**Filtro para autorização de usuários**

**Autorização através de filtros**

Agora temos uma lógica de autenticação na aplicação, porém o usuário ainda consegue acessar qualquer página mesmo não estando logado. Para resolvermos esse problema, na listagem de produtos, por exemplo, precisamos verificar se o usuário está logado antes de mostrar a lista:

public class ProdutoController : Controller

{

public ActionResult Index()

{

object usuarioLogado = Session["usuarioLogado"];

if(usuarioLogado != null)

{

// lógica para mostrar a lista

}

else

{

return RedirectToAction("Index", "Login");

}

}

}

Mas também precisamos proteger as outras actions do ProdutoController, então essa lógica de verificação do usuário logado terá que ser copiada para todas as actions do controller.

Para resolvermos esse problema, utilizaremos o filtro do Asp.Net MVC 5. Filtros são componentes que conseguem executar lógicas antes e depois do código do controller. Para criarmos um filtro, precisamos de uma classe que herda de ActionFilterAttribute do namespace System.Web.Mvc, então no projeto criaremos uma nova pasta chamada Filtros e dentro dessa pasta criaremos a classe AutorizacaoFilterAttribute:

public class AutorizacaoFilterAttribute : ActionFilterAttribute

{

}

Dentro de um filtro, quando queremos executar uma ação antes do código da action, precisamos sobrescrever o método OnActionExecuting e para executar uma ação depois do código da action, sobrescrevemos oOnActionExecuted. Em nosso caso, queremos verificar se o usuário está logado antes da lógica do controller, então sobrescreveremos o OnActionExecuting:

public override void OnActionExecuting(ActionExecutingContext filterContext)

{

}

Dentro desse método, recebemos um argumento do tipo ActionExecutingContext, dentro desse objeto podemos conseguir informações sobre a requisição que está sendo tratada atualmente.

No código do OnActionExecuting, se o usuário estiver logado, queremos deixar a lógica do controller executar normalmente, senão precisamos redirecionar o usuário para a página de login da aplicação. Então precisamos inicialmente acessar a sessão do servidor para buscar o usuário logado. Quando queremos acessar a sessão do servidor a partir do código do filtro, precisamos ler a propriedade HttpContext do filterContext e dentro doHttpContext, podemos acessar a Session do servidor:

public override void OnActionExecuting(ActionExecutingContext filterContext)

{

object usuarioLogado = filterContext.HttpContext.Session["usuarioLogado"];

}

Nos filtros, quando queremos deixar o usuário executar a lógica do controller, não precisamos fazer nada. Então precisamos tratar apenas o caso em que o usuário não está logado, nesse caso, não queremos permitir a execução do código do controller e para isso, precisamos escrever na propriedade Result do filterContext.

No caso em que o usuário não está logado, queremos escrever no Result um resultado fará um redirect no navegador do usuário, porém dentro do filtro não podemos utilizar o método RedirectToAction da mesma forma que fazíamos no controller, precisamos instanciar diretamente o objeto devolvido pelo RedirectToAction que é oRedirectToRouteResult.

Dentro do RedirectToRouteResult, precisamos passar o nome do controller e da action para onde queremos enviar o usuário. Essas informações são enviadas dentro de um objeto do tipo RouteValueDictionary e dentro desse dicionário precisamos passar as informações dentro de um objeto anônimo do C#:

filterContext.Result = new RedirectToRouteresult(

new RouteValueDictionary(

new { action="Index", controller="Login" }));

E todo esse código ficará dentro de um if que verifica se o usuário está logado:

public override void OnActionExecuting(ActionExecutingContext filterContext)

{

object usuarioLogado = filterContext.HttpContext.Session["usuarioLogado"];

if(usuarioLogado == null)

{

filterContext.Result = new RedirectToRouteResult(

new RouteValueDictionary(

new { action = "Index", controller = "Login" }));

}

}

Agora que o código do filtro está pronto, precisamos avisar ao Asp.Net MVC que ele será utilizado na action Indexdo ProdutoController, para isso precisamos anotar essa action com a classe do filtro que foi criado:

public class ProdutoController : Controller

{

[AutorizacaoFilterAttribute]

public ActionResult Index()

{

// código para a lista de produtos

}

}

Mas como foi dito anteriormente no curso, quando usamos uma classe como anotação não precisamos colocar o sufixo Attribute no nome da anotação:

public class ProdutoController : Controller

{

[AutorizacaoFilter]

public ActionResult Index()

{

// código para a lista de produtos

}

}

Mas se quisermos proteger todas as actions desse controller, teríamos que colocar a anotação em todos os métodos, ao invés disso, podemos colocar a anotação sobre a declaração da classe ProdutoController:

[AutorizacaoFilter]

public class ProdutoController : Controller

{

public ActionResult Index()

{

// código para a lista de produtos

}

}

Quando o filtro é colocado sobre um controller, ele será executado antes e depois de cada uma das actions desse controller.

Também podemos falar para o Asp.Net MVC que um determinado filtro precisa ser executado em toda requisição que chega na aplicação. Quando queremos registrar um filtro global na aplicação, precisamos abrir um arquivo chamado Global.asax do projeto, é nele que fazemos as configurações programáticas globais de uma aplicação Asp.Net MVC.

Dentro do método Application\_Start do Global.asax, precisamos adicionar a instância do filtro dentro da coleção de filtros globais da aplicação o GlobalFilters.Filters.

protected void Application\_Start()

{

GlobalFilters.Filters.Add(new AutorizacaoFilterAttribute());

// resto do código continua igual

}

Veja que se colocarmos muitas configurações globais dentro do Global.asax, o arquivo acaba ficando grande e bagunçado, para resolver esse problema, geralmente colocamos as configurações dentro de classes separadas e depois simplesmente chamamos a classe de configuração dentro do Global.asax. A classe que é normalmente utilizada para registrar filtros globais é uma classe chamada FilterConfig que é usualmente criada dentro da pastaApp\_Start da aplicação.

Dentro dessa classe, os filtros serão registrados dentro de um método chamado RegisterGlobalFilters que recebe um argumento do tipo GlobalFilterCollection, o mesmo tipo do GlobalFilters.Filters:

public class FilterConfig

{

public static void RegisterGlobalFilters(GlobalFilterCollection filters)

{

filters.Add(new AutorizacaoFilterAttribute());

}

}

E agora podemos chamar o RegisterGlobalFilters dentro do Application\_Start:

protected void Application\_Start()

{

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

// resto do código

}

Agora o filtro será executado para toda requisição que chega ao servidor.

**Cross Site Request Forgery**

Até agora, nossas actions tratam todas as requisições feitas para o servidor, porém imagine que sua aplicação tenha uma funcionalidade que permite recuperar senhas perdidas por e-mail:

public ActionResult RecuperaSenha(String email)

{

UsuarioDao dao = new UsuarioDao();

Usuario usuario = dao.BuscaPorEmail(email);

GeraNovaSenha(usuario);

dao.Atualiza(usuario);

EnviaNovaSenhaParaOEmailDoUsuario(usuario);

return View();

}

E um formulario que atualiza o e-mail atual do usuário logado:

<form action="seu\_site/Usuario/AtualizaEmail" method="post">

<input name="novoEmail" />

<input type="submit" />

</form>

Esse formulário será enviado para o método AtualizaEmail do UsuarioController, que simplesmente muda o e-mail cadastrado do usuário logado.

Agora suponha que um usuário se autentique no seu site e esqueça de fazer o logout. Nesse caso, para o servidor da aplicação, o usuário continuará logado. Agora esse usuário entra em um site malicioso que envia automaticamente o seguinte formulário html:

<form action="seu\_site/Usuario/AtualizaEmail" method="post">

<input name="novoEmail" value="email.hacker@hackers.com"/>

</form>

Como o usuário continua logado em nossa aplicação, a requisição será processada normalmente pelo servidor, que irá trocar o e-mail cadastrado do usuário para email.hacker@hackers.com e, utilizando a funcionalidade de recuperação de senhas, o hacker poderá tomar o controle da conta do usuário.

O problema ilustrado acima é conhecido como cross site request forgery (csrf). A solução dada pelo ASP.NET MVC é colocar um token (uma chave secreta que só o browser e o servidor conheçam naquele momento) entre os dados enviados pelo formulário. Esse token é conhecido como csrf token e pode ser gerado pelo métodoAntiForgeryToken do HtmlHelper. Vamos modificar nosso formulário de cadastro de produtos para utilizar o csrf token.

<form action="/Produto/Adiciona">

@Html.AntiForgeryToken()

<label>Nome: <input name="produto.Nome" /></label>

@Html.ValidationMessage("produto.Nome")

<label>Preco: <input name="produto.Preco" /></label>

@Html.ValidationMessage("produto.Preco")

<label>Quantidade: <input name="produto.Quantidade" /></label>

@Html.ValidationMessage("produto.Quantidade")

<label>Descricao: <input name="produto.Descricao" /></label>

@Html.ValidationMessage("produto.Descricao")

<label>

Categoria:

<select name="produto.Categoria.Id">

@foreach(var categoria in Model) {

<option value="@categoria.Id">@categoria.Nome</option>

}

</select>

@Html.ValidationMessage("produto.Categoria")

</label>

<input type="submit" />

</form>

Porém, apenas adicionar o token ao formulário não resolve o problema. Devemos agora fazer com que o servidor verifique se o token enviado é válido. Para isso utilizaremos um filtro definido pelo ASP.NET MVC, oValidateAntiForgeryTokenAttribute. Para utilizá-lo, devemos apenas anotar nossa action. Vamos modificar o método Adiciona do ProdutoController para verificar o token:

class ProdutoController : Controller

{

// outras actions do controller

[ValidateAntiForgeryToken]

public ActionResult Adiciona(Produto produto)

{

// implementação do adiciona

}

}

Agora apenas requisições que enviem um token com valor correto serão tratadas pela action.