**Custom authentication filter in ASP.NET MVC**

In this article, we shall learn how to create custom authentication filter in ASP.NET MVC 5+ using built in Authentication filter.

**Introduction**

AuthenticationFilter has been introduced newly in ASP.NET MVC 5+ to provide fine-grain control over how users are authenticated for controllers and action methods.

In this example, we shall learn how to create a custom authentication in ASP.NET MVC 5+.

**Creating a custom Authentication Filter**

**Problem Scenario**

Assume a scenario, where we want our Controller or an action methods available only to user having role as *Admin*or *SuperAdmin*.

Normally, we would write following attribute to the controller

    [Authorize(Roles = "Admin, SuperAdmin")]  
    public class AdminSuperAdminController : Controller  
    {     
  
    }

In general, there is no problem in hard coding the Authorize attribute for role as Admin and SuperAdmin, however think about the scenario, where we have to filter many controllers or action methods in the project. In general, what would we do is to copy-paste the attribute everywhere.

This raises a concern of duplication and maintenance. Tomorrow, if we need to change role names to something else or need to add another role into it then we will have to find and replace from entire project. There are chances or errors in this plus a lot of work too.

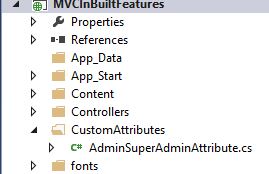
There can be many other scenarios where custom authentication filter is required and in all those scenario, this approach can be followed.

**Solution**

The solution of this problem is to create a custom authentication filter and use that in the controller or action methods. If we need any change, we just need to modify the custom authentication filter code at one place and that will affect all controller and action methods that is using this custom filter.

**How to create a custom Authentication Filter?**

To create a custom authentication filter, add a folder into the project called "CustomAttributes" (the folder name can be anything but better to keep it meaningful) and add a file called AdminSuperAdminAttribute.cs.



Now add following lines of code into AdminSuperAdminAttribute.cs file.

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Security.Principal;  
using System.Web;  
using System.Web.Mvc;  
using System.Web.Mvc.Filters;  
  
namespace MVCInBuiltFeatures.CustomAttributes  
{  
    public class AdminSuperAdminAttribute : FilterAttribute, IAuthenticationFilter  
    {  
        string superAdminRole = "SuperAdmin"; // can be taken from resource file or config file  
        string adminRole = "Admin"; // can be taken from resource file or config file  
  
        public void OnAuthentication(AuthenticationContext context)  
        {  
            if (context.HttpContext.User.Identity.IsAuthenticated &&   
               (context.HttpContext.User.IsInRole(superAdminRole)   
                || context.HttpContext.User.IsInRole(adminRole)))  
            {  
                // do nothing  
            }  
            else  
            {  
                context.Result = new HttpUnauthorizedResult(); // mark unauthorized  
            }  
        }  
  
        public void OnAuthenticationChallenge(AuthenticationChallengeContext context)  
        {  
            if (context.Result == null || context.Result is HttpUnauthorizedResult)  
            {  
                context.Result = new RedirectToRouteResult("Default",  
                    new System.Web.Routing.RouteValueDictionary{  
                        {"controller", "Account"},  
                        {"action", "Login"},  
                        {"returnUrl", context.HttpContext.Request.RawUrl}  
                    });  
            }  
        }  
    }  
}

In the above code snippet, we are doing following:

First, we have inherited this class file with FilterAttribute and IAuthenitcaitonFilter interface. For this, we need to add couple of namespaces as shown in the above code snippet.

Next, we have taken two variables and hard coded the role names. In real time scenario, we might not hard code  role name but can get from resource file, web.config file or from the database so that if role name changes, we do not need to change the code, rebuild and deploy the project. We can just change the source (Resource file, web.config or database) and we are done.

Interface IAuthenitcaitonFilter force us to implement a method named **OnAuthentication** and **OnAuthenticationChallenge** event only executes when OnAuthentication method has set the result**.**

In above code snippet, OnAuthenticationChallenge method will only executes when a request goes into the else block on OnAuthentication method and it redirects the user to Login page with returnUrl as querystring.

In OnAuthentication method, we have checked following

* if user is authenticated and
* if user role is SuperAdmin or
* if user role is Admin

then do nothing else set the context.Result to HttpUnathorizedResult (unauthorized). The same is is being checked inside OnAuthenticationChallenge method and if it is null or HttpUnauthorizedResult then user is being redirected to the Login page of the website with current url as returnUrl querystring.

Once our Custom authentication filter is ready, we need to hook it up into the controller or action method as per our requirement; so lets do it.

To hook it up the controller, do something like below

    [AdminSuperAdmin]  
    //[Authorize(Roles = "Admin, SuperAdmin")]  
    public class AdminSuperAdminController : Controller  
    {   
  
    }

and for controller action method, do something like below

        //  
        // GET: /AdminSuperAdmin/Create  
        [AdminSuperAdmin]  
        public ActionResult Create()  
        {  
            return View();  
        }

Remember that when we apply the attribute, we skip the "attribute" word suffix to the custom attribute class name.