRepository>();
        services.AddTransient<IDeliveryFeeService, DeliveryFeeService>();
    }

    public static void AddServices(this IServiceCollection services)
    {
        services.AddTransient<IDeliveryFeeService, DeliveryFeeService>();
    }
}
```

Extension Methods

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRepositories();
builder.Services.AddServices();

var app = builder.Build();
...
```

Outras formas de DI

```
public static void AddRepositories(this IServiceCollection services)
{
    services.AddTransient<CustomerRepository>();
    services.AddTransient(new CustomerRepository());
}
```

DEMO #02

Extension Methods na prática

DEMO #03

Lifetime na prática ⏰

Um pouco mais...

- Se podemos ter mais de uma implementação por interface...
- O que acontece quando registramos mais de um serviço?

```
public interface IService
{
}

public class ServiceOne : IService
{
}

public class ServiceTwo : IService
{
}
```

Um pouco mais...

```
builder.Services.AddTransient<IService, ServiceOne>();
builder.Services.AddTransient<IService, ServiceTwo>();
```

Sempre o último...

- Neste caso, como não especificamos a implementação, sempre será retornado a **última registrada**
- No exemplo seria o **ServiceTwo**

```
private readonly IService _service;

public OrderController(IService service)
    => _service = service;

[Route("/")]
[HttpGet]
public IActionResult Get()
{
    return Ok(new
    {
        _service.GetType().Name
    });
}
```

Inclusive pode isto aqui 👇

```
builder.Services.AddTransient<IService, ServiceOne>();
builder.Services.AddTransient<IService, ServiceOne>();
builder.Services.AddTransient<IService, ServiceOne>();
builder.Services.AddTransient<IService, ServiceTwo>();
```

Inclusive pode isto aqui 👇

```
private readonly IEnumerable<IService> _service;

public OrderController(IEnumerable<IService> service)
    => _service = service;

[Route("/")]
[HttpGet]
public IActionResult Get()
{
    return Ok(_service.Select(x => x.GetType().Name));
}
```

```
[
    "ServiceOne",
    "ServiceOne",
    "ServiceOne",
    "ServiceTwo"
]
```

Em resumo...

- Os serviços estão sendo registrados
- Porém o comportamento quando resolvemos um serviço é de **obter apenas o último**

```
private readonly IService _service;

public OrderController(IService service)
    => _service = service;

[Route("/")]
[HttpGet]
public IActionResult Get()
{
    return Ok(new
    {
        _service.GetType().Name
    });
}
```

Service Descriptor

- Descreve **como resolver** uma dependência
- Determina o **tipo** e **tempo de vida** dela
- *AddTransient*, *AddScoped* e *AddSingleton* são “wrapers” deste item

Service Descriptor

```
var descriptor = new ServiceDescriptor(
    typeof(IService), // Abstração
    typeof(ServiceOne), // Implementação
    ServiceLifetime.Singleton); // Tempo de vida

builder.Services.Add(descriptor);
```

TryAdd*

- Inverte o comportamento
- Não dá erro, mas não duplica
- Compara apenas a abstração
  - Não registra duas implementações para uma mesma abstração (Interface)

TryAdd*

```
builder.Services.TryAddTransient<IService, ServiceOne>();
builder.Services.TryAddTransient<IService, ServiceOne>();
builder.Services.TryAddTransient<IService, ServiceOne>();
builder.Services.TryAddTransient<IService, ServiceTwo>();
```

TryAdd*

- Só vai registrar o **primeiro** item
- Como já existe uma implementação registrada para a interface **IService**, vai ignorar as próximas tentativas de registro.

TryAddEnumerable

- TryAddEnumerable
- Permite registrar ambos (1 e 2)
- Porém não permite duplicar (2 e 2 por exemplo)
- Único (Interface e implementação).

DEMO #04

Registrando múltiplas implementações 

Resolvendo Dependências

- Construtor
- Na assinatura do método
- No program
- No HttpContext

No constructor...

- Private Readonly?
  - Qual a diferença de const?

```
private readonly IWeatherService _service;

public WeatherController(IWeatherService service)
    => _service = service;

[HttpGet("/")]
public IEnumerable<WeatherForecast> Get()
    => _service.Get();
```

FromServices

- Obtém direto dos serviços
- No .NET 7 não precisa mais especificar **[FromServices]**

```
[HttpGet("/")]
public IEnumerable<WeatherForecast> Get(
    [FromServices]IWeatherService service)
    => service.Get();
```

FromServices

- Obtém direto dos serviços
- No .NET 7 não precisa mais especificar **`[FromServices]`**

```
[HttpGet("/")]
public IEnumerable<WeatherForecast> Get(
    IWeatherService service)
    => service.Get();
```

No Program.cs

```
var app = builder.Build();
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    var repository = services.GetRequiredService<ICustomerRepository>();
    repository.CreateAsync(new Customer("André Baltieri"));
}
```

Via HttpContext

- Podemos “recuperar” os serviços registrados utilizando o **HttpContext**

```
public async Task OnActionExecutionAsync(
        ActionExecutingContext context,
        ActionExecutionDelegate next)
{
    var service = context
      .HttpContext
      .RequestServices
      .GetService<IWeatherService>();

    var forecasts = service.Get();
```

DEMO #05

Utilizando serviços fora dos Controllers 

ENTREVISTA

Hora de responder as perguntas

Hora da entrevista...

“Qual a diferença entre AddTransient, AddScoped e AddSingleton?”

Hora da entrevista...

“Qual a finalidade do atributo FromServices?”

Hora da entrevista...

“Podemos resolver dependências fora dos controladores?”

Hora da entrevista...

"De forma resumida, você consegue me dizer o que é injeção de dependência?"

Hora da entrevista...

“O que é Inversão de Controle?”

Hora da entrevista...

“O que é Inversão de Dependência?”

Hora da entrevista...

“Qual a relação entre injeção de dependência, inversão de controle e inversão de dependência?”

Sugestão de projeto

- Reserva de quarto
  - Utilizar AddTransient, Scoped e Singleton
  - Separar em serviços e repositórios
  - Postar o resultado no repositório do GitHub do curso