

Desafio Umblar:

Refatoração & Arquitetura

O Cenário Original

Mapeamento e identificação dos gargalos críticos encontrados na estrutura do código legado e na infraestrutura.

Principais Dores Encontradas



Alto

Aceplamento

O exemplo insinuava bibliotecas externas (LookupClient/WhoisClient) diretamente, inviabilizando qualquer teste isolado.



Bloqueio de

Infra

Tentativas de acesso ao MySQL da Umbler resultavam em Timeout constante devido a bloqueios de rede na porta 3306.



Vazamento de Dados

A API retornava a Entidade de domínio inteira diretamente para o client, expondo chaves primárias e tempos de cache (TTL).



UX

Precária

A interface do usuário exibia o retorno da consulta Whols como um bloco de texto JSON cru e desestruturado na tela.

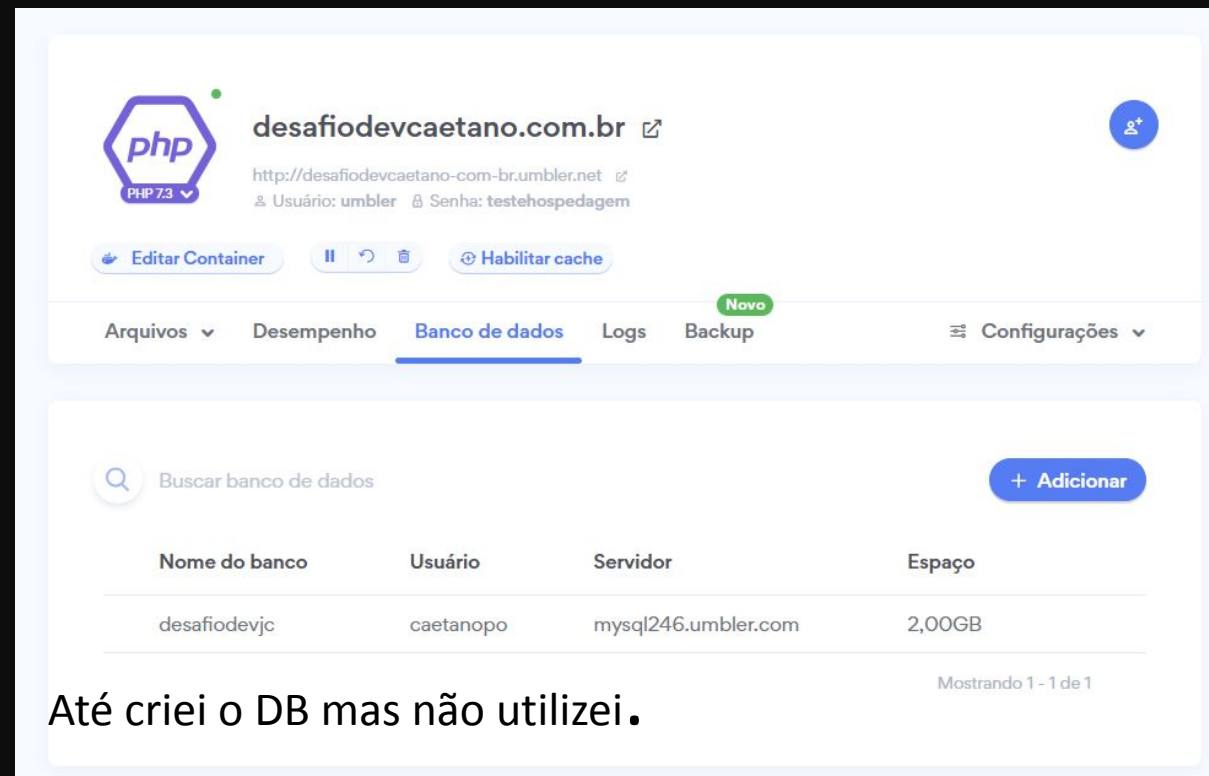
Soluções Arquiteturais

Aplicando princípios S.O.L.I.D, Clean Code e
Padrões de Projeto robustos na camada de Back-end.

O Desafio da Infraestrutura

A transição pragmática de banco de dados para destravar o ambiente de desenvolvimento local.

- **O Problema:** O firewall local impedia o acesso ao banco remoto, causando quebras na execução do Entity Framework.
- **A Decisão:** Substituição da dependência do MySQL pelo provedor Microsoft.EntityFrameworkCore.SqlServer.
- **O Resultado:** O projeto rodou de forma autossuficiente e isolada. Essa mudança simples atestou o excelente poder de abstração do ORM sobre o domínio.



Até criei o DB mas não utilizei.

```
0 referências
public void ConfigureServices(IServiceCollection services)
{
    var connectionString = Configuration.GetConnectionString("DefaultConnection");

    services.AddSwaggerGen();
    services.AddScoped<IWhoisService, WhoisService>();
    services.AddScoped<IDnsService, DnsService>();

    services.AddDbContext<DatabaseContext>(options =>
    {
        options.UseSqlServer(connectionString)
            .LogTo(Console.WriteLine, LogLevel.Information)
            .EnableSensitiveDataLogging()
            .EnableDetailedErrors();
    });

    services.AddControllersWithViews();
}
```

Injeção de Dependência

Quebrando as amarras do Controller e descentralizando as responsabilidades através de interfaces.

Extraímos as regras rígidas de comunicação externa para duas novas abstrações: **IDnsService** e **IWhoisService**.

Agora, o DomainController não precisa saber "como" buscar na internet, apenas "o que" solicitar. As instâncias são entregues limpas e prontas pelo Container de DI nativo do .NET 6 no Startup.cs.

```
using Desafio.Umbler.DTOs;  
using Desafio.Umbler.Interfaces;  
using Desafio.Umbler.Models;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.EntityFrameworkCore;  
using System;  
using System.Threading.Tasks;
```

```
namespace Desafio.Umbler.Controllers  
{  
    [Route("api")]  
    public class DomainController : Controller  
    {  
        private readonly DatabaseContext _db;  
        private readonly IWhoisService _whoisService;  
        private readonly IDnsService _dnsService;  
  
        public DomainController(DatabaseContext db, IWhoisService whoisService, IDnsService dnsService)  
        {  
            _db = db;  
            _whoisService = whoisService;  
            _dnsService = dnsService;  
        }  
    }  
}
```

```
[HttpGet, Route("domain/{domainName}")]  
public async Task<IActionResult> Get(string domainName)  
{  
    if (string.IsNullOrEmpty(domainName))  
        return BadRequest("O nome do domínio é obrigatório.");  
  
    var domain = await _db.Domains.FirstOrDefaultAsync(d => d.Name == domainName);  
  
    if (domain == null || DateTime.Now.Subtract(domain.UpdatedAt).TotalMinutes > domain.Ttl)  
    {  
        domain ??= new Domain();  
  
        var (ip, ttl) = await _dnsService.GetIpAndTtlAsync(domainName);  
        var whoisRaw = await _whoisService.GetWhoisRawDataAsync(domainName);  
        var hostedAt = await _whoisService.GetHostedAtAsync(ip);  
    }  
}
```

Proteção e Padrão DTO

- **Riscos no Legado:** O retorno original Ok(domain) enviava o modelo de dados integral para o navegador. Campos vitais de infraestrutura (Id do registro, Ttl de cache) viajavam expostos na rede.
- **Solução:** Implementação do Padrão Data Transfer Object (DTO).
- **Ganhos Imediatos:** A classe DomainResponseDTO foi criada para mapear apenas o Nome, IP, Whois e Hospedagem. O Payload de rede ficou muito mais leve e seguro, blindando a entidade central da aplicação.

```
namespace Desafio.Umbler.DTOs
{
    3 referências
    public class DomainResponseDTO
    {
        3 referências | 2/2 passing
        public string Name { get; set; }
        3 referências | 2/2 passing
        public string Ip { get; set; }
        2 referências | 1/1 passing
        public string HostedAt { get; set; }
        1 referência
        public string WhoIs { get; set; }
    }
}
```

100%

Testabilidade Restaurada

Suíte de Testes Confiável

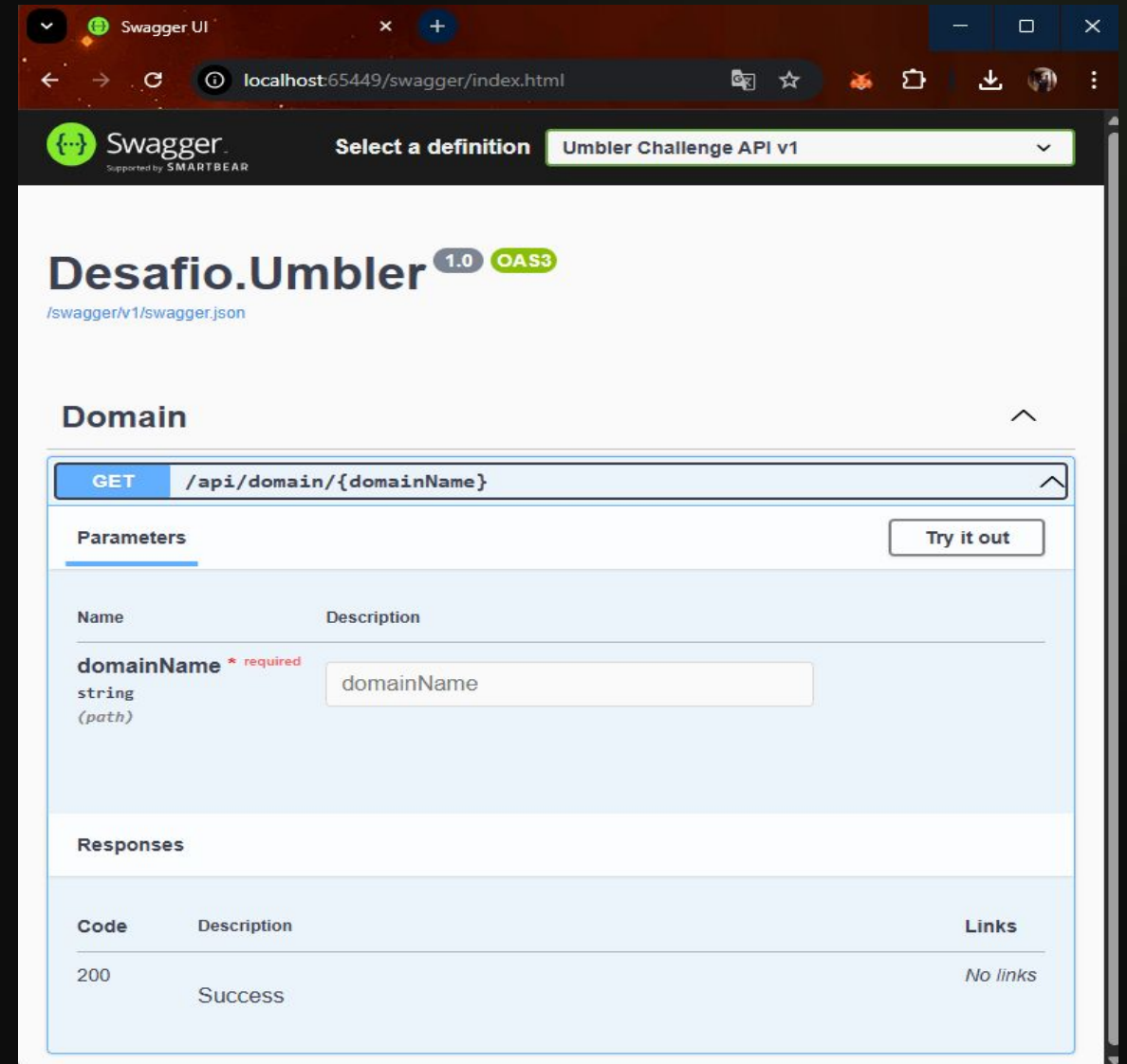
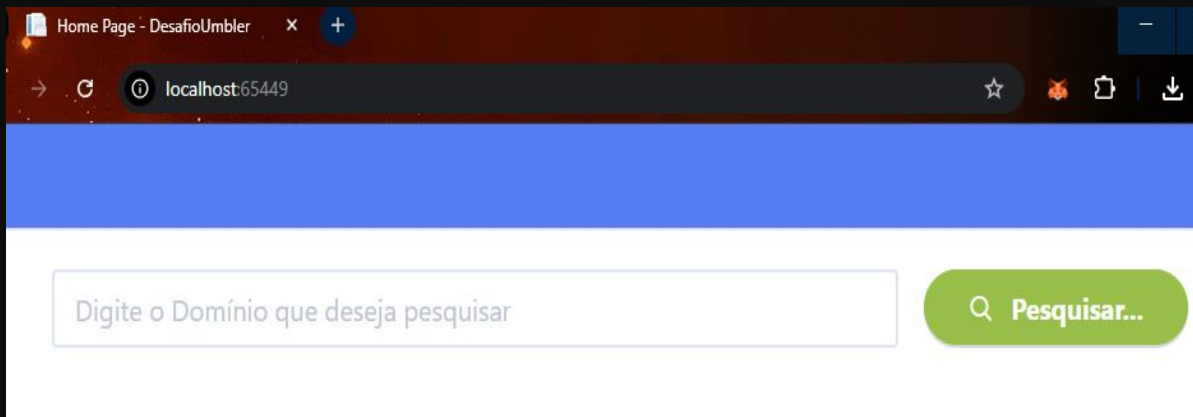
Os testes originais falhavam sem acesso à internet. Com a nova arquitetura desacoplada baseada em interfaces, reescrevemos o cenário no `ControllersTests.cs`.

Utilizando a biblioteca **Moq** e o Banco em Memória, simulamos cenários de sucesso e falha instantaneamente, sem gerar requisições reais ou depender de terceiros estarem no ar.

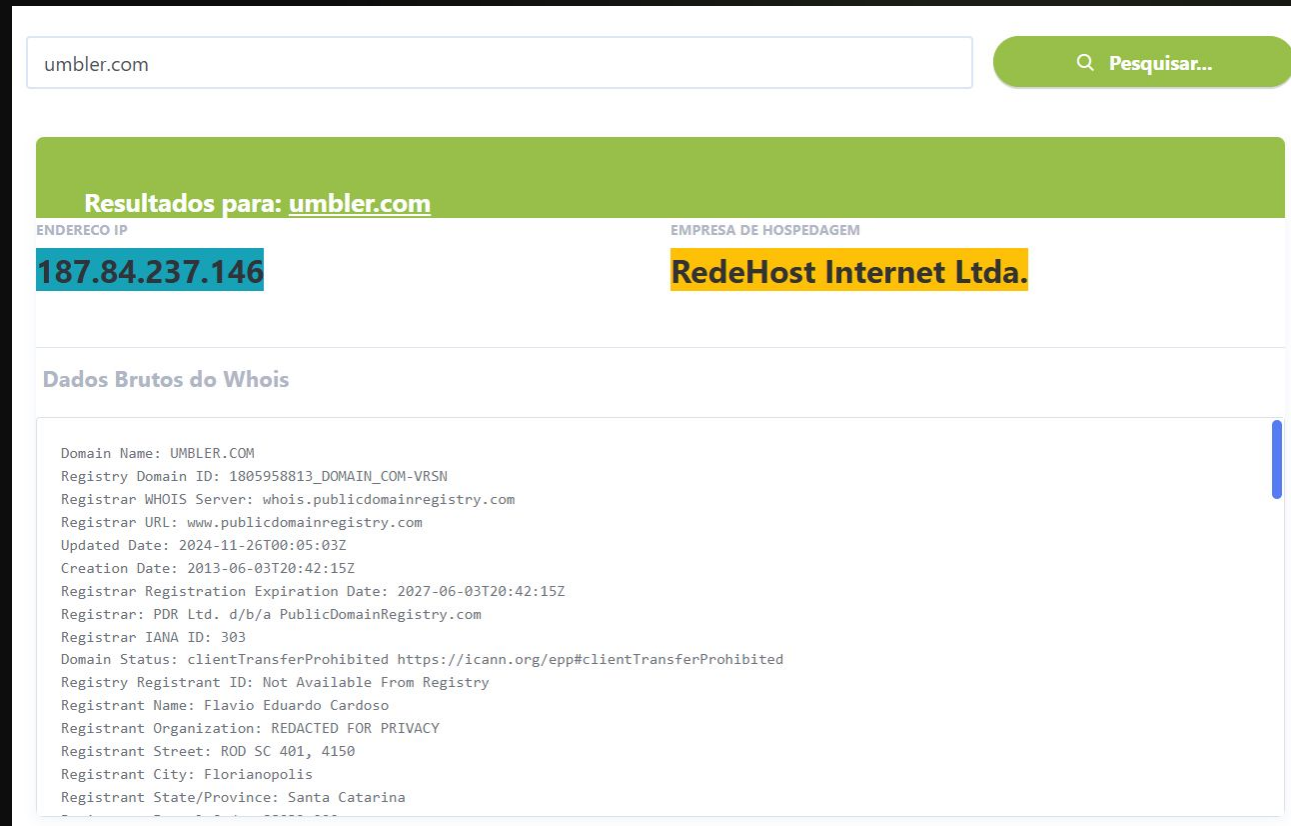
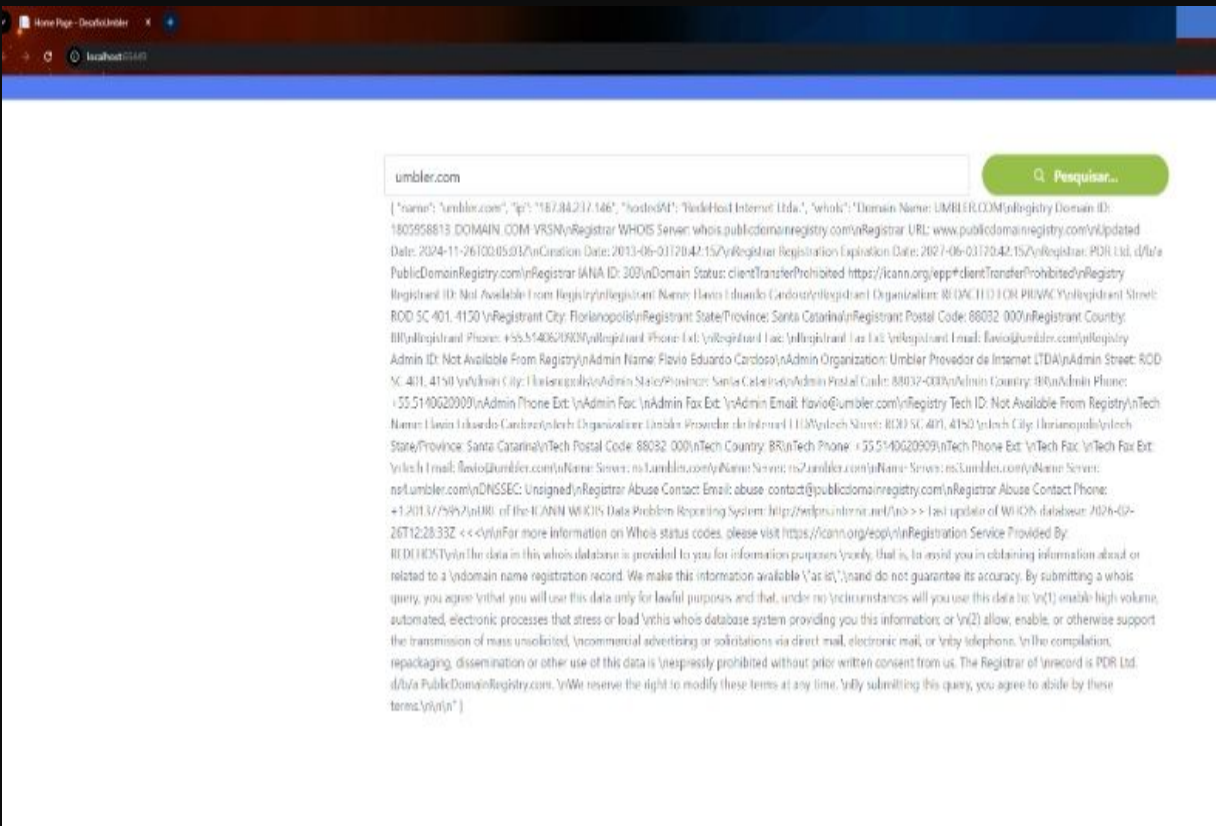
Governança com Swagger

Foi adicionado o pacote Swashbuckle.AspNetCore no Startup para gerar uma documentação automática (OpenAPI).

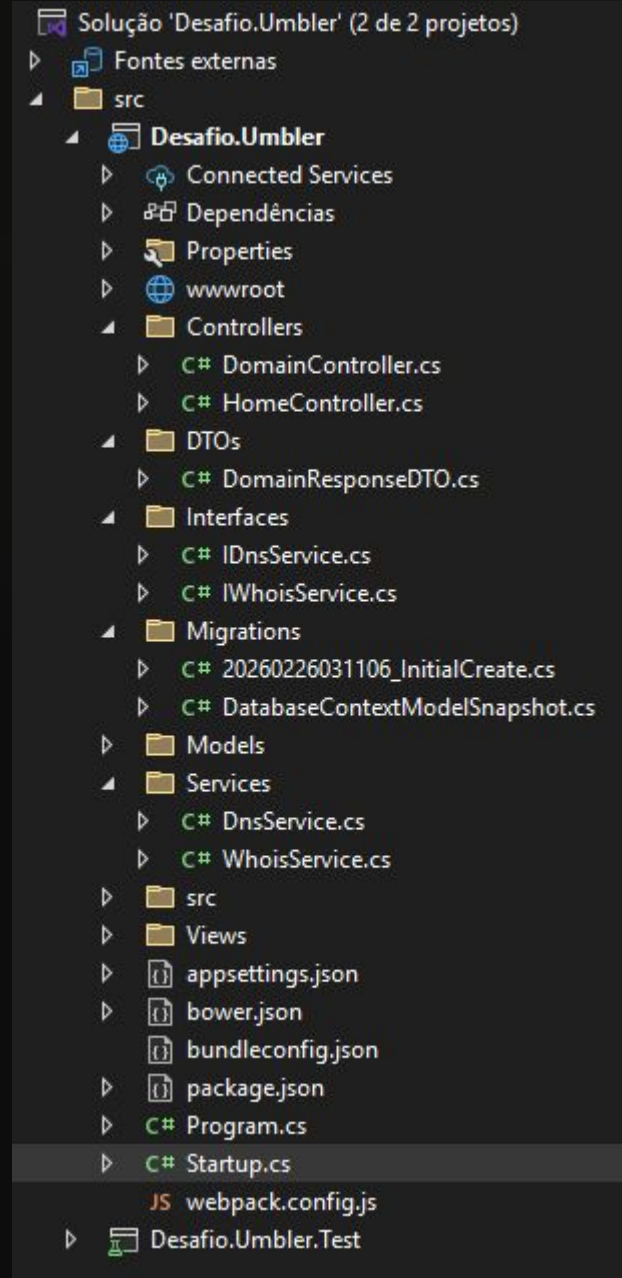
Isso permite que times de front-end testem os endpoints, visualizem o formato exato do DomainResponseDTO e compreendam os códigos de erro, tudo por meio de uma interface interativa.



○ Antes e o Depois.



A estrutura do Projeto.



Pronto para Produção.

A arquitetura foi estabilizada, refatorada e o débito técnico zerado.

Muito obrigado pela oportunidade!
