FMX Academy

INFORMAÇÕES		
MÓDULO	AULA	INSTRUTOR
06 ASP.NET Core Web API – Avançado e Segurança Inicial	05 - Padronizando Ações com Commands e Mensageria	Guilherme Paracatu

1. O "Service Monstro" e o Alto Acoplamento

1. Até agora, nossas APIs seguem um fluxo linear:
Controller ➡ Service ➡ Repository.

Embora funcione, esse modelo gera um problema chamado **Alto Acoplamento.** Se o FabricanteService precisar realizar 5 tarefas (Salvar no banco, Logar, Enviar E-mail, Validar Fraude e Atualizar Cache), ele precisará de 5 dependências no seu construtor.

- **Consequência:** Se o sistema de Log mudar, o Service quebra. Se o Service mudar, o Controller quebra. É um efeito dominó que dificulta a manutenção.

2. A Solução: Padrão Mediator (O Mediador)
O padrão **Mediator** introduz um intermediário no sistema. O Controller não chama mais uma "ação" de um Service; ele envia uma **Mensagem (Command)** para um **Barramento (Bus).**

- **Controller:** Não sabe quem executa a tarefa.
- **Handler (Executor):** Não sabe quem pediu a tarefa.
- **Analogia:** É como uma Torre de Controle de aeroporto. O piloto (Controller) não fala com outros aviões; ele fala com a Torre (Bus), que coordena tudo.

Anatomia de uma Operação com Mediator
Para que o desacoplamento funcione, dividimos uma única tarefa em **três responsabilidades distintas**:

Peça	Responsabilidade	Exemplo Prático
Command	**O QUE** deve ser feito? (Dados)	"Nome: Coca-Cola, Preço: 5.00"
Handler	**COMO** deve ser feito? (Lógica)	"Abrir banco -> Validar -> Salvar Produto"
Bus	**QUEM** deve fazer? (Roteamento)	"Procurar quem resolve o pedido de Produto"

3. Entendendo as Abstrações (Domain.Core)
Para suportar essa arquitetura, criamos a camada **ITB.Domain.Core.** Ela contém as "regras de ouro" que não mudam, independente do negócio.

Ela funciona como o "Manual de Regras" do sistema. Nela, definimos **Abstrações** (Interfaces) que servem para qualquer projeto.

O uso de Generics e Constraints (where T)
Nas nossas interfaces, você verá algo como: IHandler<T> where T : ICommand.

- **O que é o <T>?** É um tipo genérico. Permite que a interface se adapte a qualquer Comando.

- **O que é o where T : ICommand?** É uma **restrição de tipo**. Ela garante que o sistema só aceite objetos que sejam tecnicamente "Comandos" (que possuam a etiqueta ICommand). Isso evita que enviemos dados errados para o Barramento.

Tradução para o Português: "Aceite qualquer tipo **T**, **DESDE QUE** esse **T** implemente a interface **ICommand**".

Por que usar?
1. **Segurança:** Impede que o desenvolvedor cometa o erro de enviar algo que não é um comando.
2. **IntelliSense:** O Visual Studio passa a entender que T é um comando, liberando sugestões de propriedades específicas dessa interface (se houver).
3. **Organização:** Garante que o nosso "Ônibus" só transporte passageiros autorizados (objetos que carregam a etiqueta ICommand).

2. Implementação Prática: Módulo de Fabricantes

Implementação Prática: Módulo de Fabricantes

🚀 **Passo 1: Criar o Projeto Físico**

Rode este comando na raiz da pasta ITB:

dotnet new classlib -n ITB.Domain.Core -o ITB.Domain.Core

🗁 **Passo 2: Criar a Solution Folder "7 - Domain.Core"**
Como essa pasta ainda não existe no seu arquivo .sln, primeiro a criamos virtualmente:

dotnet sln add ITB.Domain.Core/ITB.Domain.Core.csproj --solution-folder "7 - Domain.Core"

Note que ao usar --solution-folder, o dotnet já entende que deve criar o agrupamento se ele não existir.

🔗 **Passo 3: Referenciar o Core nos outros projetos**

Agora, precisamos "ligar os fios". O Core é a base, então quase todos vão precisar dele. Rode na sequência:

 # Adiciona o Core na Domain

dotnet add ITB.Domain/ITB.Domain.csproj reference ITB.Domain.Core/ITB.Domain.Core.csproj

 # Adiciona o Core na Application

dotnet add ITB.Application/ITB.Application.csproj reference ITB.Domain.Core/ITB.Domain.Core.csproj

 # Adiciona o Core na Infra (para o Bus funcionar)

dotnet add ITB.Infrastructure/ITB.Infrastructure.csproj reference ITB.Domain.Core/ITB.Domain.Core.csproj

A) As Interfaces Base (Domain.Core)
Local: ITB.Domain.Core/Messages

Código C#

```
// Etiqueta para identificar o que é um pedido
public interface ICommand { }

// Contrato para o especialista (Handler) que resolve o pedido
public interface IHandler<T> where T : ICommand
{
    Task Handle(T comando);
}

// Contrato para o motor de transporte (Bus)
public interface IMessageBus
{
    Task EnviarComando<T>(T comando) where T : ICommand;
}
```

B) O Pedido: CriarFabricanteCommand (Application)
Local: ITB.Application/Commands
O Command é uma classe simples que carrega apenas os dados necessários.

Código C#

```
public class CriarFabricanteCommand : ICommand
{
    public string Nome { get; set; } = string.Empty;
    public string Cnpj { get; set; } = string.Empty;
}
```

C) O Especialista: CriarFabricanteHandler (Application)
Local: ITB.Application/Handlers

Focado exclusivamente em salvar o fabricante no banco de dados.

Código C#

```
public class CriarFabricanteHandler : IHandler<CriarFabricanteCommand>
{
    private readonly IFabricanteRepository _repository;
    public CriarFabricanteHandler(IFabricanteRepository repository) =>
_repository = repository;

    public async Task Handle(CriarFabricanteCommand comando)
    {
        var fabricante = new Fabricante { Nome = comando.Nome, Cnpj =
comando.Cnpj };
        await _repository.Adicionar(fabricante);
    }
}
```

5. Demonstração de Poder: O Handler de Log
Local: ITB.Application/Handlers

O grande trunfo do Mediator é adicionar funcionalidades **sem mexer no código que já existe**. Vamos criar um segundo especialista para o mesmo comando.

Código C#

```
public class LogFabricanteHandler : IHandler<CriarFabricanteCommand>
{
    public Task Handle(CriarFabricanteCommand comando)
    {
```

```
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine($"[AUDITORIA] - Fabricante {comando.Nome} está sendo
processado...");
        Console.ResetColor();
        return Task.CompletedTask;
    }
}
```

6. O Motor: InMemoryBus (Infrastructure)
Local: ITB.Infrastructure/Bus

Este componente percorre a memória da aplicação, localiza todos os especialistas (Handlers) registrados para aquele comando e os executa.

Instalar o pacote de Injeção de Dependência no projeto onde o InMemoryBus está (geralmente na **Infrastructure**). Rode no terminal:

```
dotnet add ITB.Infrastructure package
Microsoft.Extensions.DependencyInjection.Abstractions
```

Código C#

```
public sealed class InMemoryBus : IMessageBus
{
    private readonly IServiceProvider _serviceProvider;
    public InMemoryBus(IServiceProvider serviceProvider) => _serviceProvider =
serviceProvider;

    public async Task EnviarComando<T>(T comando) where T : ICommand
    {
        // Localiza todos os especialistas registrados para o comando T
        var handlers = _serviceProvider.GetServices<IHandler<T>>();

        foreach (var handler in handlers)
            await handler.Handle(comando);
    }
}
```

7. Registro de Dependências (IoC)
Local: ITB.IoC/DependencyInjection.cs

Para o .NET "ligar os fios", precisamos registrar no DependencyInjection.cs:

Código C#

```
services.AddScoped<IMessageBus, InMemoryBus>();

// Registramos múltiplos Handlers para o mesmo comando!
services.AddScoped<IHandler<CriarFabricanteCommand>, CriarFabricanteHandler>();
services.AddScoped<IHandler<CriarFabricanteCommand>, LogFabricanteHandler>();
```

8. Conclusão: O Novo Controller
Note como o Controller se torna "imune" ao crescimento do sistema. Ele não precisa saber se existem 1, 2 ou 10 tarefas sendo executadas por trás.
Código C#

```
[HttpPost]
public async Task<IActionResult> Post([FromBody] CriarFabricanteCommand
command)
{
    // O Controller apenas despacha a mensagem. O resto é com o Bus!
    await _bus.EnviarComando(command);
    return Ok();
}
```

Glossário Rápido para Revisão

1. **Command:** O "O Quê" (Dados).
2. **Handler:** O "Como" (Lógica).
3. **Bus:** O "Quem" (Transporte).
4. **InMemory:** Ocorre dentro da memória RAM da aplicação.
5. **SRP (Princípio da Responsabilidade Única):** Cada classe (Handler) faz apenas uma coisa.

Reflexões sobre o novo padrão

Responsabilidade Única (SRP): "Vejam que o CriarFabricanteHandler só cuida de banco de dados. Ele nem sabe que o Log existe. Isso deixa o código limpo e fácil de testar."

Extensibilidade: "Se amanhã o cliente pedir para enviar um E-mail também, eu não altero o código que já funciona. Eu apenas crio um EnviarEmailHandler, registro no IoC e o sistema passa a enviar e-mails magicamente."

O Controller continua intacto: "Olhem para o Controller. Ele continua chamando apenas _bus.EnviarComando(command). Ele não faz ideia de que agora o sistema salva no banco E faz um log. Para ele, a única coisa que importa é que o pedido foi enviado."

Desafio em Aula: Implemente o DeletarFabricanteCommand e crie um Handler que, além de deletar, exiba um alerta de segurança no console.