SETUP BÁSICO DO PROJETO

PASSO 01 - CRIANDO O PROJETO

Criando as pastas:

```
> mkdir Tarefas
> cd Tarefas
> mkdir src
```

Criando a solução do projeto .Net:

```
> dotnet new sln
```

- Note que irá criar a solução com o mesmo nome da pasta onde o comando está sendo executado.

Abrir o Visual Studio Code no diretório "Tarefas":

Na primeira vez que entrar na pasta com a solução o Code baixará as dependências para trabalhar com C#.

PASSO 02 - CONFIGURAÇÕES SDK

O Arquivo “global.json” deve ficar na raiz do projeto, no mesmo nível da solução. Para criar o “global.json” com o último “sdk” instalado.

As configurações variam de acordo com o SDK instalado, por isso é importante checar qual versão temos disponível no nosso ambiente de desenvolvimento. Para isso execute:

```
> dotnet --list-sdks
```

Caso existem mais de um SDK instalado, você pode selecionar qual deseja utilizar dessa forma:

```
> dotnet new globaljson --sdk-version <VERSÃO>
```

Ou apenas e deixar o dotnet selecionar o default:

```
> dotnet new globaljson
```

Agora devemos verificar o arquivo gerado.

- Por que criar o arquivo “global.json”?

  Para definir a versão do “sdk” que será utilizado na solução e em todos os projetos que serão criados dentro da pasta da solução.

- Devemos fazer mais alguma alteração em global.json?

  Sim, por causa da nossa estrutura de pastas src (código fonte) e test (código de teste), precisamos incluir esta estrutura no arquivo, com a propriedade projects.

Edite o arquivo da seguinte maneira:

- Inclua linha "projects", deixando o arquivo desta forma:

```
{
  "projects" : ["src"],
  "sdk": {
    "version": "7.0.100"
  }
}
```

PASSO 03 - CRIANDO O PROJETO MVC

Criando o diretório com o nome do projeto web:

```
> cd src
> mkdir Tarefas.Web
> cd Tarefas.Web
> dotnet new mvc
```

Apesar de ter criado o projeto "Tarefas.Web" na pasta "src", esse projeto ainda não é conhecido pela solução.

- Incluindo o projeto na solução

Devemos estar no diretório da solução, caso esteja dentro do diretório "src" suba um nível, caso esteja no diretório "tarefas.web" suba dois níveis e então execute o comando:

```
> dotnet sln add ./src/Tarefas.Web/Tarefas.Web.csproj
```

Pode ser necessário fechar e abrir o Code para que a inclusão de projetos seja reconhecida.

PASSO 03 - EXECUTANDO O PROJETO PELA PRIMEIRA VEZ

Devemos estar no diretório do projeto "tarefas.web".

```
> cd src/Tarefas.Web
```

E então executar:

```
> dotnet run
```

A esta altura, você já deve conseguir visualizar uma pagina gerada pelo aspnet. Porém, se o navegador apresentar alguma mensagem de erro por conta do certificado HTTPS, pare a execução do projeto e faça o seguinte procedimento:

```
> dotnet dev-certs https --trust // no windows
> dotnet dev-certs https // no linux
```

E então execute novamente a aplicação.

```
> dotnet run
```

Basta abrir o navegador e acessar a url https://localhost:{porta} (https://localhost:%7Bporta%7D) // porta é um número randômico gerado durante o build

Se você chegou até aqui. Finalizamos nossa missão de Setup do projeto.

NOSSA PRIMEIRA FUNCIONALIDADE

PASSO 01 - RECONHECENDO ARQUIVOS GERADOS AUTOMATICAMENTE

Dentr os arquivos gerados automaticamente pelo aspnet na criação do projeto MVC estão estes:

- /Controllers/HomeController.cs
- /Views/Home
- /Views/Home/Index.cshtml
- /Views/Home/Privacy.cshtml

PASSO 02 - NOSSO PRIMEIRA FUNCIONALIDADE

Vamos criar na pasta 'Controllers' um arquivo chamado: "TarefaController.cs". E dentro, implemente uma 'Action' chamada "Create" retornando uma "View".

- Lembre-se que você pode usar a lâmpada amarela no canto esquerdo da linha para solucionar alguns 'problemas' comuns para acelerar o desenvolvimento.

```
using Microsoft.AspNetCore.Mvc;

namespace Tarefas.Web.Controllers
{
    public class TarefaController : Controller
    {
        public IActionResult Create ()
        {
            return View();
        }
    }
}
```

Na pasta "Views" vamos criar a pasta com o nome "Tarefa".

- Lembrete: A pasta com as "Views" de um "Controller" devem possuir o mesmo nome do "Controller", isso é um padrão do Framework, simplesmente não funciona se não for seguido.

Dentro da pasta "Tarefa", criaremos um arquivo chamado: "Create.cshtml" (que tem o mesmo nome da 'Action' que criamos na 'Controller').

Neste arquivo que acabamos de criar, vamos adicionar um texto apenas para identificarmos que o arquivo está sendo carregado corretamente.

```
<form asp-action="Create">
    <input asp-for="Titulo" />
    <input asp-for="Descricao" />
    <input type="submit" />
</form>
```

Neste momento, você pode executar o projeto e testar executando:

```
> dotnet run
```

Ao abrir o navegador acesse: http://localhost:{porta}}/Tarefa/Create (http://localhost:%7Bporta%7D%7D/Tarefa/Create)

Se tudo der certo até aqui, você deve ver o texto "Criar nova tarefa" na tela. O que significa que a amarração entre View > Controller > Action está correta e podemos seguir o desenvolvimento normalmente.

PASSO 03 - CRIANDO O MODELO DE DADOS

Na a pasta "Models", selecione um arquivo "Tarefa.cs" e inclua as propriedades: 'Titulo', 'Descricao' e 'Concluida'.

```
public string Titulo { get; set; }
public string Descricao { get; set; }
public string Concluida { get; set; }
```

Ficando desta forma:

```
namespace Tarefas.Web.Models
{
    public class Tarefa
    {
        [DisplayName("Titulo")]
        public string? Titulo { get; set; }

        [DisplayName("Descricao")]
        public string? Descricao { get; set; }

        [DisplayName("Concluída")]
        public bool IsDone { get; set; }
    }
}
```

- Lembrem-se que existem várias formas de utilizar as DataAnnotations. Aproveite para pesquisar sobre elas.

PASSO 04 - CONFIGURANDO A VIEW DE TAREFAS

Primeiramente devemos adicionar ao arquivo "Create.cshtml" a referência do 'Model' que criamos para as tarefas.

```
@model Tarefas.Web.Models.Tarefa
```

Em seguida podemos começar a montar o formulário que irá capturar as informações que o usuário irá digitar e enviar para o servidor processar.

```
<form asp-action="Create">
</form>
```

Inclua os campos "Titulo" e "Descricao" por enquanto.

```
<form asp-action="Create">
    <input asp-for="Titulo" />
    <input asp-for="Descricao" />
</form>
```

Em seguida, inclua um botão que submeterá o formulário para a "Controller".

```
<form asp-action="Create">
    <input asp-for="Titulo" />
    <input asp-for="Descricao" />
    <input type="submit" />
</form>
```

Para submeter o Formulário criado na "TarefaController" precisamos criar um método que receberá os dados do formulário. Vamos abrir o "TarefaController.cs" e acrescentar o método 'Post', seguindo os passos:

O método "Create" é idêntico ao primeiro, com a diferença de estar recebendo como parâmetro a "Model" "Tarefa".

```
[HttpPost]
public IActionResult Create(Tarefa tarefa)
{
    return view();
}
```

PASSO 05 - DEBUGANDO PELA PRIMEIRA VEZ

Crie um "breakpoint" na linha que inicia o método que acabamos de criar. Em seguida, no lado esquerdo do "Code", selecione "Executar e Depurar" e clique no botão com o mesmo nome. Depois, selecione a opção ".NET Core", para criar o arquivo "launch.json".

Uma vez que o arquivo seja criado, você poderá iniciar a aplicação em modo DEBUG, clicando no botão ".NET Core Launch (web)".

Agora vá no navegador e acesse a URL "/Tarefa/Create".

Preencha os dados do formulário e clique em 'enviar'.

- Neste momento o breakpoint deve ser acionado e você poderá inspecionar o ponteiro da aplicação e analisar as variáveis durante a execução da aplicação.
- A qualquer momento, você pode clicar no botão 'Interromper' e encerrar a execução da aplicação.

Se você conseguiu debugar a aplicação, então concluímos mais este ponto da nossa aplicação.

MELHORIAS DE USABILIDADE

PASSO 01 - AJUSTANDO A NAVEGACAO

Na a pasta "Views > Shared", selecione o arquivo "_Layout.cshtml" ele contém o menu e a estrutura da página que herdamos quando acessamos a página "Tarefas > Create.cshtml".

Inclua mais um link no menu horizontal, que apontará para a "Action" "Create" em "TarefaController". Remova os demais links que não precisamos.

Execute a aplicação e já podemos acessar a tela pelo menu.

```
> dotnet run
```

PASSO 02 - MELHORANDO O FORMULÁRIO

Vamos adicionar labels para identificar os campos do formulário para os usuários.

Inclua as "divs" e "classes" do "Bootstrap" para criar um layout profissional para o formulário.

Para realizar as alterações de front-end, baseadas em formatação html e css, uma boa dica é iniciar a aplicação em modo "watch", que atualiza a página ao salvar o arquivo, e agiliza a visualização das alterações do front-end enquanto você vai ajustando cada componente e classe css.

```
> dotnet watch run
```

PASSO 03 - TELA DE LISTAGEM DE TAREFAS

Na a pasta "Controllers", selecione o arquivo "TarefaController" e inclua um novo método "Index".

Neste método implemente temporariamente uma lista fixa de tarefas e retorne para a 'View' esta lista.

```
public IActionResult Index()
{
    var listaDeTarefas = new List<Tarefa>()
    {
        new Tarefa() { Titulo = "Escovar os dentes" },
        new Tarefa() { Titulo = "Arrumar a cama" },
        new Tarefa() { Titulo = "Por o lixo para fora", Descricao = "somente terças e quintas" }
    };
    return View(listaDeTarefas);
}
```

Em seguida, já podemos criar uma tabela em "html" para exibir essa lista de "Model > Tarefa".

- Note que criamos uma tabela com 3 colunas cada uma com o nome de uma propriedade do "Model > Tarefa" e na seção "tbody" do "Index.cshtml", estamos listando o conteúdo da lista.
- Note que antes dos comandos em "C#" tem a "@" que identifica a "string" como um código "C#" e não "html".

Salve todos os arquivos e execute a aplicação. Ainda não criamos o menu então teremos que acessar pela URL.

PASSO 04 - REFACTORAÇÃO

Note que temos uma repetição de títulos em nossas duas "Views" "Tarefa > Index.cshtml" e "Tarefa > Create.cshtml", todas elas possuem "Título" e "Descrição". Caso fosse necessário mudar um desses títulos iriamos ter que alterar em todas as "Views", apesar de só existirem duas esse problema poderia ser maior caso existissem várias "Views", além do problema de esquecer de atualizar uma delas caso todas tivessem que ser alteradas.

Lembre-se sempre: "NÃO SE REPITA", esse é o conceito DRY.

Vamos resolver isso decorando as propriedades com "DisplayName".

```
> @Html.DisplayNameFor(model => model.Titulo)
> @Html.DisplayNameFor(model => model.Descricao)
```

Em seguida, vamos alterar a navegação do menu para termos acesso tanto à página de listagem de tarefas como a de criar novas tarefas.

Na a pasta "Views > Shared", selecione o arquivo "_Layout.cshtml" e altere o menu para atingir este objetivo.

PASSO 01 - PÁGINA DE DETALHES

Para iniciar o desenvolvimento da página de detalhes, precisamos adicionar um ID único ao modelo de dados, para permitir identificar cada registro de forma individual.

Vamos editar o arquivo "Tarefa.cs" dentro de 'Models'.

```
> public int Id { get; set; }
```

Vamos incluir esta propriedade na página de listagem, na forma de um link para edição.

```
<td>
    <a asp-controller="Tarefa" asp-action="Details" asp-route-id="@item.Id">Detalhes</a>
</td>
```

Agora precisamos alterar a lista que criamos temporariamente com os dados das tarefas para adicionar valores à propriedade Id. Para isso, vamos na 'TarefaController.cs' e editamos a lista de tarefas, informando um Id diferente para cada objeto na lista.

```
{
    new Tarefa() { Id = 1, Titulo = "Escovar os dentes" },
    new Tarefa() { Id = 2, Titulo = "Arrumar a cama" },
    new Tarefa() { Id = 3, Titulo = "Por o lixo para fora", Descricao = "somente terças e quintas" }
};
```

Em seguida, precisamos passar essa lista para o construtor do 'Controller' e deixar lá o preenchimento da lista de tarefas. Desta forma, a propriedade ficará acessível para todos os métodos da classe.

Primeiro crie uma propriedade na classe:

```
> List<Tarefa> listaDeTarefas = new List<Tarefa>();
```

Crie um método de construtor na classe:

```
> public TarefaController() { ... }
```

Passe o preenchimento da lista para o método construtor:

```
public TarefaController()
{
    listaDeTarefas = new List<Tarefa>()
    {
        new Tarefa() { Id = 1, Titulo = "Escovar os dentes" },
        new Tarefa() { Id = 2, Titulo = "Arrumar a cama" },
        new Tarefa() { Id = 3, Titulo = "Por o lixo para fora", Descricao = "somente terças e quintas" }
    };
}
```

Agora podemos criar uma 'Action' 'Details' que receba o Id de uma tarefa:

```
> public IActionResult Details(int id) { ... }
```

Dentro do método de Details, podemos utilizar o Id passado por parâmetro para encontrar a tarefa da lista que será visualizada. Para isso, usamos o Linq (sistema de pesquisa em listas de objetos).

- Aproveite para pesquisar um pouco sobre o Linq.

```
public IActionResult Details(int id)
{
    var tarefa = listaDeTarefas.Find(tarefa => tarefa.Id == id);
    return View(tarefa);
}
```

Agora precisamos criar a 'View' que irá exibir estas informações.

Podemos neste momento copiar a View 'Create.cshtml' e renomeá-la para 'Details.cshtml'.

Execute a aplicação e verifique se os dados de uma tarefa são carregados no formulário conforme planejado.

PASSO 01 - NOVOS PROJETOS

A patir deste momento, vamos precisar incluir mais dois projetos a nossa aplicação, criar as referências entre eles e instalar suas dependências. Até agora só temos o projeto "Tarefas.Web" criado e já estamos acostumados com sua estrutura MVC.

Agora vamos incluir mais dois projetos a nossa solução, um DAO (Objetos para acesso a dados) e um DTO (Objetos para transferência de dados).

Ao final, a "Controller" irá conhecer o "Tarefas.DTO" e o "Tarefas.DAO", durante a criação de um registro a "Controller" irá preencher um "Tarefas.DTO" com os dados da "ViewModel" e enviar para a "Tarefas.DAO" que será responsável por gravar o registro no banco de dados.

Vá até a pasta "src" do projeto e execute o seguinte comando:

```
> dotnet new classlib -n Tarefas.DTO
```

Este comando, criará o projeto "Tarefas.DTO", que irá conter o objetos que serão gravados no banco de dados. Em seguida, inclua a referência no projeto "Tarefas.Web", para que as "Controllers" possam utilizar os "DTOs" que iremos criar.

```
> dotnet add .\Tarefas.Web\Tarefas.Web.csproj reference .\Tarefas.DTO\Tarefas.DTO.csproj
```

Crie o projeto "Tarefas.DAO", que irá fazer o acesso ao banco de dados, gravar e recuperar os objetos "DTOs".

```
> dotnet new classlib -n Tarefas.DAO
```

Inclua a referência no projeto "Tarefas.Web" para o projeto "Tarefas.DAO"

```
> dotnet add .\Tarefas.Web\Tarefas.Web.csproj reference .\Tarefas.DAO\Tarefas.DAO.csproj
```

Inclua a referência no projeto "Tarefas.DAO" para o projeto "Tarefas.DTO"

```
> dotnet add .\Tarefas.DAO\Tarefas.DAO.csproj reference .\Tarefas.DTO\Tarefas.DTO.csproj
```

Na a pasta "DAO", inclua as referências para o Dapper e SQLLite:

```
> cd .\Tarefas.DAO\
> dotnet add package Dapper --version 2.0.78
> dotnet add package System.Data.SQLite.Core --version 1.0.113.7
```

BANCO DE DADOS

PASSO 01 - INSERINDO NO BANCO DE DADOS

Até este momento, estamos trabalhando apenas com dados em memória e armazenados diretamente nas classes controladoras, mas não iremos conseguir chegar muito longe assim. Isso foi importante para entendermos o processo de construção das views e os métodos de POST e GET. Porém, a partir de agora, vamos trazer a introdução dos mecanismos de banco de dados propriamente ditos. Mesmo assim, ainda será um mecanismo simples para manter o objetivo didático da implementação.

Clique com o botão direito na pasta do projeto "Tarefas.DAO" e clique em "novo arquivo", nomeie a classe como: 'TarefaDAO'.

Inclua as referências para as dependências na seção "using".

```
using Dapper;
using System;
using System.Data.SQLite;
using System.IO;
using System.Linq;
using Tarefas.DTO;
```

Em seguida, vamos criar as estruturas de conexão do banco de dados.

```
namespace Tarefas.DAO
{
    public class TarefaDAO
    {
        private string DataSourceFile => Environment.CurrentDirectory + "TarefasDB.sqlite";
        public SQLiteConnection Connection => new SQLiteConnection("DataSource="+ DataSourceFile);
    }
}
```

Note que temos o "DataSourceFile" que contém a pasta atual do projeto e o nome do banco de dados, nesse caso "TarefasDB.sqlite" poderíamos ter utilizado quando outro nome para o arquivo, mas para manter o padrão utilizei o mesmo nome da solução "Tarefas" seguido de um sufixo "DB".

Crie o construtor que verificará se o arquivo do banco de dados já foi criado, caso não tenha sido ele será criado no momento em que a classe for instanciada.

```
public TarefaDAO()
{
    if(!File.Exists(DataSourceFile))
    {
        CreateDatabase();
    }
}
```

Implemente o método "CreateDatabase" que irá criar o banco de dados com base no script.

```
private void CreateDatabase()
{
    using(var con = Connection)
    {
        con.Open();
        con.Execute(
            @"CREATE TABLE Tarefa
            (
                Id          integer primary key autoincrement,
                Titulo      varchar(100) not null,
                Descricao   varchar(100) not null,
                Concluida   bool not null
            )"
        );
    }
}
```

Note que criamos uma tabela em "SQL" chamada "Tarefa" que possui as mesmas propriedades que nossa classe "Tarefa", e também possui e exatamente os mesmos tipos.

No projeto "DTO", crie o "TarefaDTO" com as mesmas propriedades de "Tarefa". Ela deve ficar dessa forma:

```
using System;

namespace Tarefas.DTO
{
    public class TarefaDTO
    {
        public int Id { get; set; }

        public string? Titulo { get; set; }

        public string? Descricao { get; set; }

        public bool Concluida { get; set; }
    }
}
```

No projeto "Tarefa.DAO", selecione o arquivo "TarefaDAO" e crie o método para salvar os dados da classe "TarefaDTO".

```
public void Criar(TarefaDTO tarefa)
{
    using (var con = Connection)
    {
        con.Open();
        con.Execute(
            @"INSERT INTO Tarefa
            (Titulo, Descricao, Concluida) VALUES
            (@Titulo, @Descricao, @Concluida);", tarefa
        );
    }
}
```

Note que não estamos passando o "Id" como parâmetro no "INSERT", porque definimos o "Id" como "AUTOINCREMENT", então ele será criado assim que o registro for gravado no banco de dados.

PASSO 02 - MODIFICANDO A PÁGINA PARA UTILIZAR O BANCO

No projeto "Tarefas.Web", abra a pasta "Controllers", selecione o arquivo "TarefaController" e altere o método "Post" do "Create".

```
[HttpPost]
public IActionResult Create(Tarefa tarefa)
{
    var tarefaDTO = new TarefaDTO
    {
        Titulo = tarefa.Titulo,
        Descricao = tarefa.Descricao,
        Concluida = tarefa.Concluida
    };

    var tarefaDAO = new TarefaDAO();
    tarefaDAO.Criar(tarefaDTO);

    return View();
}
```

Note que temos dois métodos "Create" o primeiro é "HttpGet", que pode ser padrão não precisa ser escrito, o segundo é "HttpPost" que recebe como parâmetro a "Model" "Tarefa" passada pela "View". Tivemos que passar os dados do "Tarefa" para o "TarefaDTO" porque a classe "TarefaDAO" só conhece ou seja só referência o projeto "DTO". Instanciamos a "DAO" e chamamos o método "Criar" passando o "DTO" como parâmetro.

Execute a aplicação e inclua uma nova 'Tarefa'.

```
> dotnet run
```

Volte para a solução e veja que o arquivo "TarefasDB.sqlite" foi criado na pasta "Tarefas.Web".

LISTAGEM

PASSO 01 - LISTANDO TAREFAS

Agora vamos criar os métodos para recuperar os registros de "Tarefas" do banco de dados, um dos métodos retornará todos os "Tarefas" que será utilizado na listagem e o outro método retornará por "Id" de forma individual, que será utilizado para visualização e edição.

No projeto "Tarefas.DAO", selecione o arquivo "TarefaDAO" e crie o método para consultar a lista de Tarefas.

```
public List<TarefaDTO> Consultar()
{
    using(var con = Connection)
    {
        con.Open();
        var result = con.Query<TarefaDTO>(
            @"SELECT Id, Titulo, Descricao, Concluida FROM Tarefa"
        ).ToList();
        return result;
    }
}
```

Note que será necessário incluir a referência a "System.Collections.Generic" na seção "using", porque estamos retornando uma "List".

No projeto "Tarefas.Web", selecione o arquivo "TarefaController" e reescreva o método "Index".

```
public IActionResult Index()
{
    var tarefaDAO = new TarefaDAO();
    var listaDeTarefasDTO = tarefaDAO.Consultar();

    var listaDeTarefa = new List<Tarefa>();

    foreach (var tarefaDTO in listaDeTarefaDTO)
    {
        listaDeTarefa.Add(new Tarefa()
        {
            Id = tarefaDTO.Id,
            Titulo = tarefaDTO.Titulo,
            Descricao = tarefaDTO.Descricao,
            Concluida = tarefaDTO.Concluida
        });
    }

    return View(listaDeTarefa);
}
```

Note que após consultar a lista de "TarefaDTO" precisamos converte-los em uma lista de "Tarefas", porque a "View > Index" espera receber uma lista de objetos do tipo "Tarefa" e não do tipo "TarefaDTO".

Execute a aplicação e veja a lista de tarefas.

PASSO 02 - LISTANDO UMA TAREFA ESPECÍFICA

No projeto "Tarefas.DAO", selecione o arquivo "TarefaDAO" e crie o método para consultar a tarefa por "Id".

Nesse método utilizamos o parâmetro "Id" e na clausula "WHERE" do "SELECT", diferente do método anterior que retornava todos os registros. Esse método será uma sobrecarga do primeiro método "Consultar" porque apesar de ter o mesmo nome sua assinatura é diferente por causa do parâmetro "Id".

```
public TarefaDTO Consultar(int id)
{
    using (var con = Connection)
    {
        con.Open();
        TarefaDTO result = con.Query<TarefaDTO>
        (
            @"SELECT Id, Titulo, Descricao, Concluida FROM Tarefa
            WHERE Id = @Id", new { id }
        ).FirstOrDefault();
        return result;
    }
}
```

No projeto "Tarefas.Web", selecione o arquivo "TarefaController" e reescreva o método "Details".

```
public IActionResult Details(int id)
{
    var tarefaDAO = new TarefaDAO();
    var tarefaDTO = tarefaDAO.Consultar(id);

    var tarefa = new Tarefa()
    {
        Id = tarefaDTO.Id,
        Titulo = tarefaDTO.Titulo,
        Descricao = tarefaDTO.Descricao,
        Concluida = tarefaDTO.Concluida
    };

    return View(tarefa);
}
```

Execute a aplicação e inclua algumas tarefas e ao clicar em 'Detalhes' as informações devem ser exibidas no formulário.

Após testar a aplicação, vamos fazer uma melhoria na usabilidade, que ao criar uma nova tarefa irá redirecionar para a listagem. Para isso, faça uma pequena mudança no resultado do método "Create" na "TarefaController", que agora irá redirecionar o usuário para a página "Index", ou seja após a criação de uma "Tarefa" o usuário será redirecionado para a página de listagem.

```
> return View();
```

Mudamos para:

```
> return RedirectToAction("Index");
```

Agora, ao clicar em Enviar, o usuário foi direcionado para a listagem que já contém o registro incluído.

EDIÇÃO E EXCLUSÃO

PASSO 01 - EDITANDO TAREFAS

Vamos criar os métodos para editar e excluir os registros de "Tarefa" do banco de dados, um método atualizará todas propriedades da "Tarefa" por "Id" e o método de exclusão utilizará somente o "Id" para deletar o registro do banco de dados.

No projeto "Tarefas.DAO", selecione o arquivo "TarefaDAO" e crie o método para atualizar a "Tarefa".

```
public void Atualizar(TarefaDTO tarefa)
{
    using (var con = Connection)
    {
        con.Open();
        con.Execute(
            @"UPDATE Tarefa
            SET Titulo = @Titulo, Descricao = @Descricao, Concluida = @Concluida
            WHERE Id = @Id;", tarefa
        );
    }
}
```

No projeto "Tarefas.Web", selecione o arquivo "TarefaController" e escreva o método "Update". Reparem que é um método praticamente igual ao método de 'Criar', porém, agora definindo também o Id da tarefa.

```
[HttpPost]
public IActionResult Update(Tarefa tarefa)
{
    var tarefaDTO = new TarefaDTO
    {
        Id = tarefa.Id,
        Titulo = tarefa.Titulo,
        Descricao = tarefa.Descricao,
        Concluida = tarefa.Concluida
    };

    var tarefaDAO = new TarefaDAO();
    tarefaDAO.Atualizar(tarefaDTO);

    return RedirectToAction("Index");
}
```

Crie o método "HttpGet" do "Update", assim como o método "Create" tem seu correspondente "HttpGet" que irá retornar o formulário preenchido com a "Tarefa".

```
public IActionResult Update(int id)
{
    var tarefaDAO = new TarefaDAO();
    var tarefaDTO = tarefaDAO.Consultar(id);

    var tarefa = new Tarefa()
    {
        Id = tarefaDTO.Id,
        Titulo = tarefaDTO.Titulo,
        Descricao = tarefaDTO.Descricao,
        Concluida = tarefaDTO.Concluida
    };

    return View(tarefa);
}
```

Note que não precisamos decorar o método com [HttpGet], porque esse é o padrão. Agora, precisamos ir no projeto "Tarefas.Web", criar uma "View" com o nome "Update.cshtml".

```
<form asp-action="Update">
    <input asp-for="Id" hidden />
    <input asp-for="Concluida" hidden />
    ... semelhante ao que já existia ...
</form>
```

No projeto "Tarefas.Web", vamos criar o link para "Editar" na listagem.

```
<td>
    <a asp-controller="Tarefa" asp-action="Details" asp-route-id="@item.Id">Detalhes</a>
    <a asp-controller="Tarefa" asp-action="Update" asp-route-id="@item.Id">Atulizar</a>
</td>
```

Execute a aplicação e clique em Atualizar.

PASSO 02 - EXCLUINDO TAREFAS

No projeto "Tarefas.DAO", selecione o arquivo "TarefaDAO" e crie o método para excluir a "Tarefa".

```
public void Excluir(int id)
{
    using (var con = Connection)
    {
        con.Open();
        con.Execute(
            @"DELETE FROM Tarefa
            WHERE Id = @Id", new { id }
        );
    }
}
```

No projeto "Tarefas.Web", selecione o arquivo "TarefaController" e escreva o método "Delete".

```
public IActionResult Delete(int id)
{
    var tarefaDAO = new TarefaDAO();
    tarefaDAO.Excluir(id);

    return RedirectToAction("Index");
}
```

No projeto "Tarefas.Web", crie o link para "Excluir" na listagem.

```
<td>
    <a asp-controller="Tarefa" asp-action="Details" asp-route-id="@item.Id">Detalhes</a>
    <a asp-controller="Tarefa" asp-action="Update" asp-route-id="@item.Id">Atulizar</a>
    <a asp-controller="Tarefa" asp-action="Delete" asp-route-id="@item.Id">Excluir</a>
</td>
```

Excute a aplicação, e clique no link 'Excluir'.

PASSO 03 - MELHORIA DE USABILIDADE

Para mais uma melhoria de usabilidade, podemos bloquear os campos do formulário na página de detalhes. Para isso, selecione o arquivo "Details" e inclua a "tag" "disabled" para que os campos não possam ser editados.

```
<div class="form-group">
    <label asp-for="Titulo">@Html.DisplayNameFor(model => model.Titulo)</label>
    <input asp-for="Titulo" class="form-control" disabled/>
</div>

<div class="form-group">
    <label asp-for="Descricao">@Html.DisplayNameFor(model => model.Descricao)</label>
    <input asp-for="Descricao" class="form-control" disabled/>
</div>
```

Inclusa também um link para voltar para a página de listagem:

```
<a asp-controller="Tarefa" asp-action="Index" asp-route-id="@Model.Id">Voltar</a>
```

E alguns campos ocultos para armazenar informações que o usuário não precisa visualizar ainda.

```
<input asp-for="Id" hidden />
<input asp-for="Concluida" hidden />
```

Execute a aplicação e clique no link "Detalhes"

REFATOÇÃO

PASSO 01 - REFATORANDO CLASSES

No projeto "Tarefas.Web", selecione o arquivo "TarefaController", vamos remover as declarações repetidas do "TarefaDAO".

Coloque a declaração da "TarefasDAO" comum a toda a classe e instancie no construtor.

```
private TarefaDAO tarefaDAO;

public TarefaController()
{
    tarefaDAO = new TarefaDAO();
}
```

Então vamos remover as declarações repetidas em cada 'Action'.

```
var tarefaDAO = new TarefaDAO();
```

No projeto "Tarefas.Web", renomeie a "Model" "Tarefa" para "TarefaViewModel". Este é apenas uma refatoração para manter o padrão.

Com esta mudança, precisaremos trocar os models registrados nas views:

- /Tarefa/Create.cshtml
- /Tarefa/Details.cshtml
- /Tarefa/Index.cshtml
- /Tarefa/Update.cshtml

Natualmente, as declarações na 'Controller' também precisarão ser ajustadas.

```
// alguns exemplos
var tarefa = new TarefaViewModel();
 ...
var listaDeTarefa = new List<TarefaViewModel>();
 ...
public IActionResult Create(TarefaViewModel tarefa)
```

Prontinho! Código mais uma vez refatorado!

VALIDAÇÕES

PASSO 01 - VALIDANDO DADOS DO FORMULÁRIO

Vamos aprender a validar os formulários utilizando "DataNotations", decorando as propriedades da "ViewModel", isso vai permitir uma validação sem condicionais (ifs) espalhados pelo método "Create" e "Update", além de definir uma mensagem que será utilizada pela "View" se a validação falhar.

No projeto "Tarefas.Web", selecione o arquivo "TarefaViewModel", inclua a referencia "DataAnnotations" na seção "using".

Decore as propriedades da classe com "Required", com exceção do "Id".

```
using System.ComponentModel.DataAnnotations;
```

Cada propriedade pode ter vários 'atributos'. Basta empilhá-los desta forma:

```
[Required]
[DisplayName("Descrição")]
public string? Descricao { get; set; }
```

No projeto "Tarefas.Web", selecione o arquivo "TarefaController" e no método "Create", inclua a validação do "ModelState", caso o "ModelState" não seja válido a mesma "View" será retornada.

```
if(!ModelState.IsValid)
{
    return View();
}
```

Copie e cole esse mesmo trecho de código no método "Update".

PASSO 02 - MOSTRANDO AS INFORMAÇÕES DE VALIDAÇÃO

No projeto "Tarefas.Web", selecione o arquivo "Create", inclua as "tags" que mostrarão a validação dos campos.

No início do formulário inclua "asp-validation-summary" que contém a lista de completa de erros. Para todos os outros campos do formulário inclua a "tag" "asp-validation-for" que exibirá o erro para a propriedade.

- https://learn.microsoft.com/pt-br/aspnet/core/tutorials/first-mvc-app/validation?view=aspnetcore-7.0 (https://learn.microsoft.com/pt-br/aspnet/core/tutorials/first-mvc-app/validation?view=aspnetcore-7.0) (documentação sobre validação no ASPNET MVC)
- Note que você também pode usar o razor da seguinte forma, ao invés das tags:

```
@Html.ValidationSummary()
```

Execute a aplicação, na página de criação do "Tarefa" clique em "Enviar" sem ter preenchido nenhum campo.

- Note que as mensagens estão genéricas e em inglês vamos melhorar isso.

PASSO 03 - MELHORANDO A VALIDAÇÃO

Em "TarefaViewModel" inclua as mensagens nas "tags" "Required".

```
[Required(ErrorMessage = "A descrição da tarefa deve ser preenchida.")]
```

Execute a aplicação, clique em Enviar, veja que as mensagens já estão funcionado.

Veja que o arquivo "TarefaDAO" no comando "CreateDatabase" especificamos que os campos só podem ter 100 caracteres, vamos colocar essa critica nas validações dos campos.

Em "TarefaViewModel" inclua as mensagens nas "tags" "MinLength" para definir o tamanho mínimo e "MaxLength" para o tamanho máximo.

```
[Required(ErrorMessage = "A descrição da tarefa deve ser preenchida.")]
[MinLength(5, ErrorMessage = "A descrição deve ter no mínimo 5 caracteres.")]
```

Execute a aplicação digite apenas a letra 'a' em todas os campos e clique em "Enviar".

- Note que mesmo se depois que foram incluídos outros caracteres em "Nome" a mensagem de erro continua sendo exibida, isso acontece porque a validação é feita no back-end, ou seja, só quando o formulário é submetido a "Controller" que é validado. Para habilitar a validação assíncrona, ou seja, sem submeter o formulário para a "Controller", temos que incluir o código que habilita essa validação.

No projeto "Tarefas.Web", selecione o arquivo "Create" e o "Update", inclua a seção "Scripts".

```
@section Scripts
{
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }
}
```

Execute a aplicação digite apenas a letra 'a' em todos os campos, clique em "Enviar" e depois digite os outros caracteres em Nome e veja a mensagem de erro sendo atualizada.

- Note que a mensagem de erro abaixo do controle "Nome" sumiu e dos outros controles vão atualizando assim que o texto é digitado.

NOVOS CONCEITOS

PASSO 01 - INJEÇÃO DE DEPENDÊNCIA

Agora vamos aprender a configurar a injeção de dependência no Asp.Net MVC Core. No projeto "Tarefas.Web", selecione o arquivo "TarefaController". Estamos instanciando a classe no construtor, como iremos utilizar a injeção de dependência não instanciaremos mais essa classe, receberemos essa instancia pelo construtor da "Controller".

```
private TarefaDAO tarefaDAO;

public TarefaController()
{
    tarefaDAO = new TarefaDAO(); // aqui está a inicialização diretamente no construtor
}
```

Selecione o arquivo "Program.cs", na seção "using" inclua a referencia para "Tarefas.DAO" e na linha abaixo do método "AddControllersWithViews" inclua a "AddTransient" com a classe DAO de tarefas.

```
// Classes DAO
builder.Services.AddTransient<TarefaDAO>();
```

Em seguida, remova a instanciação da classe do construtor e adicione uma 'injeção' da DAO utilizando um parâmetro diretamente no construtor. Por padrão, vamos armazenar este parâmetro numa propriedade 'readonly' com um prefixo '_' desta forma:

```
private readonly TarefaDAO _tarefaDAO;

public TarefaController(TarefaDAO tarefaDAO)
{
    _tarefaDAO = tarefaDAO;
}
```

Execute a aplicação e veja que o sistema continua funcionado como antes.

Esse exemplo foi a maneira mais simples de ver a injeção de dependência funcionando, porém não é a mais útil, geralmente utilizamos interfaces para reduzir a dependência entre as camadas, no nosso caso a camada de interface com usuário (MVC) e a camada de acesso a dados (DAO). Vamos melhorar nossa implementação

Crie uma interface chamada "ITarefaDAO", selecione o nome da classe "TarefaDAO" e clique no ícone de lâmpada, selecione a opção "Extrair a interface...".

```
public interface ITarefaDAO
{   void Atualizar(TarefaDTO tarefa);
    List<TarefaDTO> Consultar();
    TarefaDTO Consultar(int id);
    void Criar(TarefaDTO tarefa);
    void Excluir(int id);
}
```

Mude a 'Controller' para utilizar apenas a interface.

```
private readonly ITarefaDAO _tarefaDAO;

public TarefaController(ITarefaDAO tarefaDAO)
{
    _tarefaDAO = tarefaDAO;
}
```

Selecione o arquivo "Program.cs", e altere a implamentação da interface no "AddTransient" para também utilizar a interface.

```
builder.Services.AddTransient<ITarefaDAO, TarefaDAO>();
```

- Note que definimos um "De => Para", quando um construtor de um “Controller” for instanciado um tipo "ITarefaDAO" será preenchido com uma instancia de "TarefaDAO".

AUTOMAPPER

PASSO 01 - CONFIGURANDO O AUTOMAPPER

Agora vamos aprender como instalar, configurar e utilizar o "AutoMapper", esse framework nós ajudará a converter o "ViewModel" em "DTO" e vice-versa. Removendo assim código repetido e agilizando o desenvolvimento.

No projeto "Tarefas.Web", adicione a referência para o package "AutoMapper".

```
> dotnet add package AutoMapper --version 10.1.1
```

- Note que será criada a linha "PackageReference" no arquivo "Tarefas.Web.csproj"

No arquivo "Program.cs" inclua as referências "AutoMapper", "Tarefas.DTO" e "Tarefas.Web.Models".

```
using Tarefas.DAO;
using Tarefas.DTO;
using Tarefas.Web.Models;
using AutoMapper;
```

Ainda no arquivo "Program.cs", antes das demais injeções de dependências, inclua a configuração do "AutoMapper":

```
var config = new AutoMapper.MapperConfiguration(c => {
    c.CreateMap<TarefaViewModel, TarefaDTO>().ReverseMap();
});
```

Em seguida crie o "Mapper":

```
IMapper mapper = config.CreateMapper();
```

E inclua ele na injeção de dependência.

```
builder.Services.AddSingleton(mapper);
```

- Note na função "CreateMap" que faremos um "De => Para" de "TarefaViewModel x TarefaDTO" e para que possamos fazer também o contrario "TarefaDTO X TarefaViewModel", utilizando a função "ReverseMap".

PASSO 02 - UTILIZANDO O AUTOMAPPER

. No projeto "Tarefas.Web", selecione o arquivo "TarefaController.cs", inclua a referência para "AutoMapper" na seção "using":

```
using AutoMapper;
```

E a variável de instancia que será recebida pelo construtor.

```
private readonly IMapper _mapper;

public TarefaController(ITarefaDAO tarefaDAO, IMapper mapper)
{
    _tarefaDAO = tarefaDAO;
    _mapper = mapper;
}
```

No método "HttpPost" do "Create", vamos substituir o preenchimento propriedade a propriedade pelo comando do "AutoMapper". Antes, passando propriedade a propriedade.

```
var tarefaDTO = _mapper.Map<TarefaDTO>(tarefaViewModel);
```

No caso dos métodos que buscam listas:

```
foreach (var tarefaDTO in listaDeTarefasDTO)
{
    listaDeTarefa.Add(_mapper.Map<TarefaViewModel>(tarefaDTO));
}
```

- Note que no método "Details" estamos convertendo de "DTO" para "ViewModel", diferente do método "Create" que estamos convertendo de "ViewModel" para "DTO".

Faça o mesmo com os demais métodos.

HORA DE DEIXAR BONITÃO

PASSO 01 - CLASSES DO BOOTSTRAP

Agora que inserimos várias camadas de desenvolvimento em nossa aplicação e ela já está 'funcionando', vamos revisar um pouco a parte de interface para tornar o sistema mais agradável antes de evoluí-lo. Para isso, vamos utilizar a documentação do Bootstrap. https://getbootstrap.com/docs/5.0/getting-started/introduction/ (https://getbootstrap.com/docs/5.0/getting-started/introduction/)

Primeiramente, vamos ajustar os espaçamentos dos formulários: Usaremos as classes css 'row g-3', desta forma:

```
<form class="row g-3">

</form>
```

Em seguida vamos ajustar o espaçamento entre os componentes do formulário: Para isso, vamos criar uma div com a classe 'col-12' 'abrançando' cada 'form-group', assim:

```
<div class="col-12">
    <div class="form-group">

    </div>
</div>
```

E para os botões de ação: 'Enviar' e 'Voltar', vamos deixá-los padronizados usando a classe 'btn' e seus derivados. Permitindo que o link e o input fiquem visualmente semelhantes.

```
<div class="col-12">
    <input type="submit" class="btn btn-primary" /> <a class="btn btn-secondary" asp-controller="Tarefa" asp-action="Index">Voltar</a
</div>
```

Para finalizar esta parte, vamos colocar um título em cada página dentro do 'form':

```
<form>
    <h3>Título da página</h3>
</form>
```

Faça estes ajustes nas páginas:

- Create.cshtml
- Update.cshtml
- Details.cshtml

PASSO 02 - CHECKBOX DE STATUS - CONCLUÍDO OU NÃO

Até este momento, as tarefas possuem apenas a possibilidade de cadastrar título e descrição. Precisamos agora, permitir que o usuário diga se a tarefa está concluída ou não. Para isso, vamos adicionar um checkbox ao formulário.

```
<div class="col-12">
    <div class="form-group">
        <input type="checkbox" asp-for="Concluida" />
        <label asp-for="Concluida">@Html.DisplayNameFor(model => model.Concluida)</label>
    </div>
</div>
```

Fazendo esta adição ao formulário o checkbox já deve estar funcionando e sendo responsável por informar se a tarefa está concluída ou não. Adicione este checkbox às páginas:

- Create.cshtml
- Update.cshtml
- Details.cshtml

Teste a aplicação e altere as informações de status de uma tarefa.

```
> dotnet watch
```

Utilizando, o dotnet watch, você poderá fazer alguns ajustes visuais e validar as alterações em tempo real. Portanto, agora que o checkbox está funcionando, vamos para deixar tudo mais 'agradável' utilizando os padrões do bootstrap. Para isso, vamos transformar o 'checkbox' em um 'switch', apenas adicionando algumas classes aos elementos, desta forma:

```
<div class="col-12">
    <div class="form-check form-switch">
        <input type="checkbox" asp-for="Concluida" class="form-check-input"/>
        <label class="form-check-label" asp-for="Concluida">@Html.DisplayNameFor(model => model.Concluida)</label>
    </div>
</div>
```

PASSO 03 - STATUS - CONCLUÍDO OU NÃO NA LISTAGEM

Agora, vamos transformar o texto 'true' e 'false' que aparecem automaticamente na listagem quando imprimimos o valor da propriedade booleana em uma informação mais amigável. Para isso vamos criar uma 'PartialView' e aproveitar para demonstrar como podemos componentizar pequenas partes do sistema em unidades mais simples e favorecer a manutenção. Neste caso, vamos criar uma 'PartialView' que ficará responsável exclusivamente por exibir a informação de SIM ou NÃO de uma tarefa com um ícone associado.

Por padrão as 'PartialView' tem o nome do arquivo iniciado com um underline '_'.

Crie um arquivo chamado '_Status.cshtml' na pasta das views de 'Tarefa':

- _Status.cshtml

Dentro deste arquivo vamos definir o model que ele irá interpretar para ser renderizado.

```
@model TarefaViewModel
```

Para selecionar o ícone, você pode visitar o site do bootstrap, na sessão de 'Icons': https://icons.getbootstrap.com/ (https://icons.getbootstrap.com/) Escolha um para ser o ícone associado ao SIM e outro para ser associado ao NÃO.

Neste caso estamos usando o 'bi-check-circle-fill' e o 'bi-circle-fill'.

```
@if(Model.Concluida)
{
    <span><i class="bi bi-check-circle-fill"></i> Sim</span>
}
else
{
    <span><i class="bi bi-circle-fill"></i> Não</span>
}
```

Com isso, podemos alterar a página de listagem e onde a página havia apenas a propriedade '@item.Concluida' pela referência da PartialView:

```
<td>
    <partial name="_Status" model="item" />
</td>
```

Para finzaliar vamos padronizar os botões de ação da lista:

```
<td style="width: 10px; white-space: nowrap;">
    <a class="btn btn-secondary btn-sm" asp-controller="Tarefa" asp-action="Details" asp-route-id="@item.Id">Detalhes</a>
    <a class="btn btn-secondary btn-sm" asp-controller="Tarefa" asp-action="Update" asp-route-id="@item.Id">Atualizar</a>
    <a class="btn btn-secondary btn-sm" asp-controller="Tarefa" asp-action="Delete" asp-route-id="@item.Id">Excluir</a>
</td>
```

Vamos navegar e validar as mudanças

```
> dotnet watch
```

LOGIN DE ACESSO

PASSO 01 - ARQUITETURA DA SOLUÇÃO

Até este momento temos uma aplicação que é capaz de gerenciar as tarefas de maneira bem isolada. Mas pensando em uma aplicação funcional, é comum haver uma estrutura de autenticação para permitir que as tarefas estejam associadas a cada usuário logado no sistema.

Para iniciar esta missão, vamos reproduzir boa parte do que aprendemos até este momento para uma nova estrutura de dados, relacionada aos usuários. Vamos utilizar estes dados para permitir que ao haver o resultado positivo da checagem de um login e senha, as informações do usuário selecionado sejam armazenadas num 'cookie' e essas informações filtrem as tarefas do banco de dados da aplicação.

Neste processo, vamos precisar refatorar algumas estratégias que funcionaram bem isoladas, mas que ao adicionarmos uma nova funcionalidade no sistema passam a não ser adequadas.

Preparados?! Prontos ou não... Vamos lá!

PASSO 02 - MODELO DE DADOS DO USUÁRIO

Primeiramente, vamos criar a estrutura de dados dos usuários, para isso, no projeto de 'Tarefas.DTO', criaremos a classe 'UsuárioDTO'.

```
public class UsuarioDTO
{
    public string Email { get; set; }
    public string Senha { get; set; }
    public string Nome { get; set; }
    public bool Ativo { get; set; }
}
```

PASSO 03 - ACESSO A DADOS DE USUÁRIO

Em seguida, vamos criar a camada de acesso a dados (DAO). Vamos agora para o projeto 'Tarefas.DAO'.

Crie uma classe chamada 'UsuarioDAO' e adicione os métodos de operação seguindo o mesmo padrão que fizemos na classe de 'TarefasDAO', lembrando que alterar o nome da tabela e as propriedades conforme o DTO que criamos no passo anterior. Usaremos por padrão a tabela chamada 'Usuario'.

Por exemplo, o método 'Criar' ficaria assim:

```
public void Criar(UsuarioDTO usuario)
{
    using (var con = Connection)
    {
        con.Open();
        con.Execute(
            @"INSERT INTO Usuario
            (Email, Senha, Nome, Ativo) VALUES
            (@Email, @Senha, @Nome, @Ativo);", usuario
        );
    }
}
```

Garanta que os métodos estejam implementados, conforme já fizemos no DAO de Tarefas:

```
- public void Criar(UsuarioDTO usuario)
- public List<UsuarioDTO> Consultar()
- public UsuarioDTO Consultar(int id)
- public void Atualizar(UsuarioDTO usuario)
- public void Excluir(int id)
```

PASSO 04 - CRIANDO O MÉTODO DE AUTENTICAÇÃO

Além dos métodos padrão de operação do CRUD (create, read, update e delete) que já criamos no passo anterior, o DAO de usuários precisará ter um método extra para a operação 'Autenticar'. Podemos criá-lo com a seguinte assinatura:

```
public UsuarioDTO Autenticar(string email, string senha)
```

A estrutura do método deve ser capaz de encontrar um usuário a partir de um email e uma senha, validando sempre que o status 'Ativo' esteja definido como 'true'. Atendendo a estes requisitos, podemos escrever a seguinte query.

```
public UsuarioDTO Autenticar(string email, string senha)
{
    using (var con = Connection)
    {
        con.Open();
        UsuarioDTO result = con.Query<UsuarioDTO>
        (
            @"SELECT Id, Email, Senha, Nome, Ativo FROM Usuario
            WHERE Email = @Email AND Senha = @Senha AND Ativo = true", new { email, senha }
        ).First();
        return result;
    }
}
```

Uma vez criada a classe DAO, podemos criar a interface que descreve as assinaturas que criamos. Também semelhante ao que já criamos para o modelo de dados de 'Tarefa'. Chamaremos a interface de 'IUsuarioDAO'.

```
public interface IUsuarioDAO
{   void Atualizar(UsuarioDTO conta);
    List<UsuarioDTO> Consultar();
    UsuarioDTO Autenticar(string email, string senha);
    UsuarioDTO Consultar(int id);
    void Criar(UsuarioDTO conta);
    void Excluir(int id);
}
```

Verifique se o projeto está compilando corretamente até este ponto, pode ser necessário ajustar algumas referências: 'using' em cada classe que criamos.

Execute o comando de build e corriga os problemas que podem surgir no console, até receber a mensagem de um build bem sucedido.

```
> dotnet build
```

Compile o projeto com sucesso e finalizamos esta etapa.

LOGIN DE ACESSO

PASSO 01 - REFACTORAÇÃO: POLIMORFISMO

As duas classes DAO que criamos até o momento já possuem algum códigos repetidos. Numa aplicação real, a possíbilidade desta duplicação causar problemas é enorme, justamente porque a casa nova DAO criada, estaríamos aumentando a quantidade de duplicações e qualquar alteração exigiria voltar em todas as classes ajustando.

Para fins didáticos, vamos criar uma classe base e deixar nesta classe as propriedades comuns às classes DAO. Para evitar que esta classe base seja instanciada podemos usar o modificador 'abstract', que garante que esta classe possa apenas ser herdada por outras, mas nunca utilizada diretamente.

```
public abstract class BaseDAO
```

Nesta classe vamos definir as propriedades 'Connection' e 'DataSourceFile' que até este momento estão definidos nas classes de DAO.

```
public abstract class BaseDAO
{
    public string DataSourceFile => Environment.CurrentDirectory + "AppTarefasDB.sqlite";
    public SQLiteConnection Connection => new SQLiteConnection("DataSource=" + DataSourceFile);

    public BaseDAO()
    {
    }
}
```

Em seguida, podemos aplicar o polimorfismo e através da herança utilizar a 'BaseDAO' nas classes 'TarefaDAO' e 'UsuarioDAO'. Assim, removendo as propriedades de lá e permitindo que qualquer alteração na configuração de banco de dados seja feita num ponto centralizado.

No CSharp (.NET), podemos herdar apenas uma classe, mas podemos implementar quantas interfaces quisermos numa classe. Por isso, é necessário colocar depois dos 'dois pontos' o nome da classe que será herdada e em seguida separar por vírgulas todas as interfaces que quisermos implementar na classe.

Ficando desta forma em 'TarefaDAO'.

```
public class TarefaDAO : BaseDAO, ITarefaDAO
```

O mesmo para a classe de 'UsuarioDAO'.

```
public class UsuarioDAO : BaseDAO, IUsuarioDAO
```

Fazendo isso, podemos remover as propriedades 'Connection' e 'DataSourceFile' que estavam definidas nas classes 'TarefaDAO' e 'UsuarioDAO'.

Verifique se o projeto está compilando corretamente até este ponto:

```
> dotnet build
```

PASSO 02 - REFACTORAÇÃO: CRIAÇÃO DO BANCO DE DADOS

Agora que temos duas classes DAO, a lógica simples de verificar a existência do arquivo para executar ou não a criação da tabela no SQLite não é suficiente para manter o sistema coerente, pois após a primeira DAO ser acessada o arquivo é criado e a lógica não funcionará na classe seguinte, criando um sério BUG na aplicação.

Para resolver este problema, vamos extrair a lógica da criação da estrutura do banco de dados das classes de manipulação de dados DAO.

Neste caso, criaremos uma classe chamada 'DatabaseBootstrap' e moveremos para ela a responsabilidade de criar as tabelas e até podemos adicionar uma lógica de popular algumas informações preliminares no banco. Permitindo por exemplo, que um usuário padrão exista no sistema para que possamos efetuar o login na aplicação logo que a aplicação for executada pela primeira vez.

Ela também irá herdar de BaseDAO e terá sua propria interface.

```
public class DatabaseBootstrap : BaseDAO, IDatabaseBootstrap
```

Nesta classe vamos criar um método chamado Setup.

```
public void Setup()
```

E incluir nele toda lógica de criação de tabelas que estavam nas 'UsuarioDAO' e 'TarefaDAO'.

```
public void Setup()
{
    using (var con = Connection)
    {
        if(!File.Exists(DataSourceFile))
        {
            con.Execute(
                @"CREATE TABLE Tarefa
                (
                    Id          integer primary key autoincrement,
                    Titulo      varchar(100) not null,
                    Descricao   varchar(100) not null,
                    Concluida   bool not null
                )"
            );

            con.Execute(
                @"CREATE TABLE Usuario
                (
                    Id          integer primary key autoincrement,
                    Email       varchar(100) not null,
                    Senha       varchar(100) not null,
                    Nome        varchar(100) not null,
                    Ativo       bool not null
                )"
            );
        }
    }
}
```

PASSO 03 - REFACTORAÇÃO: ADICIONANDO DADOS PRELIMINARES

Como o banco é criado assim que a aplicação é iniciada, podemos aproveitar para inserir alguns dados básicos para utilização da mesma. Por enquanto, vamos adicionar apenas um usuário (pois não temos nenhuma página de gestão de usuários ainda).

Faremos isso adicionando um método chamado InsertDefaultData.

```
private void InsertDefaultData(SQLiteConnection con)
```

Este método vai receber a conexão por parâmetro para aproveitar a mesma conexão aberta na criação das tabela e inserir os dados. Vamos chamar este método dentro do 'using', mas no final do método de 'Setup'.

```
private void InsertDefaultData(SQLiteConnection con)
{
    var conta = new ContaDTO()
    {
        Email = "andre@gmail.com",
        Senha = "biscoito",
        Nome = "André Paulovich",
        Ativo = true
    };

    con.Execute(
        @"INSERT INTO Conta
        (Email, Senha, Nome, Ativa) VALUES
        (@Email, @Senha, @Nome, @Ativo);", conta
    );
}
```

Criamos então a nossa interface com o único método público da classe:

```
using System;
using System.Collections.Generic;
using Tarefas.DAO;

namespace Tarefas.DAO
{
    public interface IDatabaseBootstrap
    {
        void Setup();
    }
}
```

Podemos então remover a responsabilidade de criação de tabelas das classes 'UsuarioDAO' e 'TarefaDAO'.

Bastando remover os métodos CreateDatabase() e os construtores que testavam a existência do arquivo.

Agora vamos compilar mais uma vez e verificar a nossa aplicação.

```
> dotnet build
```

Compile o projeto com sucesso e finalizamos esta etapa.

NOVAS CONFIGURAÇÕES DE INJEÇÃO DE DEPÊNDENCIA

PASSO 01 - INJETANDO DAO DE USUÁRIO E SETUP DO BANCO

Na classe de incialização da aplicação 'Program.cs', no ponto onde já incluímos a injeção de dependência do TarefaDAO, vamos incluir duas novas configurações, da seguinte forma:

```
// Classes DAO
builder.Services.AddSingleton<IDatabaseBootstrap, DatabaseBootstrap>();
builder.Services.AddTransient<ITarefaDAO, TarefaDAO>();
builder.Services.AddTransient<IContaDAO, ContaDAO>();
```

Neste caso, estamos usando o AddSingleton para o DatabaseBootstrap, para garantir que exista apenas uma instância da mesma durante a execução da aplicação. Para conhecer melhor os modificadores de escopo de 'lifetime' dos registros de injeção, vale a pena a leitura do seguinte tópico da documentação. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-7.0 (https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-7.0)

Agora, precisamos garantir que o método Setup da classe DatabaseBootstrap seja executado imediatamente antes da inicialização da aplicação e apenas uma vez. Para isso, vamos adicionar uma chamada do serviço antes do 'app.Run();' ainda na classe de 'Program.cs'. Desta forma:

```
app.Services.GetService<IDatabaseBootstrap>()!.Setup();

app.Run();
```

Compile a aplicação e execute-a em seguida usando:

```
> dotnet run
```

E verifique se o arquivo do banco de dados foi criado dentro da pasta 'src' e se a estrutura do banco inclui as duas tabelas 'Usuario' e 'Tarefas' e se o usuário padrão esta devidamente inserido na tabela de 'Usuario'.

PASSO 02 - CRIANDO AS VIEWS NECESSÁRIAS PARA O LOGIN

Primeiramente, vamos criar a ViewModel de Login. Crie uma classe chamada 'LoginViewModel' dentro da pasta 'Models' no projeto 'Tarefas.Web'. Nesta classe usaremos tudo que já aprendemos sobre os decoradores de Required, DisplayName e etc.

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel;
using System.Diagnostics.CodeAnalysis;

namespace Tarefas.Web.Models
{
    public class LoginViewModel
    {
        [AllowNull]
        [Required(ErrorMessage = "O Email deve ser informado.")]
        [EmailAddress(ErrorMessage = "Digite um email em formato válido.")]
        [DisplayName("Email")]
        public string Email { get; set; }

        [AllowNull]
        [Required(ErrorMessage = "A senha deve ser informada.")]
        [DataType(DataType.Password)]
        [DisplayName("Senha")]
        public string Senha { get; set; }
    }
}
```

Em seguida, crie uma pasta 'Login' dentro de 'Views' e então adicione um arquivo chamado 'Index.cshtml' nesta pasta.

Lembre-se de definir o model que será utilizado na View com o atributo @model.

```
@model Tarefas.Web.Models.LoginViewModel
```

Construa um formulário simples com inputs de 'email' e 'senha' e um botão de submit.

```
<div class="row justify-content-md-center">
    <div class="col-md-4 g-3">
        <form method="post" class="g-3">
            <h1 class="h3 mb-3 font-weight-normal">Minhas Tarefas</h1>
            <p>Informe seu login e senha para entrar.</p>
            <hr>
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group mb-3">
                <label asp-for="Email">@Html.DisplayNameFor(model => model.Email)</label>
                <input asp-for="Email" class="form-control">
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>

            <div class="form-group mb-3">
                <label asp-for="Senha">@Html.DisplayNameFor(model => model.Senha)</label>
                <input asp-for="Senha" class="form-control">
                <span asp-validation-for="Senha" class="text-danger"></span>
            </div>

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

                <button type="submit" class="btn btn-primary btn-block mb-4">Entrar</button>

            </div>
        </form>
    </div>
</div>
```

Lembre-se de no final do arquivo definir a área de Scripts que importa os arquivos necessários para validação com javascript do aspnet.

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")
}
```

No vaso da página de Login, um detalhe de usabilidade é bastante importante. Pois o '_Layout.cshtml' que é o arquivo que define a estrutura padrão das páginas do sistema, possui um cabeçalho com menus e algumas informações que só fazem sentido para quem já estiver logado no sistema.

Por isso, vamos copiá-lo e renomear a cópia para '_Login.cshtml'.

```
@{
    Layout = "_Login";
}
```

Neste arquivo '_Login.cshtml' faremos algumas modificações para deixar o login centralizado e remover os componentes desnecessários como: rodapé e menus.

Para isso, remova incialmente os elementos de cabeçalho:

```
<header></header>
```

E também o rodapé:

```
<footer></footer>
```

Removendo junto, todos os elementos dentro destas tags.

Em seguida, vamos adicionar um arquivo de configuração visual (CSS) específico para a tela de login.

```
<link rel="stylesheet" href="~/css/login.css" asp-append-version="true" />
```

Este arquivo deve ser criado dentro de "wwwroot/css/" no projeto 'Tarefas.Web':

```
html,
body {
  height: 100%;
}

body {
  display: -ms-flexbox;
  display: -webkit-box;
  display: flex;
  -ms-flex-align: center;
  -ms-flex-pack: center;
  -webkit-box-align: center;
  align-items: center;
  -webkit-box-pack: center;
  justify-content: center;
  padding-top: 40px;
  padding-bottom: 40px;
  background-color: #f5f5f5;
}

.form-signin {
  width: 100%;
  max-width: 330px;
  padding: 15px;
  margin: 0 auto;
}
```

Com isso, a tela de login deve ficar limpa e com o formulário centralizado na página.

Porém, para conseguirmos visualizar a página de login, precisaremos criar um 'Controller'. Para isso, crie um arquivo chamado 'LoginController.cs' dentro da pasta 'Controllers' no projeto 'Tarefas.Web'.

E defina o método 'Index' retornando a 'View' para que as convenções de nomes façam a página ser carregada corretamente:

```
namespace Tarefas.Web.Controllers
{
    public class LoginController : Controller
    {
        public LoginController()
        {
        }

        public IActionResult Index()
        {
            return View();
        }
    }
}
```

Execute a aplicação e verifique nossa evolução:

```
> dotnet run
```

CONCEITOS DE SEGURANÇA

PASSO 01 - AUTENTICAÇÃO

Vamos concluir a configuração de automapper do novo modelo que criamos para os usuários:

```
var config = new AutoMapper.MapperConfiguration(c => {
    c.CreateMap<TarefaViewModel, TarefaDTO>().ReverseMap();
    c.CreateMap<UsuarioViewModel, UsuarioDTO>().ReverseMap();
});
```

Em seguida vamos adicionar o pacote de gerenciamento de autenticação, ao projeto 'Tarefas.Web' através do comando:

```
dotnet add package Microsoft.AspNetCore.Authentication.Abstractions;
```

Uma vez instalado, adicione o using à classe 'Program.cs':

```
using Microsoft.AspNetCore.Authentication.Cookies;
```

E então logo abaixo da configuração de injeção de dependência, vamos adicionar uma configuração que dirá ao aspnet que usaremos uma autenticação via 'cookie' e que a página de autenticação funcionará no path: '/Login'.

```
// Autenticação
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(x => x.LoginPath = "/Login");
```

Isso fará com que o aspnet ao detectar um usuário 'anônimo', o envie diretamente para a página de autenticação.

Para finalizar, precisamos adicionar ao pipeline de processamento do aspnet as instruções de autenticação. Fazemos isso, adicionando a chamada 'UseAuthentication'. Podemos adicionar essa linha, bem abaixo do UseAuthorization (que já está definido por padrão).

```
app.UseAuthentication();
```

Agora vamos compilar mais uma vez e verificar a nossa aplicação.

```
> dotnet run
```

Acesse a página http://localhost:{porta}/Login (http://localhost:%7Bporta%7D/Login) e veja se o formulário de login aparece corretamente.

MÉTODO DE AUTENTICAÇÃO

PASSO 01 - TELA DE LOGIN

Vamos adicionar o comportamento de validação do login na 'LoginController'. Para isso, precisamos usar a estratégia da injeção de dependência no construtor da controller.

```
private readonly IUsuarioDAO _usuarioDAO;
private readonly IMapper _mapper;

public LoginController(IUsuarioDAO usuarioDAO, IMapper mapper)
{
    _usuarioDAO = usuarioDAO;
    _mapper = mapper;
}
```

Como já aprendemos, precisamos criar um método POST para receber as informações do formulário de login. Neste método vamos definir o parâmetro de entrada como o 'LoginViewModel'.

```
[HttpPost]
public IActionResult Index(UsuarioViewModel usuarioViewModel)
{
    // logica de autenticação
}
```

Dentro desta rota do controller, vamos aplicar a lógica de autenticação acionando o método 'Autenticar' da 'UsuarioDAO', enviando os valores de email e senha que vieram no objeto 'LoginViewModel'.

Lembre-se que o método 'Autenticar' retorna um 'UsuarioDTO'.

```
UsuarioDTO user = _usuarioDAO.Autenticar(usuarioViewModel.Email, usuarioViewModel.Senha);
```

Caso este objeto 'user' esteja devidamente preenchido com os valores do nosso usuário, vamos criar um cookie com estas informações do usuário usando uma lista de 'Claim'. As Claim's descrevem algumas informações padronizadas do sistema de auteticação. Se desejar entender um pouco mais a fundo sobre o tema, acesse a documentação: https://learn.microsoft.com/pt-br/dotnet/api/system.security.claims.claimtypes?view=net-7.0 (https://learn.microsoft.com/pt-br/dotnet/api/system.security.claims.claimtypes?view=net-7.0)

Em nosso caso, ficará da seguinte maneira:

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Nome),
    new Claim(ClaimTypes.Email, user.Email)
};
```

Agora seguindo a estrutura básica de autenticação do AspNet precisamos criar uma 'identidade' que será trafegada durante as requisições. Fazemos isso instanciando um 'ClaimsIdentity' da seguinte forma:

```
var claimsIdentity = new ClaimsIdentity(claims, CookieAuthenticationDefaults.AuthenticationScheme);
```

Existem algumas configurações possíveis de serem definidas que podem alterar significativamente o comportamento da autenticação. Em nosso caso vamos definir apenas a expiração do token, a persistência do cookie e a url de login, para onde o usuário será redirecionado ao ser devidamente autenticado no sistema.

```
var authProperties = new AuthenticationProperties
{
    ExpiresUtc = DateTimeOffset.UtcNow.AddMinutes(10),
    IsPersistent = true,
    RedirectUri = "/Login"
};
```

Agora que configuramos todos os objetos da autenticação, precisamos adicionar o cookie ao contexto HTTP e redirecionar para a tela inicial do sistema.

```
HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    authProperties);

return LocalRedirect("/Home");
```

Agora vamos compilar mais uma vez e verificar a nossa aplicação.

```
> dotnet run
```

Acesse a página http://localhost:{porta}/Login (http://localhost:%7Bporta%7D/Login) e veja se ao informar os dados do nosso usuário de teste é possível realizar o login.

MELHORIAS DE INTERFACE

PASSO 01 - TELA DE LOGIN - VALIDAÇÃO

Se você digitar o login e senha errados, verá que o sistema apresentará uma exceção e erro na tela. Claramente este não é um comportamento que gostaríamos que acontecesse. Portanto, para ajustar, vamos utilizar o bloco "try catch".

De maneira bem simplificada, usamos este bloco para 'envolver' trechos de código que podem apresentar algum erro e então 'capturá-los' antes que sejam propagados até a tela do usuário. Para nossa prática, vamos aplicar este bloco para capturar qualquer exceção que aconteça no processo de autenticação, desta forma:

```
ContaDTO user;

[HttpPost]
public IActionResult Index(UsuarioViewModel usuarioViewModel)
{
    UsuarioDTO user;

    try
    {
        user = _usuarioDAO.Autenticar(usuarioViewModel.Email, usuarioViewModel.Senha);
    }
    catch (Exception ex)
    {
        // logica de tratamento da exceção que vamos adicionar
    }

    // demais instruções já implementadas do processo de login
}
```

O Aspnet MVC possui alguns facilitadores para este processo de validação e um deles é que podemos usar o 'ValidationSummary' que já existe na tela para apresentar os erros de validação de javascript dos campos de email e senha (formato de campo ou de obrigatoriedade) para adicionar uma mensagem do servidor para o "erro" por exemplo de um email e senha que não batem com nenhum usuário no banco de dados.

```
// exemplo
AddModelError("nome-do-campo", "mensagem que você gostaria de apresentar na tela");
```

No nosso caso, não vamos um "campo" associado, então podemos deixar uma string vazia. Quando estivermos com um erro desse tipo, não há possibilidade de seguir o processo de autenticação, então encerramos o método fazerndo o retorno da 'View'.

```
ModelState.AddModelError(string.Empty, ex.Message);
return View();
```

Este mesmo objeto 'ModelState', possui uma propriedade 'IsValid' que define se as validações definidas estão sendo cumpridas e portanto o formulário é "válido". Vamos usar este propriedade para validar e só executar a estrutura de autenticação do login caso o formulário seja válido.

```
if (ModelState.IsValid)
{
    // implementação da autenticação
}
```

PASSO 02 - BLOQUEANDO O ACESSO ÀS PÁGINAS

Até este momento, apesar de estarmos criando o cookie com as informações de um usuário, não há nada que impeça nenhum usuário de acessar o sistema. Ainda que exista uma página de login, qualquer pessoa que digite o caminho direto para a página inicial do sistema http://localhost:5240/Home (http://localhost:5240/Home) ou para a página de tarefas http://localhost:5240/Tarefa (http://localhost:5240/Tarefa), irá conseguir acessar normalmente.

Para de fato proteger as telas do sistema que serão acessíveis apenas para usuário 'logados' precisamos decorar com um atributo [Authorize]. Podemos fazer isso em cada método do Controller ou diretamente no Controller (para proteger todos os métodos).

Antes, precisamos adicionar uma nova extensão ao projeto, usando o comando:

```
dotnet add package Microsoft.AspNetCore.Authorization
```

Uma vez instalado o pacote de autorização, podemos adicionar o pacote nos controllers HomeController e TarefaController (que iremos proteger).

```
using Microsoft.AspNetCore.Authorization;
```

```
[Authorize]
public class HomeController : Controller
{
    public HomeController()
    {

    }
}
```

```
[Authorize]
public class TarefaController : Controller
{
    public TarefaController()
    {

    }
}
```

Agora podemos executar o projeto e tentar novamente acessar diretamente as páginas sem estar autenticado. Se tudo estiver correto, você será redirecionado para a página de login e nenhuma tentativa de burlar a autenticação será bem sucedida.

```
> dotnet run
```

PASSO 03 - REALIZANDO O LOGOUT (SAIR DO SISTEMA)

Ao executar o sistema mais uma vez, você irá perceber que já está 'autenticado' e não precisa mais efetuar o login para entrar no sistema, mesmo parando a execução da aplicação e iniciando-o novamente. Isso acontece porque definimos o cookie persistente e com expiração de 30 minutos.

Para criar uma forma de sair do sistema (logout). Iremos criar uma nova 'Action' no 'LoginController' da seguinte forma:

```
public IActionResult Sair()
{

}
```

Por padrão, ao não decorar a 'Action', ela será compreendida como o verbo GET. Então bastará chamar a URL '/Login/Sair' para executá-la.

O AspNet possui um método para realizar o logout e eliminar o cookie do usuário. Para isso, chamamos o método 'SignOutAsync' desta forma e em seguida por padrão podemos mandar o usuário para a tela de login caso ele deseje realizar o login com outro usuário para entrar novamente no sistema.

```
HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
return LocalRedirect("/Login");
```

Teste a aplicação:

```
> dotnet run
```

PASSO 04 - MELHORIAS DE USABILIDADE

É muito comum que uma vez que você esteja logado no sistema, seu nome apareça em algum lugar da interface indicando que você está devidamente autenticado. E próximo a esta informação podemos também incluir um botão para "sair".

Para isso, vamos incluir um 'div' simples com um link chamando o método 'Sair' do 'LoginController' que acabamos que criar no passo anterior.

Faremos isso, no arquivo _Layout.cshtml dentro da pasta /Views/Shared'. Faça isso, criando elementos de acordo com o bootstrap dentro do container do cabeçalho (header).

Também podemos acessar o nome do usuário através do 'Context.User.Identity.Name'.

```
<div id="user-info">
    <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 active" href="#">@Context!.User!.Identity!.Name</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" asp-controller="Login" asp-action="Sair">Sair</a>
            </li>
        </ul>
    </div>
</div>
```

Execute a aplicação e teste mais uma vez se está tudo funcionando corretamente.

```
> dotnet run
```

REFATOÇÃO MULTIPLOS USUÁRIOS

PASSO 01 - ESTRUTURA DO BANCO

Para conseguirmos filtrar as tarefas para cada usuário, primeiramente iremos adicionar a informação do usuário criador da tarefa à tabela de responsável por armazenar os dados de cada tarefa. Neste caso, bastará adicionar uma 'chave estrangeira', ou seja, um vínculo de registros entre as tabelas de 'Usuario' e 'Tarefa'. Onde a tarefa terá associado um Id do usuário criador.

Vamos refatorar a estrutua de criação da tabela na classe 'DatabaseBootstrap'.

```
con.Execute(
    @"CREATE TABLE Tarefa
    (
        Id          integer primary key autoincrement,
        Titulo      varchar(100) not null,
        Descricao   varchar(100) not null,
        Concluida   bool not null,
        UsuarioId   integer,
        FOREIGN KEY(UsuarioId) REFERENCES Usuario(Id)
    )"
);
```

- Você pode encontrar mais sobre o assunto em: https://sqlite.org/foreignkeys.html (https://sqlite.org/foreignkeys.html)

PASSO 02 - INSERINDO DADOS PARA TESTE

Vamos ampliar um pouco nosso método 'InsertDefaultData' para adicionar as informações de dois usuários e algumas tarefas para cada um deles. Apenas para não termos que cadastrar tarefas toda vez que estivermos testando a aplicação. Por motivos didátivos, o código mais simmples fica desta forma:

```
private void InsertDefaultData(SQLiteConnection con)
{
    var usuario1 = new UsuarioDTO()
    {
        Email = "andre@gmail.com",
        Senha = "biscoito",
        Nome = "André Paulovich",
        Ativo = true
    };

    var usuario2 = new UsuarioDTO()
    {
        Email = "ivan@gmail.com",
        Senha = "bolacha",
        Nome = "Ivan Paulovich",
        Ativo = true
    };

    con.Execute(
        @"INSERT INTO Usuario
        (Email, Senha, Nome, Ativo) VALUES
        (@Email, @Senha, @Nome, @Ativo);", usuario1
    );

    con.Execute(
        @"INSERT INTO Usuario
        (Email, Senha, Nome, Ativo) VALUES
        (@Email, @Senha, @Nome, @Ativo);", usuario2
    );

    var tarefa1 = new TarefaDTO()
    {
        Titulo = "Comprar pão",
        Descricao = "Passar na padaria da Vovó Alice e comprar 12 pães",
        Concluida = false
    };

    var tarefa2 = new TarefaDTO()
    {
        Titulo = "Levar o cachorro para pasear",
        Descricao = "Levar a Bia e a Pretinha para dar uma voltinha na Lagoa. Não esquecer de levar água para elas.",
        Concluida = false
    };

    var tarefa3 = new TarefaDTO()
    {
        Titulo = "Lavar o carro",
        Descricao = "Lavar e aspirar o carro. Lembrar que aspirar o porta-malas que está cheio de areia da praia.",
        Concluida = false
    };

    con.Execute(
        @"INSERT INTO Tarefa
        (Titulo, Descricao, Concluida, UsuarioId) VALUES
        (@Titulo, @Descricao, @Concluida, 1);", tarefa1
    );

    con.Execute(
        @"INSERT INTO Tarefa
        (Titulo, Descricao, Concluida, UsuarioId) VALUES
        (@Titulo, @Descricao, @Concluida, 1);", tarefa2
    );

    con.Execute(
        @"INSERT INTO Tarefa
```

```
        (Titulo, Descricao, Concluida, UsuarioId) VALUES
        (@Titulo, @Descricao, @Concluida, 2);", tarefa3
    );
}
```

Delete o arquivo 'Tarefas.WebAppTarefasDB.sqlite' e compile novamente a aplicação para gerar uma nova base de dados.

PASSO 03 - SEPARANDO AS TAREFAS POR USUÁRIO

Precisamos agora garantir que os usuários vejam apenas as tarefas que lhes pertencem. Para isso, vamos adicionar a informação do id do usuário nas consultas e na exclusão em seguida inserir o id do usuário no modelo que é atualizado e inserido.

A interface fica assim:

```
public interface ITarefaDAO
{   void Atualizar(TarefaDTO tarefa);
    List<TarefaDTO> Consultar(int usuarioId);
    TarefaDTO Consultar(int id, int usuarioId);
    void Criar(TarefaDTO tarefa);
    void Excluir(int id, int usuarioId);
}
```

Agora vamos refletir estas alterações na classe 'TarefaDAO'.

Modifique as querys para contemplar a nova coluna 'UsuarioId'. Por exemplo, no método de 'criação':

```
public void Criar(TarefaDTO tarefa)
{
    using (var con = Connection)
    {
        con.Open();
        con.Execute(
            @"INSERT INTO Tarefa
            (Titulo, Descricao, Concluida, UsuarioId) VALUES
            (@Titulo, @Descricao, @Concluida, @UsuarioId);", tarefa
        );
    }
}
```

E nas consultas adicionando a cláusula WHERE para filtrar apenas as tarefas que sejam do usuário associado ao id:

```
public List<TarefaDTO> Consultar(int usuarioId)
{
    using (var con = Connection)
    {
        con.Open();
        var result = con.Query<TarefaDTO>(
            @"SELECT Id, Titulo, Descricao, Concluida, UsuarioId FROM Tarefa Where UsuarioId = @UsuarioId", new { usuarioId }
        ).ToList();
        return result;
    }
}
```

Após alterar todas as querys necessárias na 'TarefaDAO', precisamos propagar a refatoração para o DTO e ViewModel.

Na classe 'TarefaDTO', iremos adicionar a informação do UsuarioId:

```
public int UsuarioId {get ; set; }
```

E no mesmo na 'TarefaViewModel':

```
public int UsuarioId { get; set; }
```

PASSO 04 - INFORMAÇÃO DO USUÁRIO LOGADO NO CONTROLLER

Precisaremos adicionar uma nova propriedade nas 'Claims' do usuário logado (que ficam armazenados no cookie). O cookie é um arquivo serializado em texto, por isso o Id precisará ser convertido em string.

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Nome),
    new Claim(ClaimTypes.Email, user.Email),
    new Claim(ClaimTypes.PrimarySid, user.Id.ToString())
};
```

No 'TarefaController', vamos adicionar o 'System.Security.Claims':

```
using System.Security.Claims;
```

E criar uma propriedade:

```
private readonly int _usuarioId;
```

Vamos injetar a dependência do 'HttpContext' e através deste 'serviço', acessar a 'Claim' que contém o Id do usuário, convertê-lo para 'int' novamente e armazená-lo na propriedade que acabamos de criar.

```
public TarefaController(ITarefaDAO tarefaDAO, IMapper mapper, IHttpContextAccessor httpContextAccessor)
{
    _tarefaDAO = tarefaDAO;
    _mapper = mapper;
    _usuarioId = int.Parse(httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.PrimarySid));
}
```

Agora vamos utilizar o '_usuarioId' em todos os métodos da Controller de acordo com as modificações que fizemos na 'TarefaDAO'.

```
public IActionResult Details(int id)
{
    var tarefaDTO = _tarefaDAO.Consultar(id, _usuarioId);
    /// demais implementações do método
}
```

Ou:

```
public IActionResult Delete(int id)
{
    _tarefaDAO.Excluir(id, _usuarioId);
    return RedirectToAction("Index");
}
```

O único cenário diferente será o de 'Create' (GET), pois neste caso, precisamos informar o Id do usuário através de uma 'ViewBag', pois não há model sendo enviada para a 'View':

```
public IActionResult Create()
{
    ViewBag.UsuarioId = _usuarioId;
    return View();
}
```

Inclusive, precisaremos adicionar um 'HiddenField' para armazenar e enviar esta informação, sempre que o formulário for postado. No arquivo: 'Views/Tarefa/Create.cshtm', inclua este input dentro da tag 'form'.

```
<input asp-for="UsuarioId" value="@ViewBag.UsuarioId" hidden />
```

Antes de testar a aplicação, precisamos apenas adicionar o 'HttpContextAccessor' ao serviço de injeção no statup da aplicação. Para isso, vamos na classe 'Program' e logo depois do builder.Services.AddAuthentication, vamos adicionar a seguinte linha:

```
builder.Services.AddHttpContextAccessor();
```

Agora vamos iniciar a aplicação e fazer o login com o usuário 'ivan@gmail.com' e em seguida com o usuário 'andre@gmail.com', criar e remover tarefas livremente. Se tudo estiver correto, os dois usuários só conseguirão afetar as informações que lhes pertence. E então podemos dizer que nosso experimento está completo.

```
dotnet run
```

Agora é só customizar seu projeto como quiser!