02

# Segurança com o Simple Membership

Agora que já implementamos as funcionalidades básicas da aplicação, precisamos nos preocupar com a segurança. Nesse capítulo implementaremos as lógicas de autenticação e autorização utilizando uma biblioteca da Microsoft chamada **Simple Membership**.

## Configuração do SimpleMembership

Para utilizarmos o Simple Membership no projeto, precisamos primeiro instalá-lo utilizando o Nuget:

```
Install-Package Microsoft.AspNet.WebPages.WebData -Version 3.1.1
```

Agora que o plugin foi instalado, precisamos configurá-lo. Abra o arquivo `Web.Config` do projeto e procure a tag `system.web`. Dentro dessa tag, adicione a seguinte configuração:

```
<system.web>
  <!-- Outras configurações -->

  <membership defaultProvider="financasProvider">
    <providers>
      <add name="financasProvider"
           type="WebMatrix.WebData.SimpleMembershipProvider, WebMatrix.WebData"/>
    </providers>
  </membership>
</system.web>
```

Além disso, precisamos falar para o simple membership qual é a página que queremos utilizar para o login na aplicação, para isso, no arquivo de configuração, procure a tag `appSettings` e dentro dela adicione a seguinte configuração:

```
<appSettings>
  <!-- Outras configurações -->
  <add key="loginUrl" value="~/Login"/>
</appSettings>
```

Para o Simple Membership conseguir fazer a autenticação e autorização de usuários, ele precisa armazenar as informações de segurança dentro de tabelas próprias no banco de dados da aplicação, mas para isso ele precisa de uma conexão com o banco de dados da aplicação. Para configurar a conexão que será utilizada pelo simple membership, utilizamos o método `InitializeDatabaseConnection` da classe `WebSecurity`, que é a classe utilizada para fazer todas as interações com o Simple Membership.

No método `InitilizeDatabaseConnection` precisamos passar qual é o nome da connection string que queremos utilizar para nos conectarmos no banco de dados ( `FinancasContext` ), qual é o nome da tabela que guarda as informações do usuário ( `Usuarios` ), qual é o nome da coluna da tabela que representa o id do usuário ( `Id` ), qual é a coluna que quequeremos utilizar para guardar o login ( `Nome` ) e, por fim, se queremos que ele crie as tabelas auxiliares necessárias automaticamente no banco de dados:

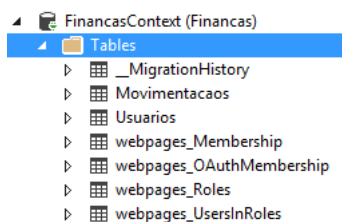
```
WebSecurity.InitializeDatabaseConnection("FinancasContext", "Usuarios", "Id", "Nome", true);
```

Essa linha só precisa ser executada uma vez na inicialização da aplicação, então podemos colocá-la no método `Application_Start` do arquivo `Global.asax` .

```
protected void Application_Start()
{
    // Outras configurações

    WebSecurity.InitializeDatabaseConnection("FinancasContext", "Usuarios", "Id", "Nome", true)
}
```

Agora quando inicializarmos a aplicação, o Simple Membership criará as seguintes tabelas no banco de dados:




## Modificando o cadastro de usuários

Agora que já terminamos a configuração inicial do simple membership, vamos modificar a lógica de cadastro de usuários para que ela também crie uma nova conta de login para o novo usuário que será cadastrado. Vamos inicialmente modificar o formulário de cadastro.

Dentro do formulário de cadastro, colocaremos dois novos campos: senha e confirmação de senha. Esses campos serão utilizados apenas para a conta de login do usuário, ou seja, eles serão armazenados pelo simple membership e não pela tabela de usuários da aplicação.

Porém ainda queremos montar o formulário de cadastro utilizando os métodos do `HtmlHelper` , então precisamos de uma view fortemente tipada para um modelo que tem as informações do usuário e os dois novos campos para o password. Esse modelo que é utilizado apenas como ponte entre a camada de visualização e a camada de negócio é conhecido como **View Model**:

```
public class UsuarioModel
{
    public string Nome { get; set; }

    public string Email { get; set; }

    public string Senha { get; set; }

    public string ConfirmacaoSenha { get; set; }
}
```

Agora para conseguirmos utilizar as validações do Asp.Net MVC, as anotações de validação precisam ficar na classe do view model:

```
public class UsuarioModel
{
    [Required]
    public string Nome { get; set; }

    [Required,EmailAddress]
    public string Email { get; set; }

    [Required]
    public string Senha { get; set; }

    public string ConfirmacaoSenha { get; set; }
}
```

Além das validações que já estavam no usuário, queremos adicionar uma nova validação que vai comparar o campo `Senha` com o campo `ConfirmacaoSenha`, para isso utilizaremos uma nova anotação chamada `CompareAttribute`:

```
public class UsuarioModel
{
    [Required]
    public string Nome { get; set; }

    [Required,EmailAddress]
    public string Email { get; set; }

    [Required]
    public string Senha { get; set; }

    [Compare("Senha")]
    public string ConfirmacaoSenha { get; set; }
}
```

Agora na action `Adiciona` do `UsuarioController`, precisamos receber o `UsuarioModel` ao invés do `Usuario`:

```
public ActionResult Adiciona(UsuarioModel model)
{
    if(ModelState.IsValid)
    {
        // grava o usuário no banco e cria
        // a conta de login com o simple membership
    }
    else
    {
        return View("Form", model);
    }
}
```

Dentro do bloco `if` do código acima precisamos primeiro gravar o usuário no banco de dados utilizando o Entity Framework e, em seguida, criar a conta de login do usuário utilizando o método `CreateAccount` da classe `WebSecurity`:

```
public ActionResult Adiciona(UsuarioModel model)
{
```

```
    if(ModelState.IsValid)
    {
        Usuario usuario = new Usuario();
        usuario.Nome = model.Nome;
        usuario.Email = model.Email;

        usuarioDAO.Adiciona(usuario);

        WebSecurity.CreateAccount(model.Nome, model.Senha);
        return RedirectToAction("Index");
    }
    else
    {
        return View("Form", model);
    }
}
```

Mas o método `CreateAccount` pode lançar uma exceção caso o nome do `Nome` do usuário seja repetido no banco de dados, logo se o usuário for repetido precisamos criar um novo erro de validação e voltar para o formulário de login:

```
public ActionResult Adiciona(UsuarioModel model)
{
    if(ModelState.IsValid)
    {
        try
        {
            Usuario usuario = new Usuario();
            usuario.Nome = model.Nome;
            usuario.Email = model.Email;

            usuarioDAO.Adiciona(usuario);

            WebSecurity.CreateAccount(model.Nome, model.Senha);
            return RedirectToAction("Index");
        }
        catch(MembershipCreateUserException e)
        {
            ModelState.AddModelError("usuario.Invalido", e.Message);
            return View("Form", model);
        }
    }
    else
    {
        return View("Form", model);
    }
}
```

Mas ainda temos um problema, pois o método `Adiciona` do `UsuarioDAO` já terminou de gravar o usuário no banco de dados. Só podemos gravar a informação do usuário no banco de dados caso a chamada para o método `CreateAccount` seja bem sucedida, mas só sabemos isso depois de gravar o usuário. Para resolver esse problema, a Microsoft criou um novo método na classe `WebSecurity` chamado `CreateUserAndAccount` .

No `CreateUserAndAccount` precisamos passar como argumentos o login e a senha que serão utilizados além de um objeto anônimo cujas informações serão adicionadas na tabela `Usuarios` do Entity Framework:

```
WebSecurity.CreateUserAndAccount(model.Nome, model.Senha, new {Email = model.Email});
```

Então a action `Adiciona` do `UsuarioController` fica com o seguinte código:

```
public ActionResult Adiciona(UsuarioModel model)
{
    if(ModelState.IsValid)
    {
        try
        {
            WebSecurity.CreateUserAndAccount(model.Nome, model.Senha,
                new { Email = model.Email });
            return RedirectToAction("Index");
        }
        catch(MembershipCreateUserException e)
        {
            ModelState.AddModelError("usuario.Invalido", e.Message);
            return View("Form", model);
        }
    }
    else
    {
        return View("Form", model);
    }
}
```

Para terminarmos, precisamos modificar o formulário de cadastro para que ele mostre a mensagem de validação do simple membership ( `usuario.Invalido` ), além de trocar o tipo da view para que ela utilize o View Model ao invés do `Usuario` diretamente:

```
@model Financas.Models.UsuarioModel

<h2>Form</h2>

@using(Html.BeginForm("Adiciona", "Usuario", FormMethod.Post))
{
    @Html.ValidationMessage("usuario.Invalido")

    @Html.LabelFor(u => u.Nome, "Nome:")
    @Html.TextBoxFor(u => u.Nome, new { @class="form-control"})
    @Html.ValidationMessageFor(u => u.Nome)

    @Html.LabelFor(u => u.Email, "E-mail:")
    @Html.TextBoxFor(u => u.Email, new {@class="form-control"})
    @Html.ValidationMessageFor(u => u.Email)

    @Html.LabelFor(u => u.Senha, "Senha:")
    @Html.PasswordFor(u => u.Senha, new { @class = "form-control" })
    @Html.ValidationMessageFor(u => u.Senha)

    @Html.LabelFor(u => u.ConfirmacaoSenha, "Confirme a senha:")
    @Html.PasswordFor(u => u.ConfirmacaoSenha, new { @class="form-control"})
    @Html.ValidationMessageFor(u => u.ConfirmacaoSenha)
```

```
    <input type="submit" value="Cadastrar"/>
}
```

Com essas modificações o cadastro de novo usuário com o simple membership está pronto!

## Autenticação de usuários

Com o cadastro de usuários finalizado, vamos implementar a página de login para a aplicação. Vamos, então, adicionar um novo controller na aplicação chamado `LoginController`. Dentro do método `Index` desse controller nós mostraremos o formulário de autenticação para o usuário.

```
public class LoginController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

Na view desse método `Index`, vamos colocar o código HTML do formulário:

```
<form action="@Url.Action("Autentica")" method="post">
    @Html.ValidationMessage("login.Invalido")

    <label for="login">Login:</label>
    <input id="login" type="text" name="login" class="form-control" />


    <label for="senha">Senha:</label>
    <input id="senha" type="password" name="senha" class="form-control" />


    <input type="submit" value="Autenticar" />
</form>
```

Agora no método `Autentica`, utilizaremos o método `Login` da classe `WebSecurity` passando o login e a senha do usuário. Esse método devolve o valor `true` caso o login seja bem sucedido e `false`, caso contrário:

```
public ActionResult Autentica(String login, String senha)
{
    if(WebSecurity.Login(login, senha))
    {
        return RedirectToAction("Index", "Movimentacao");
    }
    else
    {
        return View("Index");
    }
}
```

Para implementarmos a ação de logout, utilizamos o método `Logout` do `WebSecurity` :

```
public ActionResult Logout()
{
    WebSecurity.Logout();
    return RedirectToAction("Index");
}
```

Para finalizar o login na aplicação, vamos adicionar um novo link no menu superior ( `Views/Shared/_MenuSuperior.cshtml` ) da aplicação que levará o usuário para a página de login:

```
<nav>
    <div class="container-fluid">
        <ul class="nav nav-tabs">

            <!-- outros itens do menu -->

            <li>@Html.ActionLink("Autenticar", "Index", "Login")</li>
        </ul>
    </div>
</nav>
```

Mas se o usuário já estiver logado, queremos mostrar o link para a action de logout. Para sabermos se o usuário está logado, podemos utilizar a propriedade `IsAuthenticated` da classe `WebSecurity` .

```
if(WebSecurity.IsAuthenticated)
{
     // Usuário está logado
}
else
{
     // Usuário ainda não está logado
}
```

Mas na view do Asp.Net MVC, podemos também utilizar uma nova variável exposta pelo razor chamada `User` . Com a variável `User` , podemos descobrir se o usuário está logado com o seguinte código:

```
@User.Identity.IsAuthenticated
```

Então o código do menu pode ser modificado para:

```
<nav>
    <div class="container-fluid">
        <ul class="nav nav-tabs">

            <!-- outros itens do menu -->

            @if (User.Identity.IsAuthenticated)
            {
                <li>@Html.ActionLink("Deslogar", "Logout", "Login")</li>
```

```
            }
            else
            {
                <li>@Html.ActionLink("Autenticar", "Index", "Login")</li>
            }
        </ul>
    </div>
</nav>
```

## Autorização com o Simple Membership

Agora que já sabemos como fazer o login e logout de usuários, vamos utilizar o simple membership para fazer com que apenas usuários logados possam acessar o `MovimentacaoController` . Para isso utilizaremos um filtro definido no asp.net mvc chamado `AuthorizeAttribute` .

Para impedirmos o acesso ao `MovimentacaoController` para usuários deslogados, precisamos apenas colocar a anotação `AuthorizeAttribute` sobre a declaração da classe:

```
[Authorize]
public class MovimentacaoController : Controller
{
    // implementação da classe
}
```

Com essa modificação simples, se um usuário deslogado tentar entrar em qualquer lógica do `MovimentacaoController` ele será automaticamente redirecionado para a página de login que foi passado na configuração do simple membership.

Assim como os filtros que vimos no curso de Asp.Net MVC, o `AuthorizeAttribute` também pode ser colocado sobre uma determinada action de um controller ou também pode ser utilizado como um filtro global.

Está pronta a segurança do projeto utilizando a biblioteca Simple Membership da Microsoft!