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

**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 ensiar 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.