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, abrí-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 + .

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

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 UseCase da Clean Architecture

**ESTUDOS FUTUROS**

Tipo de propriedade Guid

Tipo de propriedade readonly

Expressões regulares em c#