**Capítulo 1: Configuração do Ambiente e Criação do Projeto**

Neste capítulo, vamos preparar nosso ambiente de desenvolvimento, criar a estrutura inicial do projeto e instalar todas as ferramentas (pacotes) necessárias para construir nossa aplicação.

**1.1 - Pré-requisitos**

Antes de começar, certifique-se de que você tem o **.NET SDK** (Software Development Kit) instalado em sua máquina. Usaremos o .NET 8 ou uma versão mais recente.

Você pode verificar sua versão abrindo um terminal (Prompt de Comando, PowerShell, ou Terminal) e digitando:

dotnet --version

**1.2 - Criando o Projeto ASP.NET Core MVC**

Agora, vamos criar o projeto.

1. Abra seu terminal.
2. Navegue até a pasta onde você deseja salvar seus projetos (por exemplo, C:\Projetos).
3. Execute o comando abaixo. Ele criará uma nova pasta chamada ClinicaApp com toda a estrutura inicial de um projeto MVC.

dotnet new mvc -o ClinicaApp

1. Após a criação, entre na pasta do projeto com o comando:

cd ClinicaApp

A partir de agora, todos os comandos devem ser executados de dentro desta pasta.

**1.3 - Instalando os Pacotes (Ferramentas)**

Nossa aplicação precisará de algumas "ferramentas" extras para se comunicar com o banco de dados e para nos ajudar a gerar código. Essas ferramentas são pacotes que adicionamos ao projeto via **NuGet** (o gerenciador de pacotes do .NET).

Execute os seguintes comandos, um de cada vez:

1. **Provedor do SQLite para o Entity Framework Core:** *Permite que nosso código "converse" com o banco de dados SQLite.*

dotnet add package Microsoft.EntityFrameworkCore.Sqlite

1. **Ferramentas de Design do EF Core:** *Necessário para executar comandos do Entity Framework no terminal, como as migrations.*

dotnet add package Microsoft.EntityFrameworkCore.Design

1. **Ferramentas de Geração de Código Web:** *Usaremos esta ferramenta mais tarde para gerar automaticamente nosso Controller e nossas Views (scaffolding).*

dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design

1. **Ferramentas de Linha de Comando do EF Core:** *Complementa as ferramentas de design para a linha de comando.*

dotnet add package Microsoft.EntityFrameworkCore.Tools

Pronto! A estrutura do nosso projeto está criada e todas as dependências necessárias foram instaladas. Agora temos o "esqueleto" da nossa aplicação.

Finalizamos a configuração inicial. No próximo capítulo, vamos definir a estrutura dos nossos dados, criando a classe que representará o cliente.

**Capítulo 2: Definindo os Dados (O Model e o Contexto)**

Com o projeto criado, o próximo passo é definir a "forma" dos nossos dados. Faremos isso em duas partes:

1. **Model**: Uma classe C# que representa os dados que queremos salvar. No nosso caso, um Cliente.
2. **DbContext**: Uma classe especial do Entity Framework que representa a sessão com o banco de dados e nos permite salvar e consultar os dados do nosso Model.

**2.1 - O Modelo do Cliente (Cliente.cs)**

Esta classe define todas as informações (propriedades) que um cliente terá.

1. No seu editor de código (VS Code, Visual Studio), encontre a pasta Models.
2. Crie um novo arquivo dentro dela chamado Cliente.cs.
3. Cole o seguinte código neste novo arquivo:

// Local do Arquivo: /Models/Cliente.cs

using System.ComponentModel.DataAnnotations;

namespace ClinicaApp.Models

{

public class Cliente

{

[Key] // Atributo que informa ao EF Core que esta é a chave primária

public int Id { get; set; }

[Required(ErrorMessage = "O campo Nome é obrigatório.")] // O campo não pode ser nulo ou vazio

[StringLength(100, ErrorMessage = "O Nome deve ter no máximo 100 caracteres.")]

[Display(Name = "Nome Completo")] // Texto que aparecerá nos formulários

public string Nome { get; set; }

[Required(ErrorMessage = "O campo Data de Nascimento é obrigatório.")]

[Display(Name = "Data de Nascimento")]

[DataType(DataType.Date)] // Ajuda o navegador a mostrar um seletor de data

public DateTime DataNascimento { get; set; }

[StringLength(15)]

public string? Telefone { get; set; } // O '?' indica que este campo pode ser nulo

[EmailAddress(ErrorMessage = "O formato do e-mail é inválido.")]

[StringLength(100)]

public string? Email { get; set; }

[Display(Name = "Data de Cadastro")]

public DateTime DataCadastro { get; set; }

}

}

**2.2 - O Contexto do Banco de Dados (ClinicaContext.cs)**

Esta classe é a ponte entre nossas classes C# e o banco de dados.

1. Crie uma nova pasta na raiz do seu projeto chamada Data.
2. Dentro da pasta Data, crie um novo arquivo chamado ClinicaContext.cs.
3. Cole o seguinte código nele:

// Local do Arquivo: /Data/ClinicaContext.cs

using ClinicaApp.Models;

using Microsoft.EntityFrameworkCore;

namespace ClinicaApp.Data

{

public class ClinicaContext : DbContext

{

// O construtor recebe as opções de configuração do banco (que definiremos depois)

public ClinicaContext(DbContextOptions<ClinicaContext> options) : base(options)

{

}

// Esta propriedade se tornará uma tabela "Clientes" no nosso banco de dados.

// Cada registro na tabela terá o formato da classe "Cliente".

public DbSet<Cliente> Clientes { get; set; }

}

}

Com esses dois arquivos, definimos toda a estrutura de dados da nossa aplicação. O Entity Framework Core agora sabe o que é um Cliente e como ele deve ser armazenado.

**Capítulo 3: Configurando a Conexão com o Banco de Dados**

Agora que temos nosso Model e nosso DbContext, precisamos dizer à nossa aplicação como usá-los. Faremos isso configurando a "Injeção de Dependência" e a "String de Conexão".

**3.1 - Injeção de Dependência (DI)**

Injeção de Dependência é um padrão de design fundamental no ASP.NET Core. Em vez de nosso Controller criar uma instância do ClinicaContext diretamente, nós pedimos ao ASP.NET Core para "injetar" uma para nós. Isso torna o código mais flexível, testável e desacoplado.

Configuramos essa injeção no arquivo Program.cs.

1. Abra o arquivo Program.cs na raiz do seu projeto.
2. Vamos adicionar o código para registrar nosso ClinicaContext.

Adicione as seguintes linhas ao seu Program.cs:

// Local do Arquivo: /Program.cs

// Adicione estes 'usings' no topo do arquivo

using ClinicaApp.Data;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// --- INÍCIO DA ADIÇÃO ---

// 1. Pega a string de conexão do arquivo appsettings.json

var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

// 2. Adiciona o DbContext ao contêiner de serviços (Injeção de Dependência)

// e configura para usar o SQLite com a string de conexão obtida.

builder.Services.AddDbContext<ClinicaContext>(options =>

options.UseSqlite(connectionString));

// --- FIM DA ADIÇÃO ---

// Add services to the container.

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

**3.2 - A String de Conexão (appsettings.json)**

Nosso código agora procura por uma string de conexão chamada "DefaultConnection", mas ainda não a definimos. Faremos isso no arquivo appsettings.json.

1. Abra o arquivo appsettings.json na raiz do projeto.
2. Adicione a seção ConnectionStrings conforme abaixo.

JSON

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"DefaultConnection": "Server=NOTE\\SQLEXPRESS01;Database=bdChamados;Trusted\_Connection=True;TrustServerCertificate=True"

}

}

**OBS Não esqueça de definir o seu servidor e o nome do BD**

**O que Data Source=Clinica.db significa?** Significa que nosso banco de dados SQLite será um único arquivo chamado Clinica.db, que será criado na raiz do projeto.

**3.3 - Criando o Banco de Dados com Migrations**

Agora que o EF Core sabe o que salvar (Cliente.cs) e como se conectar (ClinicaContext e appsettings.json), o último passo é criar o banco de dados. Usamos um recurso poderoso chamado **Migrations**.

Uma "migration" é um arquivo de código que descreve as alterações a serem feitas no banco de dados (como criar uma tabela, adicionar uma coluna, etc.).

1. No seu terminal, certifique-se de que você está na pasta raiz do projeto ClinicaApp.
2. Execute o comando para criar nossa primeira migration. Vamos chamá-la de InitialCreate:

Add-migration InitialCreate

Você verá que uma nova pasta Migrations foi criada, contendo um arquivo com o código para criar a tabela Clientes.

1. Agora, execute o comando para aplicar essa migration e, finalmente, criar o arquivo do banco de dados:

Update-database

Após este comando, você verá um novo arquivo **Clinica.db** na raiz do seu projeto. Nosso banco de dados está pronto!

**Capítulo 4: Criando o Controller e a View de Listagem (Manual)**

Neste capítulo, vamos criar manualmente o ClientesController e a primeira View, que será responsável por listar todos os clientes cadastrados no banco de dados.

**4.1 - Criando o ClientesController**

O Controller é o "cérebro" da nossa aplicação. Ele recebe as requisições do navegador, decide o que fazer (por exemplo, buscar dados no banco), e então escolhe qual "tela" (View) enviar de volta como resposta.

1. No seu editor, encontre a pasta Controllers.
2. Crie um novo arquivo dentro dela chamado ClientesController.cs.
3. Cole o seguinte código. Por enquanto, ele terá apenas a ação Index, que é a ação padrão para listar itens.

// Local do Arquivo: /Controllers/ClientesController.cs

using ClinicaApp.Data;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace ClinicaApp.Controllers

{

public class ClientesController : Controller

{

// Campo privado para armazenar o contexto do banco de dados

private readonly ClinicaContext \_context;

// O construtor recebe o contexto via Injeção de Dependência

public ClientesController(ClinicaContext context)

{

\_context = context;

}

// GET: /Clientes ou /Clientes/Index

// Esta é a ação que será executada para listar os clientes

public async Task<IActionResult> Index()

{

// 1. Acessa o banco de dados através do context

// 2. Pega todos os clientes da tabela "Clientes"

// 3. Converte para uma lista de forma assíncrona

var clientes = await \_context.Clientes.ToListAsync();

// 4. Retorna a View "Index" e passa a lista de clientes para ela

return View(clientes);

}

}

}

**4.2 - Criando a View Index.cshtml**

A View é a camada de apresentação, o código que gera o HTML que o usuário vê no navegador. Ela recebe os dados do Controller e os exibe.

1. Dentro da pasta Views, crie uma **nova pasta** chamada Clientes (o nome deve ser o mesmo do Controller, sem o sufixo "Controller").
2. Dentro desta nova pasta Views/Clientes, crie um novo arquivo chamado Index.cshtml (o nome deve ser o mesmo da nossa ação no controller).
3. Cole o código Razor abaixo.

HTML

@\* Local do Arquivo: /Views/Clientes/Index.cshtml \*@

@\* 1. A diretiva @model informa à View qual o tipo de dado ela está recebendo do Controller.

Neste caso, uma lista (IEnumerable) de objetos do tipo Cliente. \*@

@model IEnumerable<ClinicaApp.Models.Cliente>

@{

// Define o título que aparecerá na aba do navegador

ViewData["Title"] = "Lista de Clientes";

}

<h1>@ViewData["Title"]</h1>

<p>

@\* Tag Helper que cria um link para a ação "Create" do nosso controller \*@

<a asp-action="Create" class="btn btn-primary">Novo Cliente</a>

</p>

<table class="table table-striped table-hover">

<thead>

<tr>

<th>

@\* Exibe o nome do campo definido no atributo [Display] do Model \*@

@Html.DisplayNameFor(model => model.Nome)

</th>

<th>

@Html.DisplayNameFor(model => model.Email)

</th>

<th>

@Html.DisplayNameFor(model => model.Telefone)

</th>

<th></th> </tr>

</thead>

<tbody>

@\* 2. Loop "foreach" para percorrer cada cliente da lista recebida no @model \*@

@foreach (var item in Model) {

<tr>

<td>

@\* Exibe o valor da propriedade Nome do item atual \*@

@Html.DisplayFor(modelItem => item.Nome)

</td>

<td>

@Html.DisplayFor(modelItem => item.Email)

</td>

<td>

@Html.DisplayFor(modelItem => item.Telefone)

</td>

<td>

@\* 3. Links de Ação para cada item, passando o ID do item na rota \*@

<a asp-action="Edit" asp-route-id="@item.Id" class="btn btn-secondary btn-sm">Editar</a> |

<a asp-action="Details" asp-route-id="@item.Id" class="btn btn-info btn-sm">Detalhes</a> |

<a asp-action="Delete" asp-route-id="@item.Id" class="btn btn-danger btn-sm">Excluir</a>

</td>

</tr>

}

</tbody>

</table>

**4.3 - Testando a Página de Listagem**

Mesmo que ainda não tenhamos clientes, já podemos ver nossa página funcionando.

1. No terminal, na raiz do projeto, execute o comando para rodar a aplicação:

Execute o projeto

1. Abra seu navegador e acesse a URL fornecida (ex: https://localhost:PORTA).
2. Adicione /clientes ao final da URL: https://localhost:PORTA/clientes.

Você deverá ver a página "Lista de Clientes" com o cabeçalho da tabela, mas sem nenhum registro, pois o banco de dados está vazio. O botão "Novo Cliente" e os links de ação ainda não funcionarão, pois não criamos as ações correspondentes no controller.

Concluímos a funcionalidade de listagem. Nosso próximo passo será dar vida ao botão "Novo Cliente", criando a ação e a view para o formulário de cadastro.

**Capítulo 5: Criando Novos Clientes (Manual)**

A funcionalidade de "Criar" em aplicações web é composta por duas partes distintas que trabalham juntas:

1. **Ação GET**: Ocorre quando o usuário clica no link "Novo Cliente". O servidor simplesmente retorna a página com o formulário em branco.
2. **Ação POST**: Ocorre quando o usuário preenche o formulário e clica no botão "Salvar". O navegador "posta" (envia) os dados para o servidor, que os valida e salva no banco.

Vamos implementar ambas as partes.

**5.1 - As Ações Create no Controller**

Vamos adicionar as duas ações (GET e POST) ao nosso ClientesController.cs.

1. Abra o arquivo /Controllers/ClientesController.cs.
2. Adicione os dois métodos abaixo dentro da classe ClientesController (pode ser logo após o método Index()).

// Adicione este código dentro da classe ClientesController

// GET: Clientes/Create

// Ação que mostra o formulário de criação em branco.

public IActionResult Create()

{

// Apenas retorna a View correspondente, que conterá o formulário.

return View();

}

// POST: Clientes/Create

// Ação que recebe os dados do formulário e os salva no banco.

[HttpPost] // Este atributo especifica que a ação responde apenas a requisições POST.

[ValidateAntiForgeryToken] // Atributo de segurança para previnir ataques CSRF.

public async Task<IActionResult> Create([Bind("Nome,DataNascimento,Telefone,Email")] Cliente cliente)

{

// O [Bind] é um atributo de segurança que especifica quais campos do formulário

// podem ser preenchidos pelo usuário para evitar "overposting".

try

{

// Verifica se os dados recebidos são válidos, de acordo com as anotações

// no Model (ex: [Required], [StringLength], etc).

if (ModelState.IsValid)

{

// Atribui a data atual ao campo de data de cadastro.

cliente.DataCadastro = DateTime.Now;

// Adiciona o novo objeto 'cliente' ao contexto do EF Core.

\_context.Add(cliente);

// Salva as mudanças no banco de dados de forma assíncrona.

await \_context.SaveChangesAsync();

// Redireciona o usuário de volta para a página de listagem (Index).

return RedirectToAction(nameof(Index));

}

}

catch (DbUpdateException /\* ex \*/)

{

// Poderíamos adicionar um log do erro aqui.

ModelState.AddModelError("", "Não foi possível salvar os dados. " +

"Tente novamente, e se o problema persistir, contate o administrador.");

}

// Se o ModelState não for válido, retorna para a mesma View,

// mas desta vez passando o objeto 'cliente' de volta.

// Isso permite que o formulário seja exibido novamente com os dados que o

// usuário já tinha preenchido e com as mensagens de erro de validação.

return View(cliente);

}

**5.2 - A View de Criação (Create.cshtml)**

Agora precisamos criar o arquivo HTML com o formulário. Os "Tag Helpers" do ASP.NET Core (atributos como asp-action, asp-for, asp-validation-for) nos ajudarão a conectar os campos do formulário diretamente às propriedades do nosso Model.

1. Dentro da pasta Views/Clientes, crie um novo arquivo chamado Create.cshtml.
2. Cole o seguinte código Razor neste arquivo.

HTML

@\* Local do Arquivo: /Views/Clientes/Create.cshtml \*@

@model ClinicaApp.Models.Cliente

@{

ViewData["Title"] = "Novo Cliente";

}

<h1>@ViewData["Title"]</h1>

<h4>Formulário de Cadastro</h4>

<hr />

<div class="row">

<div class="col-md-6">

@\* O Tag Helper asp-action="Create" instrui o formulário a enviar os dados para a ação Create [POST] \*@

<form asp-action="Create">

@\* Adiciona um sumário para erros de validação que não são específicos de um campo \*@

<div asp-validation-summary="ModelOnly" class="text-danger"></div>

<div class="form-group mb-3">

@\* Cria a <label> para o campo Nome \*@

<label asp-for="Nome" class="control-label"></label>

@\* Cria o <input> para o campo Nome, conectando-o à propriedade do Model \*@

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

@\* Espaço onde a mensagem de erro de validação para o Nome aparecerá \*@

<span asp-validation-for="Nome" class="text-danger"></span>

</div>

<div class="form-group mb-3">

<label asp-for="DataNascimento" class="control-label"></label>

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

<span asp-validation-for="DataNascimento" class="text-danger"></span>

</div>

<div class="form-group mb-3">

<label asp-for="Telefone" class="control-label"></label>

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

<span asp-validation-for="Telefone" class="text-danger"></span>

</div>

<div class="form-group mb-3">

<label asp-for="Email" class="control-label"></label>

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

<span asp-validation-for="Email" class="text-danger"></span>

</div>

<div class="form-group">

<input type="submit" value="Salvar" class="btn btn-primary" />

<a asp-action="Index" class="btn btn-secondary">Voltar para a Lista</a>

</div>

</form>

</div>

</div>

@\* A seção Scripts é usada para adicionar referências a scripts de validação no final da página,

o que permite validação no lado do cliente (no navegador) sem precisar recarregar a página. \*@

@section Scripts {

@{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}

}

**5.3 - Testando a Funcionalidade de Criação**

1. Execute o projeto
2. Navegue para a página de listagem (/clientes).
3. Clique no botão **"Novo Cliente"**. Você será levado ao formulário de cadastro.
4. Tente salvar com os campos obrigatórios em branco. Você verá as mensagens de erro que definimos no Model.
5. Preencha o formulário corretamente e clique em **"Salvar"**. Se tudo der certo, você será redirecionado para a página de listagem, e o novo cliente aparecerá na tabela!

**Capítulo 6: Editando Clientes Existentes (Manual)**

Assim como a criação, a edição também é um processo de duas etapas:

1. **Ação GET**: O usuário clica no link "Editar" de um cliente específico. O servidor busca os dados daquele cliente no banco e exibe o formulário de edição já preenchido.
2. **Ação POST**: O usuário altera os dados no formulário e clica em "Salvar". O navegador envia os dados atualizados para o servidor, que os valida e atualiza o registro correspondente no banco.

**6.1 - As Ações Edit no Controller**

Vamos adicionar os métodos GET e POST para a edição no nosso ClientesController.cs.

1. Abra o arquivo /Controllers/ClientesController.cs.
2. Adicione os dois métodos abaixo à classe ClientesController (pode ser após os métodos Create).

C#

// Adicione este código dentro da classe ClientesController

// GET: Clientes/Edit/5 (o '5' é um exemplo de ID)

// Ação que busca os dados do cliente e mostra o formulário de edição.

public async Task<IActionResult> Edit(int? id)

{

if (id == null)

{

// Retorna um erro 404 se nenhum ID for passado na URL.

return NotFound();

}

// Busca o cliente no banco de dados pelo ID recebido.

var cliente = await \_context.Clientes.FindAsync(id);

if (cliente == null)

{

// Retorna um erro 404 se o cliente com o ID especificado não for encontrado.

return NotFound();

}

// Retorna a View "Edit" e passa o objeto 'cliente' encontrado para ela.

return View(cliente);

}

// POST: Clientes/Edit/5

// Ação que recebe os dados do formulário de edição e os salva.

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(int id, [Bind("Id,Nome,DataNascimento,Telefone,Email,DataCadastro")] Cliente cliente)

{

// Verifica se o ID passado na URL é o mesmo do objeto cliente recebido.

// Medida de segurança para garantir que o cliente correto está sendo alterado.

if (id != cliente.Id)

{

return NotFound();

}

// Verifica se os dados atualizados são válidos.

if (ModelState.IsValid)

{

try

{

// Informa ao EF Core que o objeto 'cliente' foi modificado.

\_context.Update(cliente);

// Salva as alterações no banco de dados.

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

// Este erro pode ocorrer se outro usuário deletou o registro

// enquanto você estava editando.

if (!ClienteExists(cliente.Id))

{

return NotFound();

}

else

{

throw;

}

}

// Redireciona de volta para a lista de clientes.

return RedirectToAction(nameof(Index));

}

// Se o modelo não for válido, retorna para a mesma view de edição,

// passando o cliente de volta para que os erros possam ser exibidos.

return View(cliente);

}

// Método auxiliar privado para verificar se um cliente existe.

// Usado no 'catch' do método Edit [POST].

private bool ClienteExists(int id)

{

return \_context.Clientes.Any(e => e.Id == id);

}

*(Nota: Adicionamos um método privado ClienteExists no final, que é uma boa prática para ser usado pela lógica de edição).*

**6.2 - A View de Edição (Edit.cshtml)**

A view de edição é quase idêntica à de criação. A principal diferença é que ela já virá preenchida com os dados do cliente.

1. Dentro da pasta Views/Clientes, crie um novo arquivo chamado Edit.cshtml.
2. Cole o seguinte código Razor.

HTML

@\* Local do Arquivo: /Views/Clientes/Edit.cshtml \*@

@model ClinicaApp.Models.Cliente

@{

ViewData["Title"] = "Editar Cliente";

}

<h1>@ViewData["Title"]</h1>

<h4>@Model.Nome</h4>

<hr />

<div class="row">

<div class="col-md-6">

@\* O formulário agora aponta para a ação "Edit" \*@

<form asp-action="Edit">

<div asp-validation-summary="ModelOnly" class="text-danger"></div>

@\* É FUNDAMENTAL incluir um campo oculto para o Id.

Sem ele, o EF Core não saberá qual registro deve atualizar no banco. \*@

<input type="hidden" asp-for="Id" />

<input type="hidden" asp-for="DataCadastro" /> <div class="form-group mb-3">

<label asp-for="Nome" class="control-label"></label>

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

<span asp-validation-for="Nome" class="text-danger"></span>

</div>

<div class="form-group mb-3">

<label asp-for="DataNascimento" class="control-label"></label>

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

<span asp-validation-for="DataNascimento" class="text-danger"></span>

</div>

<div class="form-group mb-3">

<label asp-for="Telefone" class="control-label"></label>

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

<span asp-validation-for="Telefone" class="text-danger"></span>

</div>

<div class="form-group mb-3">

<label asp-for="Email" class="control-label"></label>

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

<span asp-validation-for="Email" class="text-danger"></span>

</div>

<div class="form-group">

<input type="submit" value="Salvar Alterações" class="btn btn-primary" />

<a asp-action="Index" class="btn btn-secondary">Voltar para a Lista</a>

</div>

</form>

</div>

</div>

@section Scripts {

@{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}

}

**6.3 - Testando a Funcionalidade de Edição**

1. Rode a aplicação com dotnet run.
2. Navegue para a lista de clientes (/clientes).
3. Clique no link **"Editar"** de um dos clientes que você cadastrou.
4. Você deverá ver o formulário de edição com todos os dados daquele cliente já preenchidos.
5. Altere alguma informação (por exemplo, o telefone) e clique em **"Salvar Alterações"**.
6. Você será redirecionado para a lista, e a informação do cliente deverá aparecer atualizada.

**Capítulo 7: Excluindo Clientes (Manual)**

A exclusão é uma ação crítica e, por segurança, não deve ser executada com um único clique. Por isso, também a dividimos em duas etapas:

1. **Ação GET**: O usuário clica no link "Excluir". O servidor busca os dados do cliente e exibe uma página de confirmação, perguntando "Você tem certeza?". Isso previne exclusões acidentais.
2. **Ação POST**: O usuário confirma a exclusão clicando no botão "Excluir" na página de confirmação. O servidor então remove o registro do banco de dados.

**7.1 - As Ações Delete no Controller**

Vamos adicionar os métodos para a exclusão ao nosso ClientesController.cs.

1. Abra o arquivo /Controllers/ClientesController.cs.
2. Adicione os dois métodos abaixo à classe ClientesController (pode ser após o método Edit).

C#

// Adicione este código dentro da classe ClientesController

// GET: Clientes/Delete/5

// Ação que busca os dados do cliente e mostra a página de confirmação.

public async Task<IActionResult> Delete(int? id)

{

if (id == null)

{

return NotFound();

}

// Busca o cliente para exibir seus dados na página de confirmação.

var cliente = await \_context.Clientes

.FirstOrDefaultAsync(m => m.Id == id);

if (cliente == null)

{

return NotFound();

}

// Retorna a View de confirmação, passando os dados do cliente.

return View(cliente);

}

// POST: Clientes/Delete/5

// Ação que efetivamente exclui o cliente do banco após a confirmação.

[HttpPost, ActionName("Delete")] // O ActionName("Delete") permite que a URL continue sendo /Clientes/Delete,

// mesmo que o nome do método seja DeleteConfirmed.

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

// Busca o cliente que será excluído.

var cliente = await \_context.Clientes.FindAsync(id);

if (cliente != null)

{

// Remove o cliente do contexto do EF Core.

\_context.Clientes.Remove(cliente);

}

// Salva a alteração (a remoção) no banco de dados.

await \_context.SaveChangesAsync();

// Redireciona de volta para a lista.

return RedirectToAction(nameof(Index));

}

**7.2 - A View de Exclusão (Delete.cshtml)**

Esta view é mais simples. Ela não tem campos de formulário para preencher, apenas exibe os dados do cliente e pede a confirmação.

1. Dentro da pasta Views/Clientes, crie um novo arquivo chamado Delete.cshtml.
2. Cole o seguinte código Razor.

HTML

@\* Local do Arquivo: /Views/Clientes/Delete.cshtml \*@

@model ClinicaApp.Models.Cliente

@{

ViewData["Title"] = "Excluir Cliente";

}

<h1>@ViewData["Title"]</h1>

<h3 class="text-danger">Você tem certeza que deseja excluir este registro?</h3>

<div>

<h4>Cliente: @Model.Nome</h4>

<hr />

<dl class="row">

<dt class = "col-sm-3">

@Html.DisplayNameFor(model => model.Nome)

</dt>

<dd class = "col-sm-9">

@Html.DisplayFor(model => model.Nome)

</dd>

<dt class = "col-sm-3">

@Html.DisplayNameFor(model => model.DataNascimento)

</dt>

<dd class = "col-sm-9">

@Html.DisplayFor(model => model.DataNascimento)

</dd>

<dt class = "col-sm-3">

@Html.DisplayNameFor(model => model.Telefone)

</dt>

<dd class = "col-sm-9">

@Html.DisplayFor(model => model.Telefone)

</dd>

<dt class = "col-sm-3">

@Html.DisplayNameFor(model => model.Email)

</dt>

<dd class = "col-sm-9">

@Html.DisplayFor(model => model.Email)

</dd>

</dl>

@\* O formulário aponta para a ação "Delete". Como é um POST, ele chamará o método DeleteConfirmed. \*@

<form asp-action="Delete">

@\* Campo oculto para o Id, para que o método POST saiba qual cliente excluir. \*@

<input type="hidden" asp-for="Id" />

<input type="submit" value="Excluir" class="btn btn-danger" /> |

<a asp-action="Index" class="btn btn-secondary">Voltar para a Lista</a>

</form>

</div>

**7.3 - Testando a Funcionalidade de Exclusão**

1. Rode a aplicação com dotnet run.
2. Navegue para a lista de clientes (/clientes).
3. Clique no link **"Excluir"** de um dos clientes.
4. Você será levado à página de confirmação, que exibirá os detalhes daquele cliente.
5. Clique no botão vermelho **"Excluir"**.
6. Você será redirecionado de volta para a lista, e o cliente não estará mais lá.

**Capítulo 8: Exibindo os Detalhes de um Cliente (Manual)**

A página de "Detalhes" é a mais simples de todas as nossas funcionalidades. Ela consiste em apenas uma etapa:

1. **Ação GET**: O usuário clica no link "Detalhes" de um cliente. O servidor busca os dados completos daquele cliente e os exibe em uma página de visualização (somente leitura).

Não há uma ação POST aqui, pois o objetivo é apenas visualizar, não enviar dados.

**8.1 - A Ação Details no Controller**

Vamos adicionar o método para a visualização de detalhes ao nosso ClientesController.cs.

1. Abra o arquivo /Controllers/ClientesController.cs.
2. Adicione o método abaixo à classe ClientesController. A lógica é muito similar à da ação GET de Edit e Delete.

C#

// Adicione este código dentro da classe ClientesController

// GET: Clientes/Details/5

// Ação que busca os dados do cliente e mostra a página de detalhes.

public async Task<IActionResult> Details(int? id)

{

if (id == null)

{

return NotFound();

}

// Busca o cliente no banco de dados pelo ID.

// O FirstOrDefaultAsync é uma boa escolha aqui.

var cliente = await \_context.Clientes

.FirstOrDefaultAsync(m => m.Id == id);

if (cliente == null)

{

// Retorna 404 se o cliente não for encontrado.

return NotFound();

}

// Retorna a View "Details" e passa o objeto 'cliente' para ela.

return View(cliente);

}

**8.2 - A View de Detalhes (Details.cshtml)**

Esta view é focada em apresentar os dados de forma clara e legível.

1. Dentro da pasta Views/Clientes, crie um novo arquivo chamado Details.cshtml.
2. Cole o seguinte código Razor.

HTML

@\* Local do Arquivo: /Views/Clientes/Details.cshtml \*@

@model ClinicaApp.Models.Cliente

@{

ViewData["Title"] = "Detalhes do Cliente";

}

<h1>@ViewData["Title"]</h1>

<div>

<h4>Informações Completas</h4>

<hr />

@\* Usamos uma "Definition List" (<dl>) para exibir os dados de forma organizada \*@

<dl class="row">

<dt class = "col-sm-3">

@Html.DisplayNameFor(model => model.Nome)

</dt>

<dd class = "col-sm-9">

@Html.DisplayFor(model => model.Nome)

</dd>

<dt class = "col-sm-3">

@Html.DisplayNameFor(model => model.DataNascimento)

</dt>

<dd class = "col-sm-9">

@Html.DisplayFor(model => model.DataNascimento)

</dd>

<dt class = "col-sm-3">

@Html.DisplayNameFor(model => model.Telefone)

</dt>

<dd class = "col-sm-9">

@Html.DisplayFor(model => model.Telefone)

</dd>

<dt class = "col-sm-3">

@Html.DisplayNameFor(model => model.Email)

</dt>

<dd class = "col-sm-9">

@Html.DisplayFor(model => model.Email)

</dd>

<dt class = "col-sm-3">

@Html.DisplayNameFor(model => model.DataCadastro)

</dt>

<dd class = "col-sm-9">

@Html.DisplayFor(model => model.DataCadastro)

</dd>

</dl>

</div>

<div>

@\* Link para a ação de Edição, passando o ID do cliente atual \*@

<a asp-action="Edit" asp-route-id="@Model?.Id" class="btn btn-primary">Editar</a> |

<a asp-action="Index" class="btn btn-secondary">Voltar para a Lista</a>

</div>

**8.3 - Testando a Funcionalidade de Detalhes**

1. Rode a aplicação com dotnet run.
2. Navegue para a lista de clientes (/clientes).
3. Clique no link **"Detalhes"** de um dos clientes.
4. Você será levado para a página de detalhes, que exibirá todas as informações daquele cliente, incluindo a Data de Cadastro, que não mostramos na lista principal.

**Parabéns!** Nós completamos manualmente todo o ciclo CRUD (Create, Read, Update, Delete). Você construiu, linha por linha, um sistema web funcional.

**Capítulo 9: Melhorando a Aparência e a Navegação**

Neste momento, nossa aplicação é totalmente funcional, mas para acessarmos a área de clientes, precisamos digitar o endereço /clientes manualmente na URL. Além disso, a aparência geral é bem básica. Vamos resolver isso.

**9.1 - Adicionando o Link no Menu Principal**

Para facilitar o acesso, vamos adicionar um link permanente na barra de navegação superior do site. Faremos isso editando o arquivo de layout principal, que serve como um "molde" para todas as páginas da nossa aplicação.

1. Abra o arquivo de layout, que se encontra em: Views/Shared/\_Layout.cshtml.
2. Procure pela tag <ul> que tem a classe navbar-nav. Dentro dela, você verá os itens do menu para "Home" e "Privacy".
3. Adicione um novo item de lista (<li>) para "Clientes", como mostrado abaixo:

HTML

@\* Local do Arquivo: /Views/Shared/\_Layout.cshtml \*@

<header>

<nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">

<div class="container-fluid">

<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">ClinicaApp</a>

<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"

aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">

<ul class="navbar-nav flex-grow-1">

<li class="nav-item">

<a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>

</li>

@\* --- INÍCIO DA ADIÇÃO --- \*@

<li class="nav-item">

<a class="nav-link text-dark" asp-area="" asp-controller="Clientes" asp-action="Index">Clientes</a>

</li>

@\* --- FIM DA ADIÇÃO --- \*@

<li class="nav-item">

<a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>

</li>

</ul>

</div>

</div>

</nav>

</header>

**O que fizemos?**

* asp-controller="Clientes": Informa que o link deve apontar para o ClientesController.
* asp-action="Index": Informa que o link deve chamar a ação Index daquele controller.

Agora, o link "Clientes" aparecerá no menu superior em todas as páginas do site.

**9.2 - Pequena Melhoria Visual na Página de Listagem**

Vamos adicionar um "card" em volta da nossa tabela de clientes para dar um acabamento visual mais profissional, utilizando as classes do Bootstrap 5, que já vem incluído no projeto.

1. Abra o arquivo Views/Clientes/Index.cshtml.
2. Envolva o conteúdo da página (a partir do <h1>) com as divs de card, como no exemplo abaixo:

HTML

@\* Local do Arquivo: /Views/Clientes/Index.cshtml \*@

@model IEnumerable<ClinicaApp.Models.Cliente>

@{

ViewData["Title"] = "Gerenciamento de Clientes";

}

@\* --- INÍCIO DA ADIÇÃO --- \*@

<div class="card shadow-sm">

<div class="card-header">

<h1 class="card-title">@ViewData["Title"]</h1>

</div>

<div class="card-body">

@\* --- FIM DA ADIÇÃO --- \*@

<p>

<a asp-action="Create" class="btn btn-primary">Novo Cliente</a>

</p>

<table class="table table-striped table-hover">

<thead>

@\* (O conteúdo da tabela permanece o mesmo) \*@

<tr>

<th>@Html.DisplayNameFor(model => model.Nome)</th>

<th>@Html.DisplayNameFor(model => model.Email)</th>

<th>@Html.DisplayNameFor(model => model.Telefone)</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model) {

<tr>

<td>@Html.DisplayFor(modelItem => item.Nome)</td>

<td>@Html.DisplayFor(modelItem => item.Email)</td>

<td>@Html.DisplayFor(modelItem => item.Telefone)</td>

<td>

<a asp-action="Edit" asp-route-id="@item.Id" class="btn btn-secondary btn-sm">Editar</a> |

<a asp-action="Details" asp-route-id="@item.Id" class="btn btn-info btn-sm">Detalhes</a> |

<a asp-action="Delete" asp-route-id="@item.Id" class="btn btn-danger btn-sm">Excluir</a>

</td>

</tr>

}

</tbody>

</table>

@\* --- INÍCIO DA ADIÇÃO --- \*@

</div>

</div>

@\* --- FIM DA ADIÇÃO --- \*@

**9.3 - Testando as Melhorias**

1. Rode a aplicação com dotnet run (ou dotnet watch run para aplicar alterações automaticamente enquanto o código é salvo).
2. Você verá imediatamente o link **"Clientes"** no menu superior. Clique nele.
3. A página de listagem agora terá uma aparência mais organizada, com o título e a tabela dentro de um painel com uma leve sombra.

Com essas melhorias, a experiência de uso da aplicação já é muito superior. O próximo passo é adicionar uma funcionalidade poderosa: um campo de busca para filtrar clientes por nome.

**Capítulo 10: Adicionando Busca de Clientes**

Neste capítulo, vamos adicionar um campo de busca na nossa página de listagem para que o usuário possa filtrar clientes pelo nome. Isso envolve uma pequena alteração na View para adicionar o formulário de busca e uma modificação na ação Index do Controller para processar o filtro.

**10.1 - Modificando a View Index.cshtml**

Primeiro, vamos adicionar o campo de texto e o botão de busca na nossa página de listagem.

1. Abra o arquivo Views/Clientes/Index.cshtml.
2. Adicione o código do formulário (<form>) logo acima do parágrafo que contém o botão "Novo Cliente".

HTML

@\* Local do Arquivo: /Views/Clientes/Index.cshtml \*@

@model IEnumerable<ClinicaApp.Models.Cliente>

@{

ViewData["Title"] = "Gerenciamento de Clientes";

}

<div class="card shadow-sm">

<div class="card-header">

<h1 class="card-title">@ViewData["Title"]</h1>

</div>

<div class="card-body">

@\* --- INÍCIO DA ADIÇÃO DO FORMULÁRIO DE BUSCA --- \*@

<form asp-controller="Clientes" asp-action="Index" method="get" class="mb-3">

<div class="input-group">

<input type="text" name="searchString" class="form-control" placeholder="Digite o nome para buscar..." value="@ViewData["CurrentFilter"]">

<div class="input-group-append">

<button type="submit" class="btn btn-outline-secondary">Buscar</button>

</div>

</div>

</form>

@\* --- FIM DA ADIÇÃO DO FORMULÁRIO DE BUSCA --- \*@

<p>

<a asp-action="Create" class="btn btn-primary">Novo Cliente</a>

</p>

@\* O resto do arquivo (a tabela) continua igual \*@

<table class="table table-striped table-hover">

@\* ... \*@

</table>

</div>

</div>

**O que fizemos?**

* <form asp-controller="Clientes" asp-action="Index" method="get">: Criamos um formulário que, quando submetido, fará uma requisição GET para a ação Index do ClientesController. Usar GET faz com que o termo de busca apareça na URL (ex: /Clientes?searchString=ana), o que é ótimo para buscas.
* <input type="text" name="searchString" ...>: Este é o campo onde o usuário digitará. O name="searchString" é crucial, pois é o nome do parâmetro que nosso Controller irá receber.
* value="@ViewData["CurrentFilter"]": Isso faz com que, após a busca, o campo continue preenchido com o termo que o usuário pesquisou. Vamos configurar o ViewData no controller a seguir.

**10.2 - Modificando o Controller ClientesController.cs**

Agora, vamos alterar a ação Index para que ela possa receber o termo de busca e filtrar os resultados.

1. Abra o arquivo /Controllers/ClientesController.cs.
2. Substitua o método Index existente pelo código abaixo.

C#

// Substitua o método Index() existente por este

// GET: /Clientes ou /Clientes?searchString=ana

public async Task<IActionResult> Index(string searchString)

{

// Passa o termo de busca (se houver) para a View, para que o campo de busca

// permaneça preenchido após a submissão do formulário.

ViewData["CurrentFilter"] = searchString;

// 1. Começamos com uma consulta base que seleciona todos os clientes.

// Usamos 'IQueryable' porque a consulta ainda não foi executada no banco.

var clientes = from c in \_context.Clientes

select c;

// 2. Verificamos se o parâmetro searchString não é nulo ou vazio.

if (!String.IsNullOrEmpty(searchString))

{

// 3. Se houver um termo de busca, modificamos a consulta para filtrar

// os clientes cujo Nome contenha o termo de busca.

// O 'Contains' é traduzido para uma cláusula 'LIKE' em SQL.

clientes = clientes.Where(s => s.Nome.Contains(searchString));

}

// 4. Finalmente, executamos a consulta (agora com o filtro, se aplicável)

// e passamos a lista resultante para a View.

return View(await clientes.AsNoTracking().ToListAsync());

}

*(Nota: Adicionei .AsNoTracking() à consulta. É uma otimização de performance para consultas de somente leitura, como esta).*

**10.3 - Testando a Busca**

1. Rode a aplicação com dotnet run ou dotnet watch run.
2. Vá para a página de **Clientes**.
3. Você verá o novo campo de busca acima do botão "Novo Cliente".
4. Digite parte do nome de um cliente que você já cadastrou e clique em **"Buscar"**.
5. A lista deverá ser atualizada, mostrando apenas os clientes que correspondem ao seu termo de busca. O campo de busca também permanecerá preenchido.
6. Para ver todos os clientes novamente, apague o termo do campo de busca e clique em "Buscar".

**Conclusão e Próximos Passos**

**Parabéns!** Você construiu, do início ao fim, uma aplicação web CRUD funcional e robusta, escrevendo todo o código manualmente. Você agora tem uma base sólida em ASP.NET Core MVC e Entity Framework Core.