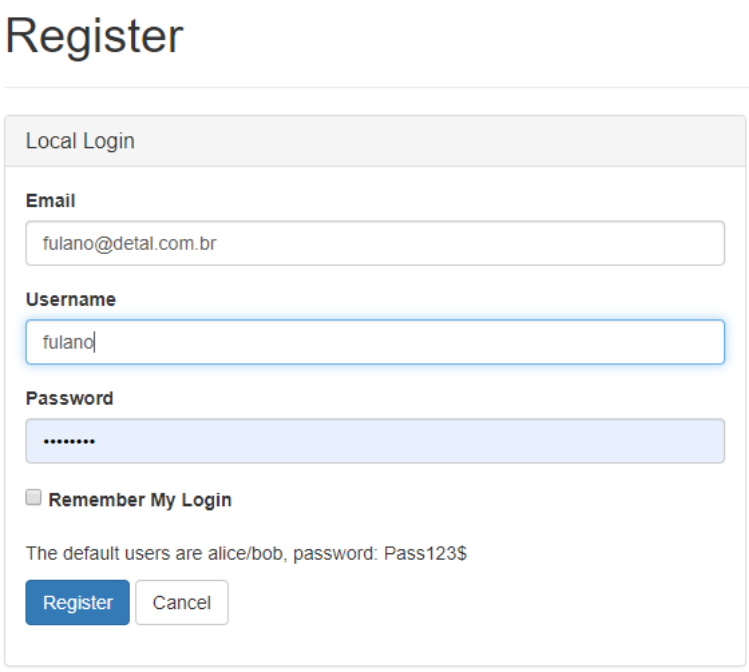
12

Criando a Página de Registro de Usuário

Tutorial: Criando uma Página de Registro de Usuário

Seja bem-vindo à **atividade extra** do curso ASP.NET Core Parte 4: IdentityServer!

Na pasta **[Item07]** você encontra projetos com **código extra** que não foi mostrado nesse curso. Nesses projetos, temos uma **nova página de registro de usuários**, que não aparece nos vídeos do curso.



Esta é uma atividade **opcional**. Você pode simplesmente rodar os projetos do Item07 para ver como o registro de usuários funciona. Se você quiser, pode pular esta atividade e finalizar o curso, sem problemas! Mas se fizer este tutorial, você aprenderá a criar uma nova página de registro de usuário para o projeto CasaDoCodigo.IdentityServer. Você poderá aplicar esse conhecimento em projetos no seu dia a dia!

Neste curso, aprendemos a criar, configurar e integrar um projeto IdentityServer ao restante da solução da Casa do Código. Como vimos, é possível fazer login através da `view Login` do IdentityServer. Para implementar essa view e sua action correspondente, o projeto IdentityServer utiliza o ciclo de vida de usuário do ASP.NET Core Identity.

O problema é que o template projeto IdentityServer, que foi criado automaticamente, não possui uma página nem action para registrar novos usuários. Se quisermos, teremos que implementar manualmente uma nova página de registro de usuários. Então, neste tutorial, vamos aprender a implementar essa nova funcionalidade para o registro de usuários.

1. Criando a pasta Item07

Como você deve ter percebido, existem na solução as pastas **[antes]** e **[depois]**. Dentro da pasta **[antes]**, faça uma cópia da pasta **[Item06]** para uma nova pasta **[Item07]**. A partir daí, defina os 3 projetos dentro de **[Item07]** como ***projetos de inicialização (startup projects)*** da solução.

Feito isso, as instruções abaixo vão orientar e explicar as mudanças necessárias para criação da nova página de registro de usuário.

2. Criando o modelo de registro

O projeto CasaDoCodigo.IdentityServer também é um projeto MVC, portanto, precisamos criar o modelo, criar a view e adaptar o controller para trabalhar com registro de usuários.

Vamos começar pelo modelo de registro de novos usuários e criar uma classe de modelo com dados a serem exibidos na view de login. Como o conjunto de dados necessários é semelhante à view de login, fazemos uma cópia da classe `LoginInputModel` e a chamamos de `RegisterInputModel`. Porém, a classe `LoginInputModel` não possui a informação de e-mail. Portanto, adicionamos uma nova propriedade para o e-mail do usuário:

```
        [Required]
        public string Email { get; set; }
```

Então, copiamos o código da classe `LoginViewModel` para uma nova classe `RegisterViewModel`, mudando o nome, assim:

```
    public class RegisterViewModel : RegisterInputModel
```

Arquivo: \Item07\CasaDoCodigo.IdentityServer\Quickstart\Account\RegisterViewModel.cs

Vamos para a classe `AccountController`, na qual iremos criar um novo método `BuildRegisterViewModelAsync`, copiado a partir do método preexistente `BuildLoginViewModelAsync`. Em seguida, vamos modificar estas 3 linhas:

```
private async Task<RegisterViewModel> BuildRegisterViewModelAsync(string returnUrl)
...
return new RegisterViewModel
...
return new RegisterViewModel
```

Arquivo: \Item07\CasaDoCodigo.IdentityServer\Quickstart\Account\AccountController.cs

Agora fazemos uma cópia do método `BuildLoginViewModelAsync(LoginInputModel model)` e o chamamos de `BuildRegisterViewModelAsync(RegisterInputModel model)`:

```
private async Task<RegisterViewModel> BuildRegisterViewModelAsync(RegisterInputModel model)
. . .
var vm = await BuildRegisterViewModelAsync(model.ReturnUrl);
```

3. Criando as actions de registro

Ótimo, já temos as classes que servirão de modelo. Agora partiremos para a action de registro de novos usuários. Para isso, criaremos o método `Register()` abaixo, na classe `AccountController`:

```
/// <summary>
/// Exibe a página de registro de usuários
/// </summary>
[HttpGet]
public async Task<IActionResult> Register(string returnUrl)
{
```

```
    // construir um modelo com dados a serem exibidos na view de login
    var vm = ...

    return View(vm);
}
```

Arquivo: \Item07\CasaDoCodigo.IdentityServer\Quickstart\Account\AccountController.cs

Note que, ao final da action, a view referente a essa action é retornada.

Mas a viewmodel, representada aqui pela variável `vm`, precisa ser inicializada com os dados que serão injetados na view de `Register.cshtml`.

Esse novo método toma como parâmetro a url de retorno, isto é, a url para onde o usuário será redirecionado ao final do processo de registro.

```
[HttpGet]
public async Task<IActionResult> Register(string returnUrl)
{
    // construir um modelo com dados a serem exibidos na view de login
    var vm = await BuildRegisterViewModelAsync(returnUrl);

    return View(vm);
}
```

Agora vamos criar um outro método `Register` no controller, porém, desta vez, com uma assinatura diferente: esta sobrecarga (overload) recebe um parâmetro `RegisterInputModel`.

```
/// <summary>
/// Handle postback from username/password register
/// </summary>
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterInputModel model, string button)
{
    if (button != "register")
    {

    }
}
```

O próximo passo é verificar o *consentimento* do usuário. Isso é feito no objeto `_interaction`, através dos métodos `GetAuthorizationContextAsync()` e `GrantConsentAsync()`.

O objeto `_interaction` é uma instância da interface `IIdentityServerInteractionService`, que fornece serviços a serem usados pela interface do usuário para se comunicar com o IdentityServer. O método obtém o contexto de autorização, enquanto `GrantConsentAsync()` notifica o IdentityServer sobre o consentimento do usuário.

```
if (button != "register")
{
```

```
    // o usuário clicou no botão "cancelar"
    var context = await _interaction.GetAuthorizationContextAsync(model.ReturnUrl);
    if (context != null)
    {
        // se um usuário cancela, enviamos o resultado de volta para o IdentityServer como acesso ne
        await _interaction.GrantConsentAsync(context, ConsentResponse.Denied);

        // podemos confiar em model.ReturnUrl, pois GetAuthorizationContextAsync retornou não-nulo
        return Redirect(model.ReturnUrl);
    }
    else
    {
        // já que não temos um contexto válido, voltamos para a página inicial
        return Redirect("~/");
    }
}
```

Se o usuário clicou em "registrar" e o estado do modelo é válido, então criamos uma nova instância de usuário da aplicação (`ApplicationUser`) e criamos o usuário na base de dados SQLite do projeto, utilizando o método `CreateAsync()` da classe `UserManager` do ASP.NET Core Identity:

```
if (button != "login")
{
    .
    .
    .
}

if (ModelState.IsValid)
{
    var user = new ApplicationUser { UserName = model.Username, Email = model.Email };
    var result = await _userManager.CreateAsync(user, model.Password);
}
```

O próximo passo é definir os dados do usuário criado. Como vimos no curso, as *declarações* (claims) são usadas para obter esses dados, incluindo e-mail e nome do usuário. Aqui, vamos chamar o método `AddClaimsAsync` da classe `UserManager` para adicionar as declarações ao usuário criado:

```
if (result.Succeeded)
{
    await _signInManager.SignInAsync(user, isPersistent: false);

    var claimsResult = _userManager.AddClaimsAsync(user, new Claim[]{
        new Claim("name", user.UserName),
        new Claim(JwtClaimTypes.GivenName, ""),
        new Claim(JwtClaimTypes.FamilyName, ""),
        new Claim("email", user.Email),
        new Claim(JwtClaimTypes.EmailVerified, "true", ClaimValueTypes.Boolean),
    }).Result;
    if (!claimsResult.Succeeded)
    {
        throw new Exception(result.Errors.First().Description);
    }
```

```
    return Redirect(model.ReturnUrl);
}
```

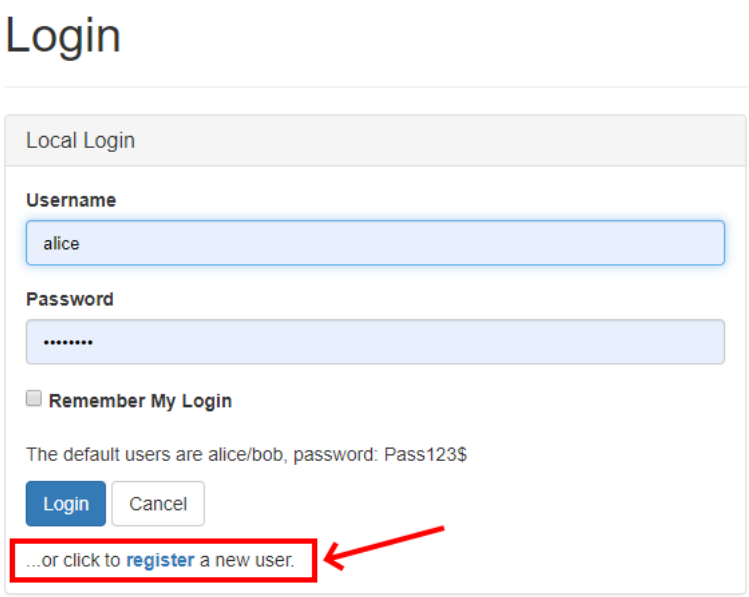
Ao final da action, caso algo tenha dado errado, uma mensagem de erro será incluída no modelo para ser exibida na view de registro:

```
            .
            .
            .
            return Redirect(model.ReturnUrl);
        }

        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(error.Code, error.Description);
        }
    }

    // algo deu errado, vamos mostar a view com o erro
    var vm = await BuildRegisterViewModelAsync(model);
    return View(vm);
}
```

4. Criando e modificando views



Vamos fazer uma cópia do arquivo `Login.cshtml` e o chamamos de `Register.cshtml`.

Nesta nova view, vamos adicionar o suporte para o novo campo `Email` e trocar o botão `Login` para `Register`:

```
@model RegisterViewModel
...
<form asp-route="Register">
...
    <fieldset>
        <div class="form-group">
            <label asp-for="Email"></label>
```

```
            <input class="form-control" placeholder="Email" asp-for="Email" autofocus>
        </div>
        <div class="form-group">
            <label asp-for="Username"></label>
            <input class="form-control" placeholder="Username" asp-for="Username">
        </div>
...
        <div class="form-group">
            <button class="btn btn-primary" name="button" value="register">Register</button>
            <button class="btn btn-default" name="button" value="cancel">Cancel</button>
        </div>
```

Arquivo: \Item07\CasaDoCodigo.IdentityServer\Views\Account\Register.cshtml

Outra modificação necessária é importar o namespace `@using CasaDoCodigo.IdentityServer.Quickstart.Account` no arquivo de imports da aplicação MVC:

```
@using IdentityServer4.Quickstart.UI
@using CasaDoCodigo.IdentityServer.Quickstart.Account
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Arquivo: \Item07\CasaDoCodigo.IdentityServer\Views_ViewImports.cshtml

Por fim, vamos modificar a view de login para incluir um ponto de acesso à nova view de registro. Isso é feito através de uma simples `AnchorTagHelper` (`<a>`) apontando para a action `Register` de `AccountController` .

```
...
<div>
    ...or click to <a asp-controller="account" asp-action="register" asp-route-returnUrl="@Model.Ret
</div>
...
```

Arquivo: \Item07\CasaDoCodigo.IdentityServer\Views\Account\Login.cshtml

5. Habilitando o e-mail no cadastro do usuário

Durante o curso, vimos que o e-mail não estava disponível no acesso aos dados do usuário. Podemos adicionar o e-mail às declarações do usuário (claims) através dos seguintes passos.

Primeiro, modifique as configurações no projeto IdentityServer para 1) incluir o e-mail como um recurso de identidade (`IdentityResource`) e 2) incluir o e-mail também como um escopo permitido ao cliente CasaDoCodigo.MVC:

```
new IdentityResources.Email()
.
.
.
AllowedScopes = { "openid", "profile", "email", "CasaDoCodigo.Relatorio" }
```

Arquivo: \Item07\CasaDoCodigo.IdentityServer\Config.cs

Agora, na classe `PedidoController`, atribua o valor do e-mail da declaração do usuário ao objeto de cadastro do modelo:

```
pedido.Cadastro.Email = User.FindFirst("email")?.Value;
```

Arquivo: \Item07\CasaDoCodigo\Controllers\PedidoController.cs

Por fim, entre na classe `Startup` do projeto MVC da Casa do Código e inclua o e-mail na lista de escopos do OpenId Connect:

```
options.Scope.Add("email");
```

Arquivo: \Item07\CasaDoCodigo\Startup.cs

Pronto! Agora você já sabe como criar uma página de registro de usuário no projeto IdentityServer!