Desenvolvimento - ASP. NET

Autenticação e Autorização no ASP.NET MVC

Com a vinda do ASP.NET MVC, grande parte dos serviços que são disponibilizados para o ASP.NET WebForms, já nasceram com o MVC ou estão aos poucos sendo adicionados à plataforma.

por *Israel Aéce*

Desde a primeira versão do ASP.NET, temos três formas de autenticação: *Windows*, *Forms* e *Passport*. A primeira opção consiste em permitir o acesso desde queo usuário faça parte do domínio/máquina. Essa opção torna o gerenciamento simples, facilita o *Single Sign-On* (SSO) interno, mas é útil em um ambiente controlado, como uma *intranet*. A opção *Forms* faz com que a aplicação se encarregue de gerenciar e autenticar seus usuários, baseando-se em um *cookie* que determinará quem é o usuário logado, e esta opção é a mais ideal para a *internet*. Finalmente, temos a *Passport*, que permitia autenticar o usuário utilizando o serviço de identidade da *Microsoft*, mas por haver um custo envolvido, acabou não evoluindo.

Com a vinda do *ASP.NET MVC*, grande parte dos serviços que são disponibilizados para o *ASP.NET WebForms*, já nasceram com o MVC ou estão aos poucos sendo adicionados à plataforma. Como já era de se esperar, a questão da autenticação e autorização já fazem parte desde a versão 1.0 do MVC, onde podemos utilizar a autenticação *Windows* ou *Forms*, mudando ligeiramente como devemos proceder para configurá-los e utilizá-los.

Para definir se vamos utilizar o modelo de autenticação *Windows* ou *Forms*, devemos recorrer ao elemento *authentication* no arquivo *Web.config*, como já fazíamos anteriormente. O atributo *mode* determina qual a opção de autenticação, e um sub-elemento chamado *forms*, nos permite customizar a autenticação baseada em *Forms*. O exemplo abaixo ilustra a configuração inicial:

```
<authentication mode="Forms">
<forms loginUrl="Authentication/LogOn" />
</authentication>
```

Aqui há uma mudança radical aqui. O *ASP.NET WebForms* é baseado em arquivos, ou seja, todas as requisições, sejam elas da aplicação ou da infraestrutura (como é o caso aqui), o alvo sempre será um arquivo físico, com extensão **.aspx*. Já no *ASP.NET MVC*, o alvo será sempre uma ação, que nada mais é que um método dentro de uma classe, conhecida como *Controller*. No exemplo acima, o *controller* é uma classe chamada *AuthenticationController* e dentro dele há um método chamado *LogOn*. É dentro deste método que você colocará todos os passos necessários para autenticar o usuário. Geralmente esse método deve receber o *login* e a senha, mas eventualmente poderá ter algo mais rebuscado.

Quando tentamos acessar um recurso protegido através do *FormsAuthentication*, ele automaticamente redireciona o usuário para a página deautenticação configurada acima, embutindo na URL uma *query string* chamada *ReturnUrl*, com o caminho (do arquivo ou da ação) que tentamos acessar. Esse parâmetro é útil para depois que validar o usuário, redirecioná-lo para a mesma seção que ele solicitou antes de efetuar a autenticação. Sendo assim, é necessário que o método que efetuaa autenticaçãodo usuário, também receba esse parâmetro.

Em tempo,a versão 2.0 do *ASP.NET MVC* fornece um atributo chamado *HttpPostAttribute*, que permite você decorar uma determinada ação, para que ela seja executada somente via POST, garantindo assim que ela seja executada somente através desta forma. Para ilustrar, o método *LogOn* fica da seguinte forma:

```
public class AuthenticationController : Controller
{
```

```
[HttpPost]
public ActionResult LogOn(string userName, string password, string returnUrl)
{
//...
}
}
```

É importante dizer que se você estiver utilizando a autenticação *Windows*, nada disso é necessário, já que a autenticação acaba sendo feita automaticamente pelo IIS/ASP.NET.

Se precisar saber qual usuário está autenticado naquele momento, você pode recorrer à propriedade *User*, exposta pela classe *Controller*. Assim como já acontece com a classe *Page* do *WebForms*, essa propriedade retorna a instância de uma classe que implementa a *interface IPrincipal*, fornecendo acesso ao *username* do usuário atual, e um método chamado *IsInRole*, caso você precisa refinar ainda mais a autorização. Para maiores detalhes sobre essa *interface*, consulte este o capítulo9 deste artigo.

Autorização

A autorização pode continuar sendo realizada da mesma forma que fazíamos antes, ou seja, recorrendo ao arquivo de configuração (*Web.config*) para especificar as regras de autorização. A diferença se dá também pelo fato de que o MVC é baseado em ações, e ao invés de determinar páginas e/ou diretórios nessas regras, utilizaremos as ações para refinar o acesso dos usuários. Se temos um *controller* chamado *MonitorController* e dentro dele uma ação (método) chamada *VisualizarItens*, e o acesso à ela somente pudesse ser feita por usuários autenticados, a regra ficaria da seguinte forma:

```
<location path="Monitor/VisualizarItens">
<system.web>
<authorization>
<deny users="?" />
</authorization>
</system.web>
</location>
```

Caso você precise permitir o acesso à uma determinada ação para somente aqueles usuários que fazem parte de um determinado grupo (*role*), você pode configurar a regra da seguinte forma:

```
<location path="Monitor/VisualizarItens">
<system.web>
<authorization>
<allow roles="Financeiro" />
</authorization>
</system.web>
</location>
```

Apesar desta técnica funcionar, é difícil de manter. Para cada novo *controller*ou ação que são criados, você precisa se preocupar em olhar para o arquivo de configuração e configurar quem poderá ou não ter acesso. O problema desta técnica é que se você esquecer, usuários que talvez não deveriam ter acesso, acabarão visualizando. No *ASP.NET MVC* há uma outra forma de se trabalhar, também declarativa, mas recorrendo à atributos dentro dos *controllers*/ações, facilitando a centralização e manutenção das regras de acesso.

Para isso, o *ASP.NET MVC* fornece um atributo chamado *AuthorizeAttribute* que podemos decorar em classes (*controllers*) ou em métodos (ações). Esse atributo fornece duas propriedades do tipo *string*: *Users* e *Roles*. Na primeira propriedade, especificamos os usuários (separados por vírgula) que podem ter acesso aquele membro (classe ou método) em que o atributo está sendo aplicado. Já a segunda propriedade, *Roles*, permite especificarmos os grupos (separados por vírgula), onde somente usuários que estão dentro de um deles é que podem ter acesso ao recurso. Utilizar a propriedade *Users* não é o mais ideal, porque usuários são muito "voláteis", e em pouco tempo, essa regra provavelmente não fará mais sentido.

Como dito acima, podemos aplicar esse atributo em nível de *controller*, e assim, as suas respectivas ações irão exigir que aquela regra seja atendida antes de permitir o acesso. No primeiro exemplo, aplicamos este atributo em nível de *controller*, e todas as ações exigirão que o usuário esteja autenticado:

```
[Authorize]
public class MonitorController : Controller
{
public ActionResult VisualizarItens()
{
//...
}

public ActionResult RecuperarIndice()
{
//...
}
}
```

Já o segundo exemplo, fará com que o *controller* seja acessado somente por usuários devidamente autenticados, mas isso não é o suficiente para o método *VisualizarItens*, que além disso, exigeque o usuário esteja contido no grupo *Financeiro*:

```
[Authorize]
public class MonitorController : Controller
{
[Authorize(Roles = "Financeiro")]
public ActionResult VisualizarItens()
{
//...
}

public ActionResult RecuperarIndice()
{
//...
}
}
```

Caso você tenha uma regra de validação muito mais complexa do que isso, nada impede você de criar um filtro para efetuar essa customização, fazendo que a autorização siga os passos necessários para atender essa regra predefinida. E para finalizar, lembre-se que o MVC trata os métodos como *"opt-out"*, ou seja, por padrão, todos os métodos públicos estão acessíveis. Se mesmo com a autenticação e/ou autorização ele jamais deverá ser invocado, então você deve decorá-lo com o atributo *NonActionAttribute*.

Membership e Roles

A partir da versão 2.0 do ASP.NET, uma série de novos serviços foram adicionados, e entre eles temos o *Memberhip Provider* e *Role Provider*, algo que já falei extensivamente nestes outros artigos. Felizmente podemos continuar utilizando esses serviços, mas com um pouco mais de cautela.

Esses serviços seguem um padrão conhecido como *Provider Model*, e ao utilizar o *Membership*ou *Roles*, eu estou trabalhando de forma independente de repositório; posso trabalhar com *SQL Server*, *Oracle* ou *Xml*, apenas configurando os *providers* através do arquivo *Web.config*. Mas apesar desses serviços seguirem um padrão de extensibilidade bem definido, isso não quer dizer que é uma boa prática você pode utilizá-los diretamente dentro dos *controllers*.

É importante dizer que ao criar aplicações MVC, a testabilidade deve estar em foco, e utilizar as classes *Membership* ou *Roles* dentro dos *controllers*, o tornará dependente destas implementações, ou melhor, destes serviços, e que por sua vez, estão fortemente acoplados ao ASP.NET. Como disse anteriormente, a arquitetura do *provider model* é bem flexível, mas para criar uma versão*"fake"* é complexo demais, pois há uma infinidade de funcinalidades que as vezes serão desnecessárias. Além disso, utilizar essas classes torna meu *controller* dependente deste tipo de serviço mas, eventualmente, eu tenho o meu próprio repositório de usuários e suas respectivas permissões, e quero fazer uso deles ao invés daqueles que o ASP.NET fornece.

A classe AccountController

Quando criamos um novo projeto MVC, por padrão, uma série de recursos já são adicionados, e entre eles, temos a classe (*controller*) chamada *AccountController*. Essa classe foi desenhada para servir como um *wrapper* para os serviços de *Membership* e *FormsAuthentication*. Há vários problemas com essa classe, e o primeiro deles é aquele que descrevi acima, que é a forte dependência da API do *Membership*. Apesar da *Microsoft* ter refatorado o código e criado uma *interface* chamada *IMembershipService*, há ainda alguns tipos que estão sendo utilizados, como é o caso do enumerador *MembershipCreateStatus*. Além disso, essa classe ainda viola o princípio de SRP (*Single Responsability Principle*), que faz com que ela faça muito mais coisas do que realmente deveria.

Conclusão: O fato da *Microsoft* tem se preocupado em manter grande parte dos recursos do *WebForms* no MVC, é importante que você analise cuidadosamente como incorporá-lo ao MVC para não comprometer um dos maiores benefícios do MVC, que são os testes. É importante notar que grande parte do conhecimento adquirido no *ASP.NET WebForms*, acaba sendo reutilizado aqui, apenas com ligeiras mudanças devido à arquitetura do MVC.

Israel Aéce - Especialista em tecnologias de desenvolvimento Microsoft, atua como desenvolvedor de aplicações para o mercado financeiro utilizando a plataforma .NET. Como instrutor Microsoft, leciona sobre o desenvolvimento de aplicações .NET. É palestrante em diversos eventos Microsoft no Brasil e autor de diversos artigos que podem ser lidos a partir de seu site http://www.israelaece.com/. Possui as seguintes credenciais: MVP (Connected System Developer), MCP, MCAD, MCTS (Web, Windows, Distributed, ASP.NET 3.5, ADO.NET 3.5, Windows Forms 3.5 e WCF), MCPD (Web, Windows, Enterprise, ASP.NET 3.5 e Windows 3.5) e MCT.

Leia também

Qualidade de Software: dicas para escrever um código de qualidade
ASP. NET

Consultar conteúdo de listas do SharePoint por Web Service
ASP. NET

Utilizando ListView para fazer um CRUD
ASP. NET

Criando RSS XML para o Seu Próprio Site
C#

Popup em destaque no site em C# e ASP.Net
C#