ARQUITETURA .NET: CLEAN ARCHITECTURE

**DOWNLOADS NECESSÁRIOS**

Visual Studio 2022 (.net sdk 8)

Projeto Inicial - <https://github.com/alura-cursos/ContainRs/archive/refs/heads/master.zip>

**CONFIGURAÇÕES INICIAIS**

Após baixar a pasta do projeto, abri-la pelo CMD e aplicar as *migrations* do projeto baixado

dotnet ef database update --project .\ContainRs.WebApp\ContainRs.WebApp.csproj --startup-project .\ContainRs.WebApp\ContainRs.WebApp.csproj

**MACETES**

Gerar construtor a partir do atributo

Selecionar a linha de um atributo de classe -> CTRL + .

Remover bibliotecas desnecessárias

Selecionar a linha cuja biblioteca não está sendo usada-> CTRL + .

*Remover usos desnecessários*

Renomear todas as ocorrências de uma variável pelo cabeçalho da função

Selecionar o parâmetro -> CTRL + RR

**IMPLEMENTANDO REGRA - SISTEMA NÃO REGISTRA MENOR DE IDADE**

Adicionar campo nascimento em Views/Registro/Index.cshtml

Abaixo da tag div Nome

<div class="form-group col-3 mt-2">

<**label** class="form-label" **asp-for**="Nascimento"></**label**>

<**input** class="form-control" **asp-for**="Nascimento" />

<**span** class="text-danger small" **asp-validation-for**="Nascimento"></**span**>

</div>

Adicionar propriedade Nascimento em Models/RegistroViewModel

Abaixo da propriedade Nome

[Display(Name = "Nascimento (\*)")]

[Required(ErrorMessage = "Campo obrigatório.")]

[DataType(DataType.Date)]

public DateTime Nascimento { get; set; }

Inserir a lógica de negar o registro caso a idade seja < 18 no método CreateAsync em Controlleres/RegistroController

var idade = DateTime.Today.Year - form.Nascimento.Year;

if (idade < 18)

{

ModelState.AddModelError("Nascimento", "Obrigatório ter mais de 18 anos.");

return View("Index", form);

}

**TEORIA - MVC**

O padrão MVC (Model-View-Controller) é uma das arquiteturas de software mais populares, especialmente no desenvolvimento de aplicações web. Ele foi introduzido como uma forma de separar responsabilidades dentro de uma aplicação, permitindo que desenvolvedores organizem código de forma modular e mantenham uma clara distinção entre a lógica de negócios, a apresentação e o controle das ações do usuário.

O MVC foi inicialmente introduzido por Trygve Reenskaug em 1978, e sua principal motivação foi facilitar o desenvolvimento e a manutenção de sistemas complexos, garantindo maior reutilização de código e melhor separação de interesses.

Dois fatores principais impulsionaram a popularização do MVC a partir dos anos 90 e 2000 foram a adoção abrangente da programação orientada a objetos e o aumento da utilização de aplicações Web em novos projetos.

Frameworks como Spring MVC, Ruby on Rails e ASP.NET MVC incorporaram o modelo de arquitetura em suas abordagens, fazendo com que o padrão fosse uma escolha comum na construção de aplicações web, ajudando desenvolvedores a estruturar seus projetos de forma mais organizada e escalável.

Os Controllers são a peça central do padrão MVC e têm a responsabilidade de gerenciar a interação do usuário. Eles recebem as entradas do usuário — sejam cliques, envios de formulários ou qualquer outra ação —, processam essas entradas, e então interagem com os Models ou Views de acordo. Além disso, o Controller é responsável por tomar decisões sobre o que deve ser exibido ao usuário e qual lógica de negócios deve ser executada, funcionando como o intermediário que conecta as diferentes partes da aplicação.

Os Models representam a lógica de negócios e os dados da aplicação. Eles são responsáveis por manipular, validar, e armazenar informações, normalmente acessando um banco de dados ou outra fonte de dados. Em uma aplicação MVC, os Models são o componente que contém a lógica necessária para tratar os dados e aplicar as regras de negócio, garantindo que a aplicação funcione de acordo com os requisitos. Essa separação permite que o núcleo da lógica de negócios seja independente da forma como os dados são apresentados ao usuário.

As Views são a camada responsável pela apresentação dos dados ao usuário. Elas são diretamente conectadas à interface do usuário, exibindo os dados processados pelos Models de acordo com as decisões tomadas pelo Controller. Uma View pode ser composta por HTML, CSS e JavaScript no caso de aplicações web, mas seu papel é sempre o de simplesmente mostrar as informações da forma mais clara possível, sem conter lógica de negócios ou processamentos complexos.

**VALIDAÇÃO DO E-MAIL**

Para validar o e-mail, que é uma questão de regras de negócio, primeiro cria-se a classe Email com a lógica de validação do campo Value (que vai guardar o conteúdo do email) em Models/Email

public class Email

{

//PROPRIEDADES

//Value - que é a string para o e-mail

public string Value { get;}

//EXPRESSÃO REGULAR PARA LÓGICA DE VALIDAÇÃO DO E-MAIL

private static readonly Regex EmailRegex = new Regex(

@"^[^@\s]+@[^@\s]+**\.**[^@\s]+$",

RegexOptions.Compiled | RegexOptions.IgnoreCase);

//CONSTRUTOR QUE JÁ FAZ A VALIDAÇÃO DO E-MAIL

public Email(string value)

{

if (!EmailRegex.IsMatch(value))

{

throw new ArgumentException("E-mail inválido.");

}

Value = value;

}

}

No momento de registrar o cliente, mudar a lógica para usar a classe Email, em Controllers/RegistroController, no método CreatAsync

var cliente = new Cliente(form.Nome, new Email(form.Email), form.CPF) ...

Em Models/Cliente mudar as propriedades e construtor para usar a classe Email

//LIMITAÇÃO DO ENTITY PARA USAR CLASSE Email NO CONSTRUTOR DE Cliente

private Cliente() { }

public Cliente(string nome, Email email, string cPF)

{

Nome = nome;

Email = email;

CPF = cPF;

}

public Guid Id { get; set; }

public string Nome { get; private set; }

public Email Email { get; private set; } ...

Configurar Data/AppDbContext para aceitar o campo Email de Cliente como um objeto do tipo Email, no método OnModelCreating

//ESTABELECENDO AS CONFIGURAÇÕES DO CAMPO Email DE Cliente

//PARA A CONVERSÃO ENTRE MODELS <-> DB

modelBuilder.Entity<Cliente>()

.OwnsOne(c => c.Email, cfg =>//1 EMAIL

{

cfg.Property(e => e.Value)//O CAMPO Value EM EMAIL

.HasColumnName("Email")//COM NOME DE COLUNA Email

.IsRequired();//CAMPO Email OBRIGATÓRIO

});

Agora, sempre que um e-mail é inserido, e ele não obedece às regras da expressão regular configurada, é lançado um erro na tela. *(não seria melhor ensinar o formato de e-mail correto?)*

**TEORIA - ENTIDADES, VALUE OBJECTS E AGREGADOS**

Na camada de Domínio, empregamos com frequência padrões de projeto como Entidades, ValueObjects e Agregados.

-Entidades representam objetos com identidade própria e ciclo de vida independente. Em nosso projeto, um Cliente é uma entidade.

-ValueObjects simbolizam conceitos que existem somente a partir de outros tipos, sendo, portanto, dependentes deles. O Email, que criamos para ilustrar um conceito de negócio importante para a ContainRs, existe somente a partir de um cliente. É, portanto, um exemplo de ValueObject.

-Agregados mantêm a integridade de um grupo de objetos relacionados a partir de um ponto-raiz que permite o acesso consistente aos dados deste agrupamento. Não temos um exemplo de agregado ainda no projeto ContainRs, mas um exemplo seria uma NotaFiscal, que relaciona seus itens de forma bastante coesa.

**TESTE AUTOMATIZADO DE EMAIL COM xUnit Test**

Botão direito em “Solução ContainRs”->Adicionar->Projeto-> Teste xUnit

Renomear classe para EmailCtor - Ela vai ser responsável por testar a inserção de e-mails na aplicação

Fazer com que o novo projeto (de teste) seja capaz de acessar as classes do projeto principal (que é ContainRs.WebApp)

Botão direito em “Dependencias”-> adicionar referência ao projeto->ContainRs.WebApp

E adicionar no início da classe EmailCtor “using ContainRs.WebApp.Models”

Criar o teste na classe

using ContainRs.WebApp.Models;

using System.Reflection;

namespace ContainRs.Testes

{

public class EmailCtor

{

[Fact]

public void Deve\_Lancar\_ArgumentException\_Quando\_Valor\_Invalido()

{

// arrange

string emailInvalido = "valor qualquer";

// act & assert

Assert.Throws<ArgumentException>(() => new Email(emailInvalido));

//AVALIA SE É LANÇADA UMA EXCEÇÃO DO TIPO ArgumentException QUANDO É

//TENTADO CRIAR UM OBJETO Email A PARTIR DE UMA STRING "valor qualquer"

}

}

}

Executar o teste - botão direito no nome do teste->executar teste

Se ficar tudo verde - significa que o teste fez o esperado, ou seja, lançou exceção para o caso onde foi tentado criar um e-mail com um string “valor qualquer”, o que não é um objeto Email aceito pela aplicação.

**CAMADAS DA CLEAN ARCHITECTURE**

1. **REGRAS DE NEGÓCIO / DOMÍNIO**

Regras e conceitos de negócio - models

1. **ADAPTADORES DE INTERFACE / INTERFACE DE ENTRADA E SAÍDA / INTERFACE**

Traduz dados de entrada/saída para/de outras camadas - no projeto, RegistroViewModel e ErrorViewModel, Controllers

A camada de Interface de Entrada e Saída na Arquitetura Limpa tem como propósito mediar a interação do sistema com o mundo externo. Ela define como os dados chegam e saem da aplicação, transformando-os em um formato compreensível para outras camadas. Com isso, ela é responsável por capturar eventos externos, sejam provenientes de uma interface de usuário, requisições HTTP ou mesmo mensagens de um sistema de filas.

Destacamos alguns padrões de projeto frequentemente encontrados na camada de Interface de Entrada e Saída:

-Mediator: responsável por orquestrar fluxos de processamento, este tipo é comumente utilizado dentro de controladores ou o próprio controlador pode ser o mediador, como RegistroController em nosso projeto.

-ViewModel ou DTO: representam os dados de entrada ou saída, que serão transportados (por isso o DTO: Data Transfer Object) para as rotinas internas do sistema. Os dados digitados em nosso formulário de registro foram representados pelo RegistroViewModel.

-Adapter: utilizado para conectar componentes externos necessários a execução de uma rotina específica do sistema. Em uma interpretação bem livre, podemos dizer que o tipo AppDbContext, que está sendo injetado no controlador RegistroController, é um exemplo de adapter, porque faz a ponte entre a rotina e a persistência de dados.

-Decorator: empregado para adicionar responsabilidades de maneira flexível, como por exemplo logging ou validação. Há várias propriedades do tipo Registro ViewModel com atributos de validação, aumentando assim sua capacidade.

Vale observar também que o padrão arquitetural MVC (Model-View-Controller) é usado para organizar todo o código que compõe a camada de interface. Controllers são responsáveis por receber os eventos externos, traduzindo as informações vindas do mundo externo, em seguida mediando as funções internas da aplicação em colaboração com os Models e por fim traduzindo de volta para o mundo externo, em geral por meio do HTML localizado nas Views.

Em C#, a camada de Interface de Entrada e Saída costuma aproveitar recursos como Controllers e Middlewares no ASP.NET Core, bem como Data Annotations para validações rápidas de entrada de dados. Classes e interfaces implementadas nesta camada também utilizam extensivamente tipos genéricos para abstrair dependências externas, e a utilização de interfaces como IHttpContextAccessor e ILogger é comum para lidar com o estado da requisição e a geração de logs.

Um anti-pattern comum nessa camada é escrever diretamente as regras de negócio no código que trata o evento externo, em nosso caso no controlador (alerta de spoiler 😁). Como esse código precisa lidar com tradução, validação e a mediação em si, colocar regras de negócio ali deixa o projeto muito vulnerável a mudanças.

1. **APLICAÇÃO / USE CASES**

Fluxos de tratamento do negócio - casos de uso.

A camada de Aplicação na Arquitetura Limpa tem como objetivo principal orquestrar os casos de uso do sistema, atuando como um intermediário entre as camadas de Domínio e as Interfaces de Entrada/Saída. Ela define a lógica de aplicação e os fluxos de trabalho que respondem às solicitações do usuário ou de outros sistemas, garantindo que as regras de negócio sejam aplicadas corretamente e que o sistema se comporte de maneira previsível e robusta.

Dentro da camada de Aplicação é comum encontrar padrões de projeto como:

Command: encapsula as requisições que representam as funções da aplicação; em nosso projeto ContainRs, RegistrarCliente é um comando.

Mediator: usado para orquestrar a comunicação entre casos de uso complexos. Apesar de não termos um exemplo deste padrão em nosso projeto, imagine um caso de uso em que seja necessário registrar um acontecimento para que outras partes da aplicação tenham ciência do ocorrido. Por exemplo, nosso registro de clientes deve disparar um evento ClienteRegistrado e o módulo de auditoria deve capturar esse evento e persisti-lo em sua base de dados. Podemos usar uma classe que implementa o padrão Mediator para o disparo desses eventos.

Result: encapsula o resultado de uma operação, incluindo informações sobre sucesso, falhas e mensagens associadas. No registro de clientes, poderíamos representar os tipos de resultado possíveis através de classes específicas. Por exemplo: cliente registrado com sucesso, falha na persistência do cliente, CPF já registrado, dentre outros.

Além destes padrões, em geral observamos um design de código baseado no CQRS, sigla para Command Query Responsibility Segregation, padrão que separa casos de uso em operações de escrita e leitura.

Algumas técnicas e recursos da linguagem C# são bastante utilizados na camada de Aplicação. Interfaces são empregadas para definir contratos de serviços e abstrações, enquanto genéricos são usados para representar serviços reutilizáveis, como tratadores de caso de uso (handlers) genéricos para comandos e/ou queries.

As palavras reservadas async/await são fundamentais nos métodos que executam os casos de uso, garantindo operações assíncronas e responsivas. Por fim, records são usados para representar objetos imutáveis de entrada e saída, facilitando a integridade e a simplicidade no transporte de dados.

1. **INFRAESTRUTURA**

Camada responsável por concluir o fluxo. Frameworks, drivers,...

Variável de conexão com BD é um exemplo

Serviços configurados em Program.cs é um exemplo.

Na Arquitetura Limpa, a camada de Infraestrutura tem como principal objetivo fornecer implementações concretas para interfaces definidas em outras camadas. Ela atua como uma ponte entre o sistema e o mundo externo, lidando com detalhes de persistência, acesso a APIs externas, manipulação de arquivos, envio de emails e outros serviços específicos.

Como abordaremos em mais detalhes a seguir, essa camada deve ser mantida desacoplada do núcleo do sistema, garantindo que as dependências externas não contaminem regras de negócio ou a lógica de aplicação.

Dentre os padrões de projeto frequentemente encontrados na camada de Infraestrutura, destacam-se:

Repository, para abstrair a persistência de dados;

Adapter, para converter interfaces de terceiros em formatos compreensíveis pelo sistema;

Factory, usado na criação de objetos complexos, como conexões de banco de dados;

Unit of Work, que garante a consistência dos dados a partir da coordenação de alterações realizadas em múltiplos repositórios, gerando através de transações.

No contexto do C#, os tipos da camada de Infraestrutura frequentemente utilizam recursos como Dependency Injection para gerenciar instâncias de serviços, LINQ para consultas sobre coleções ou bancos de dados, e async/await para realizar operações assíncronas, como chamadas a APIs externas ou operações de I/O. Também é comum o uso de bibliotecas e tipos populares, como Entity Framework, Dapper e HttpClient, que facilitam o desenvolvimento de funcionalidades específicas da camada.

Anti-patterns comuns na camada de Infraestrutura incluem a dependência direta em implementações concretas ao invés de abstrações, dificultando a testabilidade do sistema e aumentando o acoplamento.

Outro erro frequente é sobrecarregar repositórios com lógica de negócio, violando o princípio da separação de responsabilidades. Além disso, o uso excessivo de conexões abertas ao banco de dados ou a falta de gerenciamento adequado de recursos pode levar a problemas de desempenho e instabilidade no sistema.

Conforme mencionado em vídeo, no projeto ContainRs (e, diga-se de passagem, em qualquer projeto padrão web que use Asp.NET Core) você reconhecerá os componentes da camada de infraestrutura na classe Program.cs.

Projetos mais antigos, anteriores à versão 6 do .NET, ainda exigiam uma classe adicional, geralmente chamada de Startup.cs para configurar a infra. Nessa classe adicional os componentes são instanciados e ficam disponíveis para serem usados nos fluxos de negócio.

**API PARA A APLICAÇÃO - REGISTRO DE CLIENTES**

Criar um controlador capaz de receber as informações que serão usadas para registrar um cliente, ou seja, um endpoint, que vai retornar um JSON ou um *response code*

Em controllers->Criar ApiRegistroController

using ContainRs.WebApp.Data;

using ContainRs.WebApp.Models;

using Microsoft.AspNetCore.Mvc;

namespace ContainRs.WebApp.Controllers

{

[ApiController] //É UM CONTROLADOR PARA API

[Route("api/registros")]//ROTA DESSE CONTROLADOR

public class ApiRegistroController : ControllerBase //HERDA DE CONTROLLER BASE E NÃO Controller

{

//ATRIBUTOS

private readonly AppDbContext context;//CONEXÃO COM BD

//CONTRUTOR - JÁ COM INJEÇÃO DE DEPENDÊNDIA PARA DbContext

public ApiRegistroController(AppDbContext context)

{

this.context = context;

}

//MÉTODO POST PARA CADASTRO DE CLIENTE

[HttpPost]

public async Task<IActionResult> CreateAsync(RegistroViewModel request)

{

//NÃO PRECISA VALIDAÇÃO DE ESTADO NUM ENDPOINT DE API, PARA CHEGAR NELE

//JÁ DEVE ESTAR VALIDADO

//if (!ModelState.IsValid) return View("Index", form);

//NO VÍDEO, NÃO TEM ESSA PARTE DE VALIDAR A IDADE - ?

//var idade = DateTime.Today.Year - request.Nascimento.Year;

//if (idade < 18)

//{

// ModelState.AddModelError("Nascimento", "Obrigatório ter mais de 18 anos.");

// return View("Index", request);

//}

var cliente = new Cliente(request.Nome, new Email(request.Email), request.CPF)

{

Celular = request.Celular,

CEP = request.CEP,

Rua = request.Rua,

Numero = request.Numero,

Complemento = request.Complemento,

Bairro = request.Bairro,

Municipio = request.Municipio,

Estado = request.Estado

};

context.Clientes.Add(cliente);

await context.SaveChangesAsync();

//return RedirectToAction("Sucesso");

return Ok();

}

}

}

Agora, com essa classe criada, precisamos criar a classe que vai representar o *usecase* de registrar clientes.

Botão direito no projeto ContainRs.WebApp->Nova Pasta de nome UseCases

Nessa nova pasta->Nova classe de nome RegistrarCliente

using ContainRs.WebApp.Data;

using ContainRs.WebApp.Models;

namespace ContainRs.WebApp.UseCases

{

public class RegistrarCliente

{

//ATRIBUTOS

//ATRIBUTO PARA CONEXÃO BD

private readonly AppDbContext context;

public string Nome { get; private set; }

public Email Email { get; private set; }

public string CPF { get; private set; }

public string? Celular { get; set; }

public string? CEP { get; set; }

public string? Rua { get; set; }

public string? Numero { get; set; }

public string? Complemento { get; set; }

public string? Bairro { get; set; }

public string? Municipio { get; set; }

public string? Cidade { get; set; }

public string? Estado { get; set; }

//CONSTRUTOR

public RegistrarCliente(AppDbContext context, string nome, Email email, string cPF,

string? celular, string? cEP, string? rua,

string? numero, string? complemento, string? bairro,

string? municipio, string? estado)

{

this.context = context;

Nome = nome;

Email = email;

CPF = cPF;

Celular = celular;

CEP = cEP;

Rua = rua;

Numero = numero;

Complemento = complemento;

Bairro = bairro;

Municipio = municipio;

Estado = estado;

}

//MÉTODO QUE ADICIONA O CLIENTE A BD E RETORNA SEUS DADOS

public async Task<Cliente> ExecutarAsync()

{

var cliente = new Cliente(Nome, Email, CPF)

{

Celular = Celular,

CEP = CEP,

Rua = Rua,

Numero = Numero,

Complemento = Complemento,

Bairro = Bairro,

Municipio = Municipio,

Estado = Estado

};

context.Clientes.Add(cliente);

await context.SaveChangesAsync();

return cliente;

}

}

}

E então fazer o uso dessa classe para todas as ocasiões onde é necessário registrar um cliente, que no caso da nossa aplicação atual são dois: no controlador da aplicação web que recebe dados do formulário e na API que recebe dados da requisição

Controllers->RegistroController

public async Task<IActionResult> CreateAsync(RegistroViewModel form)

{

if (!ModelState.IsValid) return View("Index", form);

var idade = DateTime.Today.Year - form.Nascimento.Year;

if (idade < 18)

{

ModelState.AddModelError("Nascimento", "Obrigatório ter mais de 18 anos.");

return View("Index", form);

}

var useCase = new RegistrarCliente(context, form.Nome, new Email(form.Email),

form.CPF, form.Celular, form.CEP,

form.Rua, form.Numero, form.Complemento,

form.Bairro, form.Municipio, form.Estado);

await useCase.ExecutarAsync();

return RedirectToAction("Sucesso");

}

Controllers->ApiRegistroController

public async Task<IActionResult> CreateAsync(RegistroViewModel request)

{

//NÃO PRECISA VALIDAÇÃO DE ESTADO NUM ENDPOINT DE API, PARA CHEGAR NELE

//JÁ DEVE ESTAR VALIDADO

//if (!ModelState.IsValid) return View("Index", form);

//NO VÍDEO, NÃO TEM ESSA PARTE DE VALIDAR A IDADE - ?

//var idade = DateTime.Today.Year - request.Nascimento.Year;

//if (idade < 18)

//{

// ModelState.AddModelError("Nascimento", "Obrigatório ter mais de 18 anos.");

// return View("Index", request);

//}

var useCase = new RegistrarCliente(context, request.Nome, new Email(request.Email),

request.CPF, request.Celular, request.CEP,

request.Rua, request.Numero, request.Complemento,

request.Bairro, request.Municipio, request.Estado);

await useCase.ExecutarAsync();

//return RedirectToAction("Sucesso");

return Ok();

}

Agora, temos esse usecase sendo utilizado em todas suas ocorrências da aplicação, assim como determinado pela camada de aplicação da Clean Architecture

**SEPARANDO A CAMADA DE DOMÍNIO**

Para separar a camada de domínio do resto da aplicação, cria-se um novo projeto, que vai ser uma biblioteca de classes, somente com as classes que representam o domínio

Botão direito em Solução ContainRs-> adicionar-> Novo projeto-> Biblioteca de classes-> Criar projeto de nome ContainRs.Domain e nele criar a pasta Models, que vai receber os arquivos Cliente.cs e Email.cs de ContainRs.WebApp/Models.

Botão direito em ContainRs.Domain->Sincronizar namespaces

**SEPARANDO A CAMADA DE APLICAÇÃO**

Botão direito em Solução ContainRs-> adicionar-> Novo projeto-> Biblioteca de classes-> Criar projeto de nome ContainRs.Application e nele colar a pasta UseCases do projeto ContainRs.WebApp

-Sincronizar o namespace

Também será preciso referenciar projetos, para que um possa enxergar o outro, mas isso não pode ser feito de qualquer maneira

**\*Regras da *Clean Architecture:***

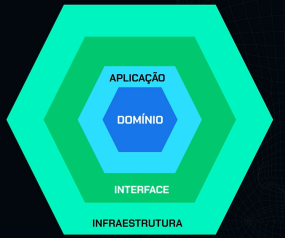
***Camadas externas só enxergam a sua interna, OU SEJA***

**-Domínio só enxerga ela mesma;**

**-Aplicação só consegue enxergar domínio;**

**-Interface só enxerga aplicação (que enxerga domínio);**

**-Infraestrutura só enxerga interface (que enxerga aplicação (que enxerga domínio)).**



Então, em ContainRs.Application/Dependências-> Adicionar referência ao projeto-> ContainRs.Domain

Feito isso, no código ainda nos resta um problema: a variável context, que é do tipo AppDbContext não tem sua devida referência, e nesse caso nem pode ser referenciada a partir dessa camada (aplicação) porque a camada de aplicação só consegue enxergar a camada de domínio, e AppDbContext é uma classe da camada de infraestrutura. Nesse caso, o que pode ser feito?

Criar uma abstração das operações de AppDbContext, que essencialmente funciona, nesse caso, para acessar os dados dos clientes na BD, ou seja, criar uma interface que em dado momento, será implementada pela camada devida. Nesse caso, uma interface na camada de aplicação que será implementada na camada de infraestrutura.

Isso é feito criando, na camada de aplicação, uma interface do tipo IClienteRepository

Botão direito em ContainRs.Application-> Criar pasta Repositories-> Nessa pasta criar uma interface IClientRepository com um método abstrato Task<Cliente>

using ContainRs.Domain.Models;

namespace ContainRs.Application.Repositories

{

public interface IClienteRepository

{

Task<Cliente> AddAsync(Cliente cliente);

}

}

E no arquivo UseCases/RegistrarCliente.cs fazer as devidas mudanças para usar IClientRepository e não AppDbContext

//ATRIBUTOS

//ATRIBUTO PARA ABSTRAÇÃO DA CONEXÃO COM BD

private readonly IClienteRepository repository;

.

.

.

//CONSTRUTOR

public RegistrarCliente(IClienteRepository repository,...

.

.

.

//this.context = context;

this.repository = repository;

.

.

.

//context.Clientes.Add(cliente);

await repository.AddAsync(cliente);

//await context.SaveChangesAsync(); //NÃO É NECESSÁRIO PQ VAI SER USADO NA IMPLEMENTAÇÃO

//REPOSITÓRIO

**RESOLVENDO ERROS NA INTERFACE**

Em ContainRs.WebApp

Controllers/ApiRegistroController e Controllers/RegistroController com erros

Referenciar ContainRs.Application (já que Interface enxerga Aplicação)

ContainRs.WebApp/Dependências-> Referenciar projeto-> ContainRs.Application

Referenciar ContainRs.Domain (já que Interface enxerga Domínio (através de aplicação))

Nesse caso só é preciso adicionar

using ContainRs.Domain.Models;

Dado que, mesmo sem referenciar o projeto ContainRs.Domain, já estamos fazendo uso de ContainRs.Application (que faz uso de ContainRs.Domain), ou seja, uma referência indireta.

Ao final, ambas classes ApiRegistroController e RegistroController ficam com suas importações dessa maneira

using ContainRs.WebApp.Data;

using ContainRs.WebApp.Models;

using ContainRs.Domain.Models;

//using ContainRs.WebApp.UseCases;

using ContainRs.Application.UseCases;

using Microsoft.AspNetCore.Mvc;

Nesse momento, o único problema restante na camada Interface é o uso da variável context, que é do tipo AppDbContext, e considerando que fizemos as mudanças na camada de aplicação, agora a classe AppDbContext precisa implementar a interface IClientRepository que criamos

Em Data/AppDbContext

using ContainRs.Application.Repositories;

using ContainRs.Domain.Models;

.

.

.

//public class AppDbContext : DbContext

public class AppDbContext : DbContext, IClienteRepository

.

.

.

//IMPLEMENTAÇÃO DO MÉTODO HERDADO DA INTERFACE ContainRs.Application/Repositories/IClienteRepository

public async Task<Cliente> AddAsync(Cliente cliente)

{

await Clientes.AddAsync(cliente);//ADICIONA CLIENTE A BD

await SaveChangesAsync();//SALVA AS MUDANÇAS

return cliente;

}

Ao final, para tornar toda a solução compilável, também são necessárias mudanças no projeto de teste, já que agora, ele referencia a camada ContainRs.Domain e não ContainRs.WebApp

Botão direito em ContainRs.Testes-> Adicionar referência de projeto-> desmarcar ContainRs.WebApp e marcar ContainRs.Domain

Mudar em ContainRs.Testes/EmailCtor

//using ContainRs.WebApp.Models;

using ContainRs.Domain.Models;

Agora, temos uma solução completa, implementada com a *Clean Architecture* , que tem um caso de uso (registrar clientes).

TEORIA - PRINCÍPIOS SOLID

Os princípios SOLID são um conjunto de diretrizes criadas para tornar os sistemas de software mais fáceis de entender, modificar e manter. Esses princípios estão profundamente alinhados com os fundamentos da arquitetura limpa, pois promovem a separação de responsabilidades, baixo acoplamento e alta coesão. Ao aplicá-los, pessoas desenvolvedoras podem criar aplicações mais robustas e flexíveis, que se adaptam bem a mudanças e são mais simples de testar e escalar.

O princípio da Responsabilidade Única (**S**ingle Responsibility Principle – SRP) afirma que uma classe deve ter apenas uma razão para mudar, ou seja, deve possuir uma única responsabilidade bem definida. Isso reduz a complexidade ao garantir que cada componente do sistema esteja focado em um propósito específico, facilitando manutenções e atualizações futuras sem introduzir efeitos colaterais.

Como já mencionado, o nome de um tipo (seja classe, interface, enum, struct ou record) deve indicar esta responsabilidade. Atente-se ao nomear um tipo: sua dificuldade pode ser um sinal de que estamos ferindo o SRP. Prefixos e sufixos no nome também ajudam a indicar padrões e, portanto, a principal responsabilidade do tipo.

O princípio Aberto-Fechado (**O**pen/Closed Principle – OCP) sugere que entidades de software devem estar abertas para extensão, mas fechadas para modificação. Em outras palavras, é preferível adicionar novas funcionalidades através de extensões em vez de alterar o código existente, minimizando o risco de introduzir erros em um sistema estável.

Em nosso projeto ContainRs, imagine que fosse necessário consultar CEPs usando outro serviço. Atualmente estamos ferindo o OCP, porque dependemos diretamente da interface IViaCepService. O ideal seria termos uma interface genérica de consulta a CEPs e injetá-la nos locais onde a consulta fosse necessária. Para usar outro serviço de consulta que não o ViaCep, bastaria criar outra implementação da interface genérica e configurar essa implementação no container de injeção de dependência.

O princípio de Substituição de Liskov (**L**iskov Substitution Principle – LSP) estipula que uma classe derivada deve poder substituir sua classe base sem comprometer o comportamento esperado do sistema. Isso garante que a herança seja usada corretamente e que os contratos entre classes sejam respeitados, promovendo a reutilização e a previsibilidade.

O princípio da Segregação de Interfaces (**I**nterface Segregation Principle – ISP) preconiza que os clientes não devem ser forçados a depender de interfaces que não utilizam. Isso implica em criar interfaces específicas e enxutas, reduzindo o acoplamento e evitando que alterações em uma parte do sistema impactem indevidamente outras partes. Em nosso projeto, podemos segregar a interface IClienteRepository em interfaces distintas, uma para cada operação do repositório (inclusão, remoção, dentre outras).

Por fim, o princípio da Inversão de Dependência (**D**ependency Inversion Principle – DIP) propõe que módulos de alto nível não devem depender de módulos de baixo nível, mas ambos devem depender de abstrações. Isso torna o sistema mais flexível e resiliente às mudanças, pois as dependências podem ser facilmente substituídas por implementações alternativas. Podemos relacionar diretamente esse princípio com a regra fundamental da arquitetura limpa: "camadas internas (alto nível) não devem depender de camadas mais externas (baixo nível)".

**INSERÇÃO DE NOVOS CLIENTES NO PROJETO**

Em ContainRs.WebApp/Connected Services/Banco de dados do SQL Server-> botão direito-> Abir no pesquisador de objetos

Em localdb/Bancos de dados/ContainRs.Database-> Nova consulta-> inserir o script abaixo

-- Script para inserir 200 clientes na tabela Clientes

DECLARE @Counter INT = 1;

DECLARE @Estados TABLE (Estado NVARCHAR(MAX));

-- Inserir estados brasileiros na tabela temporária

INSERT INTO @Estados (Estado)

VALUES

('AC'), ('AL'), ('AP'), ('AM'), ('BA'), ('CE'), ('DF'), ('ES'), ('GO'),

('MA'), ('MT'), ('MS'), ('MG'), ('PA'), ('PB'), ('PR'), ('PE'), ('PI'),

('RJ'), ('RN'), ('RS'), ('RO'), ('RR'), ('SC'), ('SP'), ('SE'), ('TO');

-- Variáveis auxiliares para dados fictícios

DECLARE @Id UNIQUEIDENTIFIER;

DECLARE @Nome NVARCHAR(MAX);

DECLARE @Email NVARCHAR(MAX);

DECLARE @CPF NVARCHAR(MAX);

DECLARE @Celular NVARCHAR(MAX);

DECLARE @CEP NVARCHAR(MAX);

DECLARE @Rua NVARCHAR(MAX);

DECLARE @Numero NVARCHAR(MAX);

DECLARE @Complemento NVARCHAR(MAX);

DECLARE @Bairro NVARCHAR(MAX);

DECLARE @Municipio NVARCHAR(MAX);

DECLARE @Cidade NVARCHAR(MAX);

DECLARE @Estado NVARCHAR(MAX);

-- Listas de nomes femininos e masculinos

DECLARE @NomesFemininos TABLE (Nome NVARCHAR(MAX));

DECLARE @NomesMasculinos TABLE (Nome NVARCHAR(MAX));

INSERT INTO @NomesFemininos (Nome)

VALUES ('Ana'), ('Maria'), ('Júlia'), ('Sofia'), ('Laura'), ('Isabella'), ('Alice'), ('Beatriz'), ('Larissa'), ('Camila');

INSERT INTO @NomesMasculinos (Nome)

VALUES ('João'), ('Pedro'), ('Lucas'), ('Gabriel'), ('Miguel'), ('Matheus'), ('Rafael'), ('Gustavo'), ('Arthur'), ('Henrique');

-- Inserir 200 clientes

WHILE @Counter <= 200

BEGIN

-- Gerar dados fictícios

SET @Id = NEWID();

-- Alternar entre nomes femininos e masculinos

IF @Counter % 2 = 0

SELECT TOP 1 @Nome = Nome FROM @NomesFemininos ORDER BY NEWID();

ELSE

SELECT TOP 1 @Nome = Nome FROM @NomesMasculinos ORDER BY NEWID();

SET @Email = LOWER(@Nome) + CAST(@Counter AS NVARCHAR) +

CASE

WHEN @Counter % 3 = 1 THEN '@exemplo.org'

WHEN @Counter % 3 = 2 THEN '@email.org'

ELSE '@email.com.br'

END;

SET @CPF = RIGHT('000' + CAST(10000000000 + (@Counter \* 11) AS NVARCHAR), 11); -- CPF fictício válido

SET @Celular = '(11) 9' + CAST(100000000 + @Counter AS NVARCHAR);

SET @CEP = '12345-' + RIGHT('000' + CAST(@Counter AS NVARCHAR), 3);

SET @Rua = 'Rua Fictícia ' + CAST(@Counter AS NVARCHAR);

SET @Numero = CAST(@Counter AS NVARCHAR);

SET @Complemento = NULLIF('Apto ' + CAST((@Counter % 10) + 1 AS NVARCHAR), 'Apto 1');

SET @Bairro = 'Bairro ' + CAST(@Counter AS NVARCHAR);

SET @Municipio = 'Municipio ' + CAST(@Counter AS NVARCHAR);

SET @Cidade = 'Cidade ' + CAST(@Counter AS NVARCHAR);

-- Selecionar um estado aleatório

SELECT TOP 1 @Estado = Estado FROM @Estados ORDER BY NEWID();

-- Inserir na tabela Clientes

INSERT INTO [dbo].[Clientes] (

[Id], [Nome], [Email], [CPF], [Celular], [CEP], [Rua], [Numero],

[Complemento], [Bairro], [Municipio], [Cidade], [Estado]

)

VALUES (

@Id, @Nome, @Email, @CPF, @Celular, @CEP, @Rua, @Numero,

@Complemento, @Bairro, @Municipio, @Cidade, @Estado

);

SET @Counter = @Counter + 1;

END

**CRIANDO UM NOVO CASO DE USO - CONSULTA DE CLIENTE POR ESTADO**

Antes de criar um novo caso de uso, vamos primeiro criar o novo model que vai representar o Estado, uma classe UnidadeFederativa

ContainRs.Domain/Models-> Novo enum UnidadeFederativa (já com o método para converter de string para enum)

namespace ContainRs.Domain.Models;

public static class UfStringConverter

{

//MÉTODO QUE CONVERTE UMA STRING PARA ENUM

public static UnidadeFederativa? From(string? uf)

{

if (uf is null) return null;//SE A STRING FOR NULL, RETORNA NULL

//SE NÃO CONSEGUIR CONVERTER, RETORNA NULL

return Enum

.TryParse<UnidadeFederativa>(uf, out var parsedUf) ? parsedUf : null;

}

}

public enum UnidadeFederativa

{

AC,

AL,

AP,

AM,

BA,

CE,

DF,

ES,

GO,

MA,

MT,

MS,

MG,

PA,

PB,

PR,

PE,

PI,

RJ,

RN,

RS,

RO,

RR,

SC,

SP,

SE,

TO

}

Adicionar também, na interface IClientRepository um método que vai ser responsável por consultar clientes utilizando um filtro

public interface IClienteRepository

{

//MÉTODO ABSTRATO PARA ADICIONAR CLIENTE

Task<Cliente> AddAsync(Cliente cliente);

//MÉTODO ABSTRATO PARA CONSULTAR CLIENTES SEGUINDO A CONDIÇÃO DA EXPRESSÃO

//Expression<Func<Cliente, bool>>? filtro = default - RECURSO DO ENTITY

//QUE REPRESENTA UM CONJUNTO DE CONDIÇÕES PARA SER USADA NA CONSULTA SQL

//RETORNA NULL POR PADRÃO, CASO NÃO HAJA RESPOSTA

Task<IEnumerable<Cliente>> GetAsync(Expression<Func<Cliente, bool>>? filtro = default);

}

Feitos o model novo e o método abstrato, agora é possível construir o novo usecase de consultar clientes

ContainRs.Application/UseCases-> Criar classe ConsultarCliente

using ContainRs.Application.Repositories;

using ContainRs.Domain.Models;

namespace ContainRs.Application.UseCases

{

public class ConsultarCliente

{

//ATRIBUTOS

private readonly IClienteRepository repository;

public UnidadeFederativa? Estado { get; }

//CONSTRUTOR

public ConsultarCliente(UnidadeFederativa? estado, IClienteRepository repository)

{

Estado = estado;

this.repository = repository;

}

//MÉTODO QUE RETORNA UM ENUMERABLE DE CLIENTES

public Task<IEnumerable<Cliente>> ExecutarAsync()

{

if(Estado is not null)

{

return repository.GetAsync(c => c.Estado == Estado);

}

//SE Estado FOR NULL, RETORNA O DEFAULT DO MÉTODO GetAsync (QUE É NULL)

return repository.GetAsync();

}

}

}

Considerando que o novo model foi construído na camada de domínio, também é preciso aplicar o seu uso em ContainRs.Domain/Models/Cliente e ContainRs.Application/UseCases/RegistrarCliente

ContainRs.Domain/Models/Cliente

...

//public string? Estado { get; set;}

public UnidadeFederativa? Estado { get; set; }

...

ContainRs.Application/UseCases/RegistrarCliente

...

//public string? Estado { get; set;}

public UnidadeFederativa? Estado { get; set; }

.

.

.

string? municipio, UnidadeFederativa? estado)

//string? municipio, string? estado)

Feito isso, as mudanças necessárias para o usecase de consultar clientes está completa nas camadas de domain e application. Agora falta mudar nas camadas de interface e infraestrutura.

Para a camada de Infraestrutura

ContainRs.WebApp/Data/AppDbContext->

Implementar método GetAsync

//IMPLEMENTAÇÃO DO MÉTODO HERDADO DA INTERFACE ContainRs.Application/Repositories/IClienteRepository

//MÉTODO QUE RETORNA UMA LISTA DE CLIENTES DA BD, DADO UM FILTRO

public async Task<IEnumerable<Cliente>> GetAsync(Expression<Func<Cliente, bool>>? filtro = default)

{

IQueryable<Cliente> queryClientes = this.Clientes;

if(filtro != null)

{

queryClientes = queryClientes.Where(filtro);

}

return await queryClientes.AsNoTracking().ToListAsync();

}

Mudar método OnModelCreating para poder converter o campo Cliente.Estado entre Enum e string

modelBuilder.Entity<Cliente>()

.Property(c => c.Estado)//O CAMPO ESTADO

.HasConversion<string>();//É CONVERTIDO DE SEU TIPO ORIGINAL PARA STRING (ENUM<->STRING)

Para a camada de Interface

ContainRs.WebApp/Models-> Criar record ClienteResponse, que vai ser responsável por mostrar apenas algumas informações do cliente quando a consulta for feita

namespace ContainRs.WebApp.Models

{

public record ClienteResponse(string Id, string Nome, string Email);

}

ContainRs.WebApp/Controllers-> Criar controlador vazio ApiClienteController

using ContainRs.Application.UseCases;

using ContainRs.Domain.Models;

using ContainRs.WebApp.Data;

using ContainRs.WebApp.Models;

using Microsoft.AspNetCore.Mvc;

namespace ContainRs.WebApp.Controllers

{

[ApiController]

[Route("api/clientes")]

public class ApiClientesController : ControllerBase

{

private readonly AppDbContext context;

public ApiClientesController(AppDbContext context)

{

this.context = context;

}

//MÉTODO QUE MOSTRA TODOS CLIENTES DO BD, DADO UM ESTADO

[HttpGet]

public async Task<IActionResult> GetAsync(string? estado)

{

var useCase = new ConsultarCliente(UfStringConverter.From(estado), context);

var clientes = await useCase.ExecutarAsync();

return Ok(clientes.Select(c =>

new ClienteResponse(c.Id.ToString(), c.Nome, c.Email.Value)));

}

}

}

E considerando que a entidade cliente agora tem o campo Estado como um objeto UnidadeFederativa, também temos que mudar seu uso nos arquivos ApiRegistroController e RegistroController

ContainRs.WebApp/Controllers/ApiRegistroController

var useCase = new RegistrarCliente(context, request.Nome, new Email(request.Email),

request.CPF, request.Celular, request.CEP,

request.Rua, request.Numero, request.Complemento,

request.Bairro, request.Municipio, UfStringConverter.From(request.Estado));

//request.Bairro, request.Municipio, request.Estado);

ContainRs.WebApp/Controllers/RegistroController

var useCase = new RegistrarCliente(context, form.Nome, new Email(form.Email),

form.CPF, form.Celular, form.CEP,

form.Rua, form.Numero, form.Complemento,

form.Bairro, form.Municipio, UfStringConverter.From(form.Estado));

//form.Bairro, form.Municipio, form.Estado);

Agora, pelo postman, acessando a url de consulta de clientes, pelo GET e passando o parâmetro estado=ES, conseguimos consultar todos clientes cujo estado é ES.

https://localhost:7035/api/clientes?estado=ES

E como configuramos o retorno do método GetAsync para usar o record ClientResponse, para cada cliente é mostrado apenas o id, nome e email, o que é um comportamento esperado da camada de interface (receber os dados do mundo externo, fazer comunicação com o sistema e devolver os dados tratados).

**ESTUDOS FUTUROS**

Tipo de propriedade Guid

Tipo de propriedade readonly

Expressões regulares em c#