

INFORMÁTICA PARA INTERNET

Senac



Lógica de programação em sistemas web com banco de dados

Nas últimas UCs, aprendemos C#, exercitamos lógica, usamos projetos console no .NET Core e programamos para a *web* com o ASP.NET Core MVC.

Agora, é o momento de integrar esses conhecimentos com o que estamos aprendendo sobre banco de dados e aprender de novos recursos do MVC.

Novo projeto

Para explorar novos recursos do ASP.NET Core MVC, vamos iniciar um novo projeto, no qual será possível cadastrar usuários, realizar *log in* e definir restrições a usuários.

Vamos abrir o VS Code, criar e abrir uma pasta chamada **Cadastros**. Ainda no VS Code, vamos criar um novo projeto MVC, acionando o **Terminal** e digitando `dotnet new mvc --no-https`.

Iniciamos fazendo uma limpeza nos arquivos padrão gerados.

- ◆ Em **Controllers/HomeController.cs**, mantemos na classe **HomeController** apenas a ação **Index**.
- ◆ Em **Models**, excluimos **cs**
- ◆ Em **Views/Home**, excluimos **cshtml**
- ◆ Em **Views/Shared**, excluimos **_cshtml**, **_ValidationScriptsPartial.cshtml** e **Error.cshtml**
- ◆ Em **Views/Shared _Layout.cshtml**, alteramos seu conteúdo excluindo *links* e trechos que não serão utilizados

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>@ViewData["Title"] - Cadastros</title>
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css"
/>
  <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
  <header>
    <nav class="navbar navbar-light bg-white border-bottom box-shadow
mb-3">
      <a class="navbar-brand" asp-area="" asp-controller="Home"
asp-action="Index">Cadastros</a>
      <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
          <a class="nav-link text-dark" asp-area="" asp-con
troller="Home" asp-action="Index">Home</a>
        </li>
      </ul>
    </nav>
  </header>
  <main role="main">
    @RenderBody()
  </main>
  <footer class="border-top footer text-muted">
    &copy; 2019 - Cadastros
  </footer>
</body>
</html>
```

Não se preocupe com os erros que aparecerem no projeto, pois eles serão solucionados a seguir.

Na pasta **Models**, vamos criar uma classe (clcando com o botão direito e optando por **New C# class**) e nomeá-la como **Usuario**. Nela, incluiremos as seguintes propriedades: **Id**, **Nome**, **Login** e **Senha**.

```
namespace Cadastros.Models
{
    public class Usuario
    {
        public int Id {get; set;}
        public string Nome {get; set;}
        public string Login {get; set;}
        public string Senha {get; set;}
    }
}
```

Essa classe modelo precisa de um correspondente no banco de dados, por isso deixamos o projeto um momento de lado para criar o banco de dados. Aproveite para fazer um teste rápido usando dotnet run e visualizando a página inicial.

Usando um editor de SQL, como o MySQL Workbench ou PHPMyAdmin, execute o seguinte *script* de criação de banco de dados:

```
CREATE DATABASE ProjCadastros;
USE ProjCadastros;
CREATE TABLE Usuario(
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(255),
    login VARCHAR(255),
    senha VARCHAR(255)
);
```

Agora, podemos voltar ao VS Code e implementar a conexão e a manipulação do banco de dados. O primeiro passo é incluir a biblioteca de conexão em nosso projeto. Para isso, usamos a extensão NuGet Package Manager e buscamos pelo projeto **MySqlConnection** ou digitamos diretamente no terminal:

```
dotnet add package MySqlConnection
```

Para mais detalhes sobre a extensão NuGet e sobre a conexão com banco de dados, consulte o conteúdo **Lógica de programação aplicada a projetos C# com banco de dados** desta UC.

Na pasta **Model**, criamos **UsuarioRepository.cs**, em que implementamos a conexão com o banco de dados.

```
using MySql.Data.MySqlClient;
namespace Cadastros.Models
{
    public class UsuarioRepository
    {
        private const string _strConexao = "Database=ProjCadastros;Data S
ource=localhost;User Id=root;";
        public void Insert()
        {
            MySqlConnection conexao = new MySqlConnection(_strConexao);
            conexao.Open();
        }
    }
}
```

Note que já criamos um método *Insert*, que será completado a seguir. Neste momento, é interessante testar se a conexão com o banco está funcionando. Para isso, em **HomeController**, na ação **Index**, um objeto **UsuarioRepository** e invoque o método **Insert()**. Ao abrir a página inicial do projeto, se o **Terminal** do VS Code não acusar nenhuma falha, está tudo certo.

Por aqui, vamos completar o método **Insert** antes de algum teste.

```
public void Insert(Usuario novoUsuario)
{
    MySqlConnection conexao = new MySqlConnection(_strConexao);
    conexao.Open();
    string sql = "INSERT INTO usuario(nome, login, senha) VALUES (@Nome,
@Login, @Senha)";
    MySqlCommand comando = new MySqlCommand(sql, conexao);
    comando.Parameters.AddWithValue("@Nome", novoUsuario.Nome);
    comando.Parameters.AddWithValue("@Login", novoUsuario.Login);
    comando.Parameters.AddWithValue("@Senha", novoUsuario.Senha);
    comando.ExecuteNonQuery();
    conexao.Close();
}
```

Realizamos a conexão, montamos nossa instrução SQL, mas com um detalhe importante: para evitar riscos SQL Injection, é recomendável usar parâmetros em vez de o valor diretamente concatenado à cláusula SQL. Cada parâmetro é preenchido por meio do método **AddWithValue()** da lista de parâmetro **Parameters** do objeto comando.

Mais informações sobre segurança em projetos *web* estão disponíveis no conteúdo **Segurança da Informação no desenvolvimento web** desta UC.

Agora, vamos criar um formulário e um cadastro para usuários.

Na pasta **Controller**, vamos criar uma nova classe *controller* chamada de **UsuarioController**. Clicamos com o botão direito do *mouse* sobre a pasta **Controllers**, selecionamos **New C# class** e informamos **UsuarioController.cs** como nome do novo arquivo.

A primeira alteração necessária é incluir uma derivação à classe **Controller**. Pode ser necessário utilizar a ferramenta de lâmpada para incluir uma cláusula **using** no arquivo.

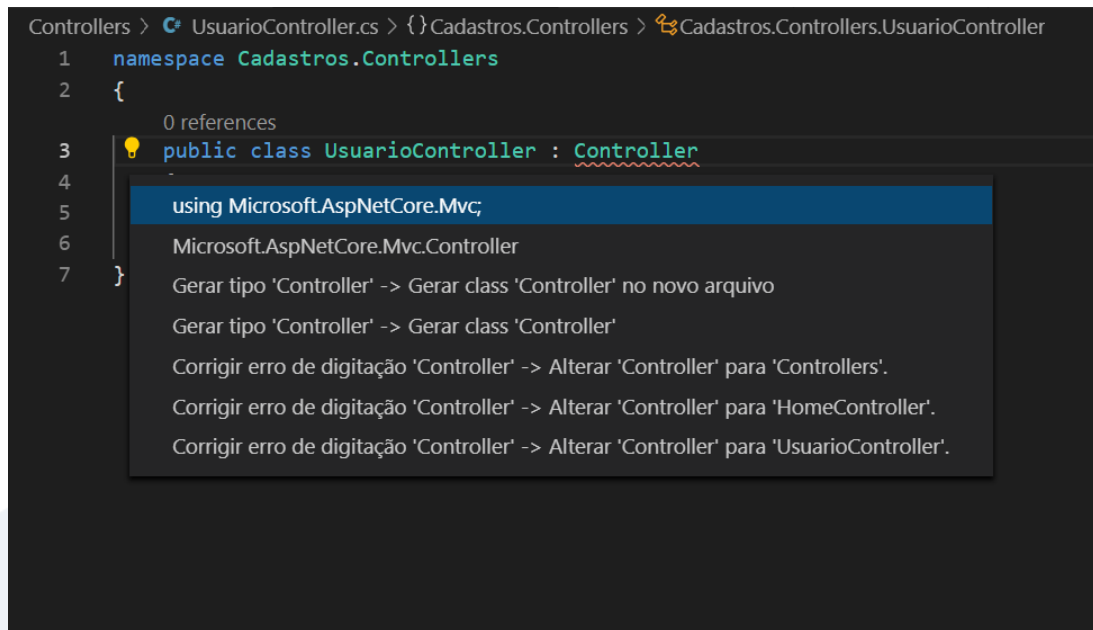


Figura 1 – tornando a classe UsuárioController uma classe derivada de Controller

no VS Code, o arquivo usuarioController.cs mostra “public class UsuarioController : Controller”. Controller está com erro; a lâmpada amarela aparece à esquerda e, clicando nela, aparece como primeira sugestão “using Microsoft.AspNetCore.Mvc;”

O que define uma classe de *controller* é justamente ela ser derivada de **Microsoft.AspNetCore.Mvc.Controller**. Em **UsuarioController**, incluímos nossa primeira ação.

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

Em seguida, criamos o *view* **Cadastro.cshtml** na pasta **Views/Usuario** (que também deve ser criada nesse momento).


```
@model Usuario
@{
    ViewData["Title"] = "Cadastro de Usuário";
}
<h2>Cadastro de Usuário</h2>
<p id="pMensagem">@ViewBag.Mensagem</p>
<form asp-action="Cadastro" method="POST">
    <p>
        <label asp-for="Nome">Nome:</label>
        <input asp-for="Nome" />
    </p>
    <p>
        <label asp-for="Login">Login:</label>
        <input asp-for="Login" />
    </p>
    <p>
        <label asp-for="Senha">Senha:</label>
        <input asp-for="Senha" />
    </p>
    <p>
        <input type="submit" value="Cadastrar" />
    </p>
</form>
```

Note que o *view* está marcado para usar como *model* a classe **Usuario** e que já reservou um espaço para um **ViewBag** com mensagem de sucesso ou de falha. Este é um bom momento para executar o projeto e visualizar se está tudo certo.

Seguindo no código, agora vamos criar uma ação específica para processar requisições do tipo **Post**.

```
using Cadastros.Models;
using Microsoft.AspNetCore.Mvc;
namespace Cadastros.Controllers
{
    public class UsuarioController : Controller
    {
        public IActionResult Cadastro()
        {
            return View();
        }
        [HttpPost]
        public IActionResult Cadastro(Usuario u)
        {
            return View();
        }
    }
}
```

A notação [HttpPost] serve para diferenciá-la da outra ação **Cadastro()** e significa que ela só será chamada por meio de uma requisição do tipo **Post** (ou seja, por formulário). Agora podemos incluir o código de inserção de dados no banco de dados.

```
[HttpPost]
public IActionResult Cadastro(Usuario u)
{
    UsuarioRepository ur = new UsuarioRepository();
    ur.Insert(u);
    ViewBag.Mensagem = "Usuario Cadastrado com sucesso";
    return View();
}
```

É importante compreender o caminho das requisições e das informações aqui.

1. O usuário chama a ação **Cadastro**, que apresenta o *view* **Cadastro** com o formulário.

2. O usuário preenche o formulário e submete. Neste momento, a aplicação organiza as informações preenchidas em um objeto do tipo **Usuario**, que é *model* do *view* **Cadastro**.
3. O **form** está configurado para enviar a informação à ação **Cadastro** via **Post**, por isso a ação **Cadastro(Usuario u)** é quem recebe suas informações.
4. O parâmetro é o *model* com as informações vindas do formulário.
5. Essa informação é usada para persistir **Usuario** no banco de dados usando **Insert()**;

É momento de testar o cadastro. Rode a aplicação e acesse, no navegador, a **URL /Usuario/Cadastro**.

[Cadastros](#) [Home](#)

Cadastro de Usuário

Nome:

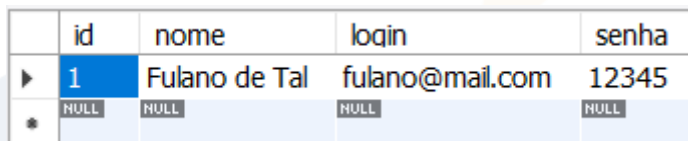
Login:

Senha:

Figura 2 – Página Usuario/Cadastro

título “Cadastro de Usuário”, campos “nome” com valor “Fulano de Tal”, “Login” com valor “fulano@mail.com” e “senha” com valor “12345”. Abaixo, botão cadastrar.

Esperamos que, após esse cadastro, a mensagem seja de “Usuário cadastrado com sucesso”. Isso nos dá liberdade para ir ao nosso editor de SQL e realizar uma consulta **Select** para verificar se o registro foi gravado.



	id	nome	login	senha
▶	1	Fulano de Tal	fulano@mail.com	12345
✱	NULL	NULL	NULL	NULL

Figura 3 – Resultado de `SELECT * FROM Usuario` realizado no MySQL Workbench

colunas com títulos “id”, “nome”, “login”, “senha” e “linha” com valores “1”, “Fulano de Tal”, “fulano@mail.com” e “12345”.

Veja que a senha está completamente visível. É recomendável que, em um cadastro, se use criptografia para esconder o valor real de valores como esse. Um uso comum é o MD5.

Voltando ao código, como um apoio, a seguir vamos implementar uma listagem. Primeiro, criamos um método **Query()** em **UsuarioRepository**. Este método vai retornar uma lista de **Usuario**.

```
public List<Usuario> Query()
{
    MySqlConnection conexao = new MySqlConnection(_strConexao);
    conexao.Open();
    string sql = "SELECT * FROM Usuario ORDER BY nome";
    MySqlCommand comandoQuery = new MySqlCommand(sql, conexao);
    MySqlDataReader reader = comandoQuery.ExecuteReader();
    List<Usuario> lista = new List<Usuario>();
    while (reader.Read())
    {
        Usuario usr = new Usuario();
        usr.Id = reader.GetInt32("Id");

        if(!reader.IsDBNull(reader.GetOrdinal("Nome")))
            usr.Nome = reader.GetString("Nome");

        if(!reader.IsDBNull(reader.GetOrdinal("Login")))
            usr.Login = reader.GetString("Login");
        if(!reader.IsDBNull(reader.GetOrdinal("Senha")))
            usr.Senha = reader.GetString("Senha");
        lista.Add(usr);
    }
    conexao.Close();
    return lista;
}
```

Não se esqueça de usar o assistente da lâmpada para completar as cláusulas **using** faltantes (no caso, para **List<>**, é necessário incluir `using System.Collections.Generic;`).

no código que estamos fazendo uma verificação com a seguinte condição:

```
if(!reader.IsDBNull(reader.GetOrdinal("Nome")))
```

Essa verificação objetiva evitar problema de conversão de valor nulo na tabela de banco de dados para o tipo especificado. Em outras palavras, estamos perguntando ao *reader* se a coluna **Nome** retornada não é nula.

Em seguida, criamos, em **UsuarioController**, uma ação chamada **Listar()**, que realiza a pesquisa e envia como informação de *model* ao *view*.

```
public IActionResult Listar()
{
    UsuarioRepository ur = new UsuarioRepository();
    List<Usuario> usuarios = ur.Query();
    return View(usuarios);
}
```

Por fim, em **Views/Usuario**, criamos um **view Listar.cshtml**.

```
@model IEnumerable<Usuario>
@{
    ViewData["Title"] = "Listagem de Usuário";
}
<h2>Listagem de Usuário</h2>
<table>
    <thead>
        <tr>
            <th>Id</th>
            <th>Nome</th>
            <th>Login</th>
            <th>Senha</th>
        </tr>
    </thead>
    <tbody>
        @foreach (Usuario u in Model)
        {
            <tr>
                <td>@u.Id</td>
                <td>@u.Nome</td>
                <td>@u.Login</td>
                <td>@u.Senha</td>
            </tr>
        }
    </tbody>
</table>
```

O *view* usa como *model* o tipo **IEnumerable<Usuario>** (**List<>** é uma classe derivada de **IEnumerable<>**) e usa a lista recebida para iterar na montagem da tabela.

Faça mais alguns cadastros para o resultado ficar interessante e depois teste a listagem acessando **/Usuario/Listar**.

Com tudo certo, é hora de fazer uma tela de *log in*.

Sessão: a memória de uma aplicação web

Uma das características mais marcantes de um sistema *web* (e uma das mais estranhas para desenvolvedores acostumados com o desenvolvimento *desktop*) é a falta de estado de uma aplicação. Uma requisição acontece de maneira independente de outra e não há exatamente uma memória na aplicação, uma variável que possamos usar para guardar uma informação e consultar essa informação algumas páginas e algumas requisições depois (algo trivial em uma aplicação *desktop*).

Para esse tipo de situação, as aplicações *web* implementam um conceito de **sessão**, que é uma espécie de memória do servidor associado ao usuário que armazena uma informação e perpassa por várias requisições realizadas no sistema.

Um caso muito comum de uso de sessão é para identificar um usuário autenticado. Veja que em *sítes* de compras, ou mesmo no seu ambiente virtual de aprendizado, o nome do usuário geralmente aparece em um canto das páginas oferecendo muitas vezes uma opção de *logout*. Isso é possível graças às sessões.

Antes de implementar uma sessão no ASP.NET Core MVC, vamos construir uma consulta e uma página para o *log in*.

Em **UsuarioRepository**, acrescentamos um método **QueryLogin()** com o seguinte código:

```
public Usuario QueryLogin(Usuario u)
{
    MySqlConnection conexao = new MySqlConnection(_strConexao);
    conexao.Open();
    string sql = "SELECT * FROM Usuario WHERE login = @Login AND senha = @Senha";
    MySqlCommand comandoQuery = new MySqlCommand(sql, conexao);
    comandoQuery.Parameters.AddWithValue("@Login", u.Login);
    comandoQuery.Parameters.AddWithValue("@Senha", u.Senha);
    MySqlDataReader reader = comandoQuery.ExecuteReader();
    Usuario usr = null;
    if(reader.Read())
    {
        usr = new Usuario();
        usr.Id = reader.GetInt32("Id");
        if(!reader.IsDBNull(reader.GetOrdinal("Nome")))
            usr.Nome = reader.GetString("Nome");

        if(!reader.IsDBNull(reader.GetOrdinal("Login")))
            usr.Login = reader.GetString("Login");
        if(!reader.IsDBNull(reader.GetOrdinal("Senha")))
            usr.Senha = reader.GetString("Senha");
    }

    conexao.Close();
    return usr;
}
```

A consulta busca por usuários que tenham exatamente o mesmo *log in* e a mesma senha informados e retorna um objeto com as informações desse usuário. Nesse caso, não precisamos retornar uma lista, apenas o primeiro registro de **Usuario** encontrado. O método retorna nulo caso não encontre um usuário com *log in* e senha (reader não traz nenhum registro).

Agora, em **UsuarioController**, criaremos duas ações chamadas **Login()**. A primeira será responsável por mostrar a página de autenticação; a segunda será usada para receber as informações que o usuário digitou e

fazer o *log in* de fato.

```
public IActionResult Login()
{
    return View();
}
[HttpPost]
public IActionResult Login(Usuario u)
{
    UsuarioRepository ur = new UsuarioRepository();
    Usuario usuario = ur.QueryLogin(u);
    if(usuario != null)
    {
        ViewBag.Mensagem = "Você está logado";
        return Redirect("Cadastro");
    }
    else
    {
        ViewBag.Mensagem = "Falha no Login";
        return View();
    }
}
```

Note que a segunda ação **Login()** (marcada com [HttpPost]) realiza a busca do log in e, em caso de sucesso (usuário retornado não é nulo), redireciona para o *view* **Cadastro**. Em caso de falha, redireciona para o *view* **Login**.

Os métodos **Redirect()** ou **RedirectToAction()** realizam a chamada do método de ação especificado no parâmetro (nesse caso, **Cadastro()**). É muito diferente de simplesmente chamar **View("Cadastro")**;, que apenas chamaria o *view* **Cadastro.cshtml**, sem executar sua ação.

Agora, podemos criar nosso *view* **Login.cshtml** em **Views/Usuario**.

```
@model Usuario
@{
    ViewData["Title"] = "Login";
}
<h2>Login</h2>
<p id="pMensagem">@ViewBag.Mensagem</p>
<form asp-action="Login" method="POST">
    <p>
        <label asp-for="Login">Login:</label>
        <input asp-for="Login" />
    </p>
    <p>
        <label asp-for="Senha">Senha:</label>
        <input asp-for="Senha" type="password" />
    </p>
    <p>
        <input type="submit" value="Entrar" />
    </p>
</form>
```

Podemos testar usando a **localhost:5000/Usuario/Login**. Não se esqueça de encerrar o servidor Kestrel caso ele esteja rodando.

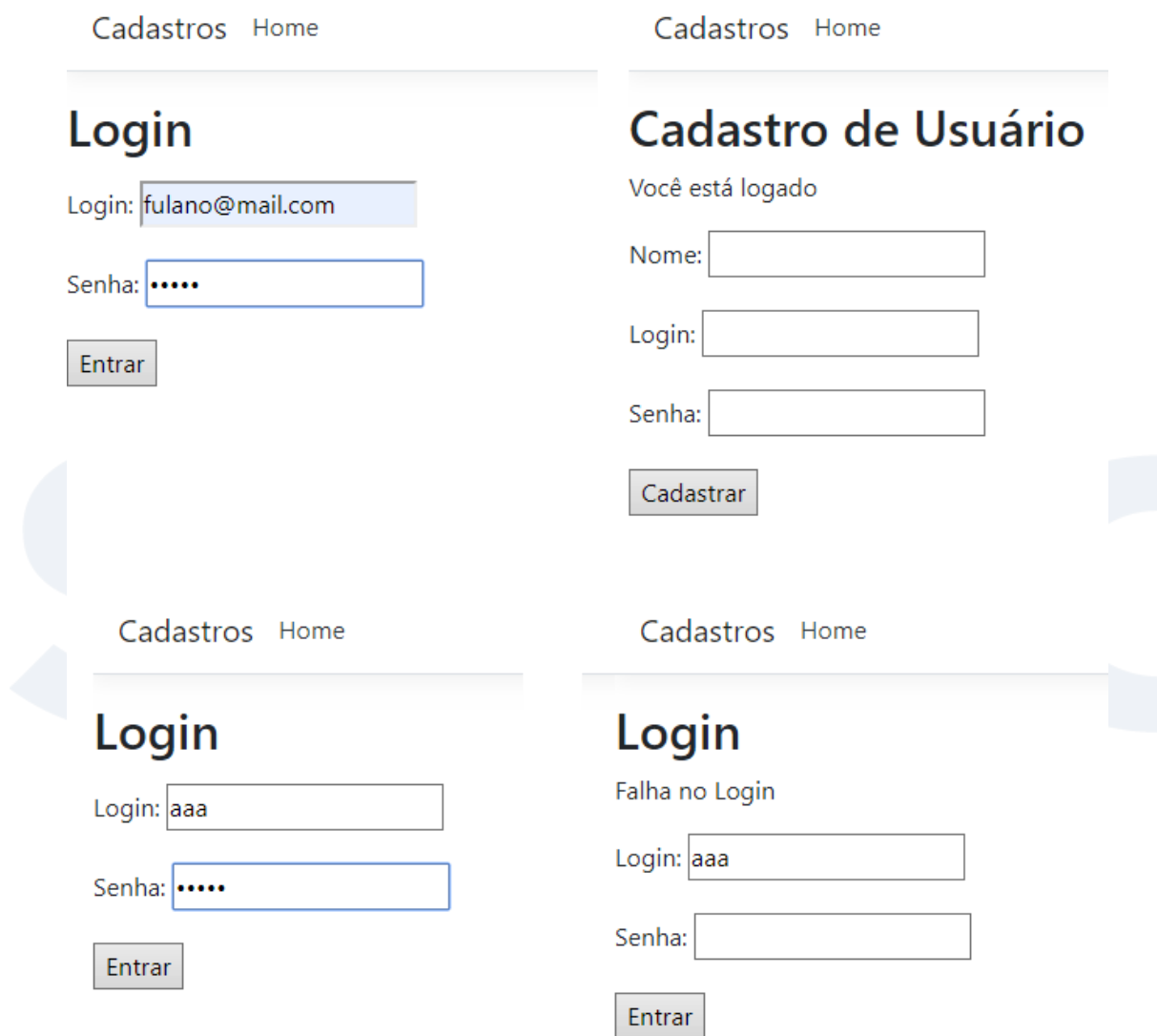


Figura 4 – Situações de teste

imagem dividida em quatro partes. Na parte superior esquerda, há login fulano@mail.com e senha oculta. Na direita, temos Cadastro de Usuário com mensagem “você está logado”. Na parte inferior esquerda, temos Login com “aaa” e senha oculta. À direita, aparece Login com mensagem “Falha no login”.

Agora que temos um *log in* funcional, podemos trabalhar com sessão. Primeiro, é necessário configurar **Startup.cs** para habilitar sessões. No método **ConfigureServices**, é necessário acrescentar chamadas `services.AddMemoryCache();` e `services.AddSession();`. Também é

necessário mudar a configuração `options.CheckConsentNeeded` para **false**. No método **Configure()**, após `app.UseCookiePolicy();`, acrescenta-se `app.UseSession();`. A classe **Startup** fica como na listagem a seguir (as chamadas acrescentadas estão em destaque):

A large, light blue watermark of the Senac logo is centered on the page. It features a stylized star above the word "Senac" in a bold, sans-serif font.

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
    public IConfiguration Configuration { get; }
    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<CookiePolicyOptions>(options =>
        {
            options.CheckConsentNeeded = context => false;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });
        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
        services.AddMemoryCache();
        services.AddSession();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
        }
        app.UseStaticFiles();
        app.UseCookiePolicy();
        app.UseSession();
        app.UseMvc(routes =>
        {
            routes.MapRoute(name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

O método **AddMemoryCache()** configura uma sessão em memória (há outros tipos de sessão, como sessão armazenada em banco de dados, mas aqui usaremos a que usa a memória do servidor – se o servidor parar,

a sessão é apagada). O método **AddSession()** registra o serviço usado para acessar dados de sessão e **UseSession()** permite que o sistema associe automaticamente requisições de usuários com as sessões.

A configuração **options.CheckConsentNeeded** é relacionada com aquela mensagem comum no leiaute padrão do MVC, mencionando que o usuário precisa aceitar *cookies*. Caso o valor esteja verdadeiro, a sessão não será gravada a menos que o usuário aceite o uso de *cookies* no *site*. Para o nosso exemplo, basta configurá-la como **false** para que esse tipo de verificação não ocorra.

Agora é momento de utilizar a sessão. Vamos preenchê-la quando o *log in* for bem-sucedido. Em **UsuarioController**, no método **Login()**, incluímos dados em uma sessão usando os métodos **HttpContext.Session.SetInt32()** e **HttpContext.Session.SetString()**. É importante saber que **SetString()** e **SetInt32()** são métodos de extensão e por isso é necessário incluir manualmente a seguinte cláusula *using* no topo do arquivo:

```
using Microsoft.AspNetCore.Http;
```

Agora, no método **Login**, vamos incluir uma entrada na sessão para o *id* do usuário autenticado e outra para o nome.

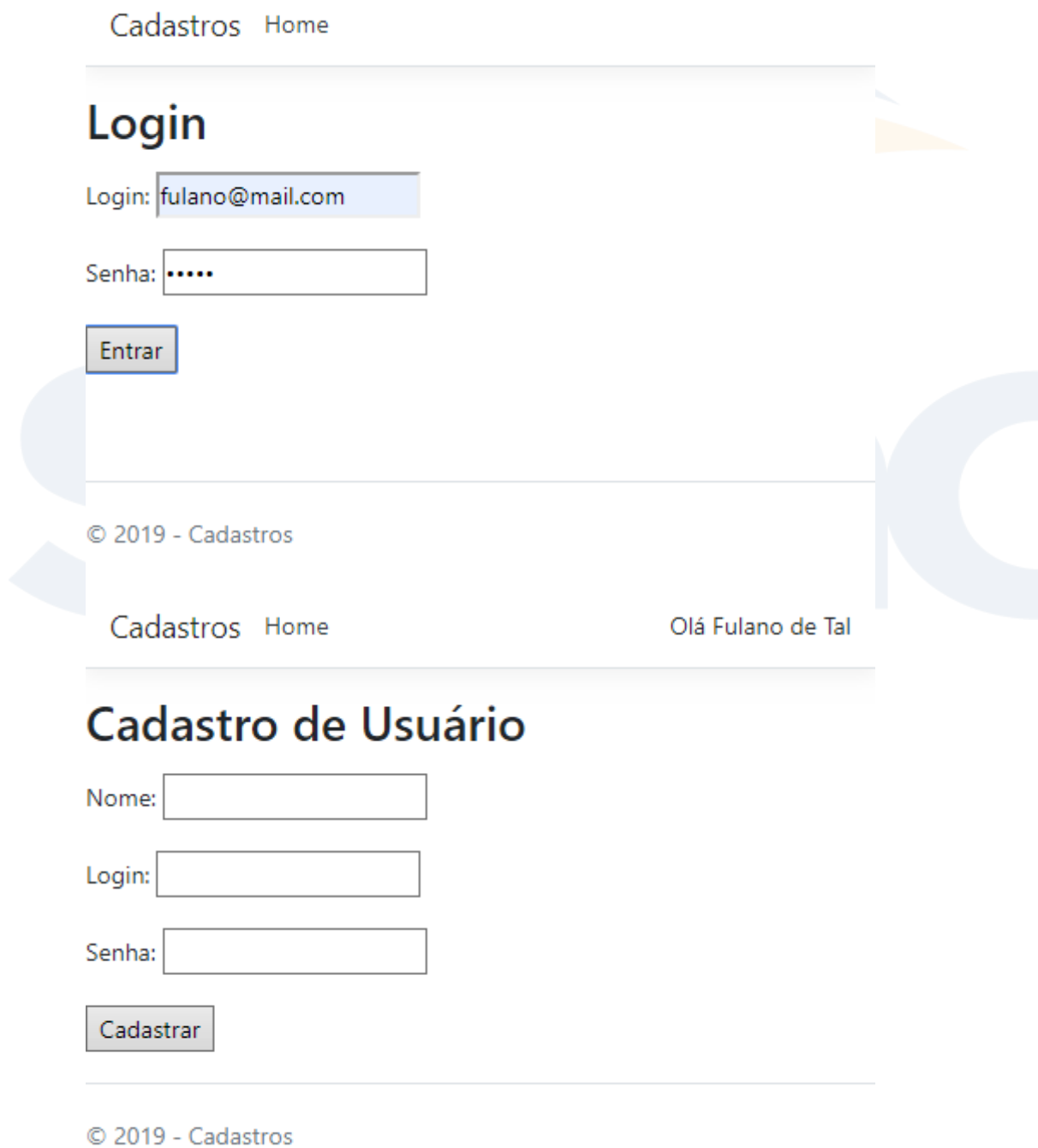
```
[HttpPost]
public IActionResult Login(Usuario u)
{
    UsuarioRepository ur = new UsuarioRepository();
    Usuario usuario = ur.QueryLogin(u);
    if(usuario != null)
    {
        ViewBag.Mensagem = "Você está logado";
        HttpContext.Session.SetInt32("idUsuario", usuario.Id);
        HttpContext.Session.SetString("nomeUsuario", usuario.Nome);
        return View("Cadastro");
    }
    else
    {
        ViewBag.Mensagem = "Falha no Login";
        return View();
    }
}
```

Vamos incluir as boas-vindas ao nosso usuário em **Views/Shared/_Layout.cshtml**, na *tag* **<header>**, que verifica se a sessão está nula; caso não esteja, escreva “Olá [usuario]”. Também é necessário acrescentar **@using Microsoft.AspNetCore.Http** no topo do CSHTML. Os trechos alterados estão em destaque na listagem a seguir.

```
@using Microsoft.AspNetCore.Http
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewData["Title"] - Cadastros</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css"
    />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-light bg-white border-bottom box-shadow
mb-3">
            <a class="navbar-brand" asp-area="" asp-controller="Home"
asp-action="Index">Cadastros</a>
            <ul class="navbar-nav flex-grow-1">
                <li class="nav-item">
                    <a class="nav-link text-dark" asp-area="" asp-con
troller="Home" asp-action="Index">Home</a>
                </li>
            </ul>
            @if(Context.Session.GetInt32("idUsuario") != null)
            {
                <span>Olá @Context.Session.GetString("nomeUsuario")</span
            }
        </nav>
    </header>
    <main role="main">
        @RenderBody()
    </main>
    <footer class="border-top footer text-muted">
        &copy; 2019 - Cadastros
    </footer>
</body>
</html>
```

É importante notar que, nos *views*, em vez de **HttpContext.Session**, acessamos a sessão simplesmente via **Context.Session**.

O resultado pode ser visto na figura a seguir.



Cadastros Home

Login

Login:

Senha:

© 2019 - Cadastros

Cadastros Home Olá Fulano de Tal

Cadastro de Usuário

Nome:

Login:

Senha:

© 2019 - Cadastros

Figura 5 – Tela de login e tela seguinte mostrando a mensagem de “Olá” no canto superior direito

À esquerda tela de login, aparece log in e senha preenchidos. À direita, temos a tela de cadastro mostrando, no canto superior direito, a mensagem “Olá Fulano de Tal”.

Podemos usar sessões para bloquear acesso a algumas páginas a menos que o usuário esteja autenticado. Vamos experimentar fazendo uma verificação de sessão com o seguinte condicional:

```
if(HttpContext.Session.GetInt32("idUsuario") == null)
    return RedirectToAction("Login");
```

Ele verifica se a chave **idUsuario** da sessão está nula e, nesse caso, redireciona a ação em execução para a ação **Login**. Podemos usá-lo em **Cadastro()**:

```
public IActionResult Cadastro()
{
    if(HttpContext.Session.GetInt32("idUsuario") == null)
        return RedirectToAction("Login");
    return View();
}
```

E em **Listar()**:

```
public IActionResult Listar()
{
    if(HttpContext.Session.GetInt32("idUsuario") == null)
        return RedirectToAction("Login");
    UsuarioRepository ur = new UsuarioRepository();
    List<Usuario> usuarios = ur.Query();
    return View(usuarios);
}
```

Agora, tente finalizar o servidor Kestrel (o que faz a sessão ser destruída), realizar compilação e executar novamente o projeto. Tente acessar **/Usuario/Listar** e verifique que o sistema mandará de volta para a tela de *log in*.

Há maneiras muito mais eficientes de realizar esse controle no ASP.NET Core MVC. A mais indicada, nesse caso, seria a criação de um atributo do tipo **[Authorize]**. Mas a implementação apresentada é a mais genérica e a lógica é aplicável a qualquer linguagem de programação *web*.

O código final de **UsuarioController** ficou da seguinte forma:

Senac

```
using System.Collections.Generic;
using Cadastros.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
namespace Cadastros.Controllers
{
    public class UsuarioController : Controller
    {
        public IActionResult Cadastro()
        {
            if (HttpContext.Session.GetInt32("idUsuario") == null)
                return RedirectToAction("Login");
            return View();
        }
        [HttpPost]
        public IActionResult Cadastro(Usuario u)
        {
            UsuarioRepository ur = new UsuarioRepository();
            ur.Insert(u);
            ViewBag.Mensagem = "Usuario Cadastrado com sucesso";
            return View();
        }
        public IActionResult Listar()
        {
            if (HttpContext.Session.GetInt32("idUsuario") == null)
                return RedirectToAction("Login");
            UsuarioRepository ur = new UsuarioRepository();
            List<Usuario> usuarios = ur.Query();
            return View(usuarios);
        }
        public IActionResult Login()
        {
            return View();
        }
        [HttpPost]
        public IActionResult Login(Usuario u)
        {
            UsuarioRepository ur = new UsuarioRepository();
            Usuario usuario = ur.QueryLogin(u);
            if (usuario != null)
            {
                ViewBag.Mensagem = "Você está logado";
                HttpContext.Session.SetInt32("idUsuario", usuario.Id);
                HttpContext.Session.SetString("nomeUsuario", usuario.Nome);
            }
            return Redirect("Cadastro");
        }
        else
    }
}
```

```
        {  
            ViewBag.Mensagem = "Falha no Login";  
            return View();  
        }  
    }  
}
```

Aqui está o código completo de **UsuarioRepository**:

Senac

```
using System.Collections.Generic;
using MySql.Data.MySqlClient;
namespace Cadastros.Models
{
    public class UsuarioRepository
    {
        private const string _strConexao = "Database=ProjCadastros;Data Source=localhost;User Id=root;";

        public void Insert(Usuario novoUsuario)
        {
            MySqlConnection conexao = new MySqlConnection(_strConexao);
            conexao.Open();
            string sql = "INSERT INTO usuario(nome, login, senha) VALUES (@Nome, @Login, @Senha)";
            MySqlCommand comando = new MySqlCommand(sql, conexao);
            comando.Parameters.AddWithValue("@Nome", novoUsuario.Nome);
            comando.Parameters.AddWithValue("@Login", novoUsuario.Login);
            comando.Parameters.AddWithValue("@Senha", novoUsuario.Senha);
            comando.ExecuteNonQuery();
            conexao.Close();
        }

        public List<Usuario> Query()
        {
            MySqlConnection conexao = new MySqlConnection(_strConexao);
            conexao.Open();
            string sql = "SELECT * FROM Usuario ORDER BY nome";
            MySqlCommand comandoQuery = new MySqlCommand(sql, conexao);
            MySqlDataReader reader = comandoQuery.ExecuteReader();
            List<Usuario> lista = new List<Usuario>();
            while (reader.Read())
            {
                Usuario usr = new Usuario();
                usr.Id = reader.GetInt32("Id");
                if (!reader.IsDBNull(reader.GetOrdinal("Nome")))
                    usr.Nome = reader.GetString("Nome");
                if (!reader.IsDBNull(reader.GetOrdinal("Login")))
                    usr.Login = reader.GetString("Login");
                if (!reader.IsDBNull(reader.GetOrdinal("Senha")))
                    usr.Senha = reader.GetString("Senha");
                lista.Add(usr);
            }
            conexao.Close();
            return lista;
        }

        public Usuario QueryLogin(Usuario u)
        {
            MySqlConnection conexao = new MySqlConnection(_strConexao);
            conexao.Open();
```

```
        string sql = "SELECT * FROM Usuario WHERE login = @Login AND  
        senha = @Senha";  
        MySqlCommand comandoQuery = new MySqlCommand(sql, conexao);  
        comandoQuery.Parameters.AddWithValue("@Login", u.Login);  
        comandoQuery.Parameters.AddWithValue("@Senha", u.Senha);  
        MySqlDataReader reader = comandoQuery.ExecuteReader();  
        Usuario usr = null;  
        if (reader.Read())  
        {  
            usr = new Usuario();  
            usr.Id = reader.GetInt32("Id");  
            if (!reader.IsDBNull(reader.GetOrdinal("Nome")))  
                usr.Nome = reader.GetString("Nome");  
            if (!reader.IsDBNull(reader.GetOrdinal("Login")))  
                usr.Login = reader.GetString("Login");  
            if (!reader.IsDBNull(reader.GetOrdinal("Senha")))  
                usr.Senha = reader.GetString("Senha");  
        }  
        conexao.Close();  
        return usr;  
    }  
}
```

Tivemos muito trabalho até aqui. Se estiver se sentindo motivado, tente realizar o desafio a seguir.

Implemente uma ação **Logout** em **UsuarioController**. Ela terá o seguinte código:

```
public IActionResult Logout()  
{  
    HttpContext.Session.Clear();//limpa toda a sessão  
    return View();  
}
```

Faça um *link* para **Logout** ao lado da saudação de “Olá”, incluída no canto superior direito do leiaute.

Considerações finais

Aqui seguimos nos estudos e nas experiências com o ASP.NET Core MVC treinando dois aspectos muito importantes. A integração com banco de dados é base para praticamente qualquer sistema e o uso de sessões é algo muito comum em páginas dinâmicas (em *log ins* e carrinhos de compra em lojas *on-line*, por exemplo).

Ainda há mais recursos a explorar, como as validações de **Model**, a customização de rotas e o uso de **PartialViews**, mas isso fica para as próximas unidades.

