# Writing Your Own Custom ASP.NET MVC [Authorize] Attribute

24 July 2011

ASP.NET's [Authorize] attribute is another cool feature that makes it easy to add authentication at the Controller level when building a website, but the real goldmine here is that like nearly everything else in ASP.NET MVC, you can pick apart the functionality and extend it yourself – In this post we will take a look at creating our own custom Authentication attribute.

```
[ASPNetMVCIsAwesome]
public ActionResult WhatYouWantToKnow()
{
    return View();
}
```

Lately I have been involved in a number of projects that have used ASP.NET MVC as the primary web service platform because of its awesome REST based approach to controllers – WCF fans out there may be shocked to hear this, but i truly believe that building as website is a pretty solution agnostic task, and ASP.NET MVC allows us to get a lot closer to how the rest of the world thinks about "web services" in the sense that your service really is *just another end point of your website* just one that returns JSON or XML and therefore doesn't really need the differentiation of being a "separate thing/entity" to your website (i.e. Web site/Web service).

ASP.NET MVC has another really neat feature tucked up its sleave when it comes to security in the form of the [Authenticate] attribute (if you have never heard of this head on over to ASP.NET to take a look at a quick tutorial). The ability to have granular security baked-in at the class or method level in a controller really appeals to me, so ASP.NET MVC's [Authenticate] attribute/annotation ticks all the right boxes by allowing you to easily hook into the ASP.NET membership provider.

The real question is; What about when you want to take an even more granular approach to the membership/[Authorize] implementation? I have seen a few developers who when facing this task start hacking the extra security into their Controller method's code – this is not where it should be done. The primary reason for this is the fact that if you are using ASP.NET's caching in your controllers, the ActionResult will be cached and your controller code will not even be hit!

## What ASP.NET MVC Controller Action authentication looks like

The following code snippet is from the standard ASP.NET MVC project's AccountController showing the **[Authorize]** attribute being used to enforce security in the user account section of the website:

```
[Authorize]
public ActionResult ChangePassword(ChangePasswordModel model)
{
    ...
}
```

The above annotation is used to let the ASP.NET MVC framework know you want to check a user is authenticated before processing the controller action – this authentication check is done even if the result of the controller action is cached (as i mentioned above this is a really important thing to point out, as if you placed your authentication inside your controller and wanted to perform caching on your output you would have to build your own caching).

## Anatomy of an [Authorize] attribute

When you place the [Authorize] attribute on a Controller's action method, a couple of calls get made to the AuthorizeAttribute class at the beginning of each request to your controller to authenticate users. There is a lot of code in the MVC framework's AuthorizeAttribute class, but the lines containing the methods we really care about are the following (as pulled from the Microsoft AuthorizeAttribute source using .NET Reflector):

```
public virtual void OnAuthorization(AuthorizationContext filterContext)
{
    if (filterContext == null)
    {
        throw new ArgumentNullException("filterContext");
    }
```

```
    if (AuthorizeCore(filterContext.HttpContext))
    {
        // Since we're performing authorization at the action level, the authorization code runs
        // after the output caching module. In the worst case this could allow an authorized user
        // to cause the page to be cached, then an unauthorized user would later be served the
        // cached page. We work around this by telling proxies not to cache the sensitive page,
        // then we hook our custom authorization code into the caching mechanism so that we have
        // the final say on whether a page should be served from the cache.
        HttpCachePolicyBase cachePolicy = filterContext.HttpContext.Response.Cache;
        cachePolicy.SetProxyMaxAge(new TimeSpan(0));
        cachePolicy.AddValidationCallback(CacheValidateHandler, null /* data */);
    }
    else
    {
        HandleUnauthorizedRequest(filterContext);
    }
}
// This method must be thread-safe since it is called by the thread-safe OnCacheAuthorization()
method.
protected virtual bool AuthorizeCore(HttpContextBase httpContext)
{
    if (httpContext == null)
    {
        throw new ArgumentNullException("httpContext");
    }
```

```
        IPrincipal user = httpContext.User;
        if (!user.Identity.IsAuthenticated)
        {
            return false;
        }
        if (_usersSplit.Length <= 0 || _rolesSplit.Length <= 0)
        {
            return false;
        }
        if (!_usersSplit.Contains(user.Identity.Name, StringComparer.OrdinalIgnoreCase) ||
            !_rolesSplit.Any(user.IsInRole))
        {
            return false;
        }
        return true;
    }
```

The great thing is the AuthorizeAttribute class shown above can be inherited from and the methods above can be overridden to allow us to disable the authentication checks for the example I have for you below. *These same methods can be used to add special authentication checks or anything else you desire*.

## An Example: A Switchable Authorize Attribute

When creating web services with ASP.NET MVC I have often been in situations where there has been a need to turn off authentication during development for access by Silverlight/Flash/JavaScript requests on the client side is pretty much a necessity to avoid drawing out the development/wasting time from liaison with other developers un-necessarily.

The example I'm going to take you through solves this in the form of a switchable [Authorization] attribute that will allow you to turn off authentication when the solution is built in DEBUG mode (i.e. during development). This will allow developers working with the site during development to test web service endpoints without having to log in every time they want to test. You could change this to run from an AppSetting property but i recommend against this as it is a pretty big vulnerability if either it doesn't get set by developers at deployment time, or someone with nefarious intentions changes it once the site is live.

Custom Attribute class code:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false, Inherited = true)]
public class SwitchableAuthorizationAttribute : AuthorizeAttribute
{
    protected override bool AuthorizeCore(HttpContextBase httpContext)
    {
        bool disableAuthentication = false;
        #if DEBUG
        disableAuthentication = true;
        #endif
        if (disableAuthentication)
            return true;
        return base.AuthorizeCore(httpContext);
    }
}
```

Usage:

```
[SwitchableAuthorization]
public ActionResult ChangePassword()
{
    return View();
}
```

As you can see from the above example's usage of the overridden *AuthorizeCore()* method, when built using Debug mode, the method returns true without asking the membership provider. This will allow Silverlight and Flash developers to test the site's code from within their IDE when developing.

## Adding other Attribute awareness

An even more powerful addition to the above example is the ability of the *OnAuthorization()* method to see the other attributes applied to the class/method at run time – allowing your attribute to change its functionality based on the other attributes applied. In my example I check that the [HttpPost] attribute is applied to the method as well by checking the count of the attribute.

Example:

```
public override void OnAuthorization(AuthorizationContext filterContext)
{
    //Check that the HttpPost attribute is also applied
    ActionDescriptor action = filterContext.ActionDescriptor;
    bool isAnHttpPost = action.GetCustomAttributes(typeof(HttpPostAttribute), true).Count() > 0;
    base.OnAuthorization(filterContext);
}
```

## Where to from here?

If you take anything from the above post, it has to be yet again how awesome and powerful the ASP.NET MVC framework is. The next time you are looking to add functionality to your ASP.NET MVC site in an elegant manner, think about creating a custom attribute.