**Authentication and Authorization in ASP.NET MVC**

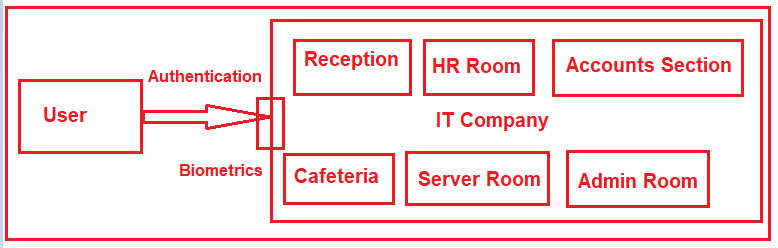
**Authentication and Authorization in ASP.NET MVC**

In this article, I am going to discuss **Authentication and Authorization in the ASP.NET MVC** application. When you are developing any web application, then the most important thing that you need to take care of is its security. That means we need to make sure that only authenticated and authorized users can access our webpage. As part of this article, we are going to discuss the following things.

1. **What are Authentication and Authorization?**
2. **What are the different types of Authentication?**
3. **How to implement Authentication and Authorization in ASP.NET MVC application?**

**What is Authentication?**

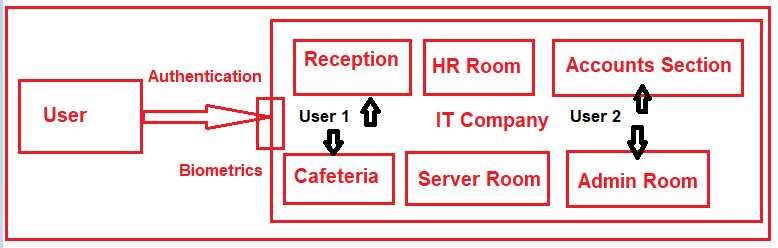
Authentication is nothing but a process that ensures and confirms a user’s identity. In other words, we can say that it is a process to validate someone against some data source. Let’s have a look at the following diagram.



Let us understand Authentication from a layman’s point of view. The above image shows the different sections of an IT Company like Reception, HR Section, Accounts Section, Server Room, etc. At the gate, we have biometrics to verify the employee. Suppose one user or one employee comes. This biometrics checks the employee credentials against some data source and if it is found the employee is a valid employee then it only allows entering into the campus. This is nothing but Authentication.

**What is Authorization?**

Authorization is a security mechanism that is used to determine whether the user has access to a particular resource or not. The most important point that you need to remember is, authentication happens first, and then only authorization. Let us have a look at the following image.



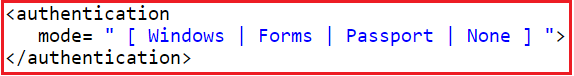
As shown in the above image, once the user is authenticated then he enters the Campus. Then Authorization comes into the picture. Within the campus in which section he may allow entering is determined by the Authorization process. This is basically done by the Role of the user. If the user is having list privileges then he may not allow each and every section. On the other hand, if the user is having the highest privileges then he may allow entering each and every section.

**Types of Authentication:**

The different types of Authentication supported by ASP.NET MVC are as follows:

1. **Forms Authentication:** In this type of authentication the user needs to provide his credentials through a form.
2. **Windows Authentication:** Windows Authentication is used in conjunction with IIS authentication. The Authentication is performed by IIS in one of three ways such as basic, digest, or Integrated Windows Authentication. When IIS authentication is completed, then ASP.NET uses the authenticated identity to authorize access
3. **Passport Authentication:** It is a centralized authentication service (paid service) provided by Microsoft which offers a single logon and core profile services for member sites.
4. **None:** No Authentication provided. This is the default Authentication mode

In the **web.config** file of your application, you can specify the Authentication mode as shown below.



**Different ways to implement Authentication in MVC:**

There are many different ways to implement Authentication in ASP.NET MVC. Here in this article series, we are going to use the following two ways to implement Authentication and Authorization in the MVC applications.

1. **FormsAuthentication**
2. **ASP.NET Identity**

**Forms Authentication in ASP.NET MVC**

1. **How to Sign up a new user into our application?**
2. **Implementing the User Login page.**
3. **How to use the built-in Authorize Attribute?**
4. **Implementing the logout functionality?**
5. **How to use Forms Authentication in ASP.NET MVC application to achieve all the above points?**

Let us start the implementation.

**SQL Script:**

Please use the below SQL Script to create the necessary data and its related tables.

-- Creating a database

**CREATE** **DATABASE** MVC\_DB

**GO**

**USE** MVC\_DB

**GO**

-- Creating Employee table

**CREATE** **TABLE** Employee

(

**ID** **INT** **PRIMARY** **KEY** **IDENTITY**(1,1),

Name **VARCHAR**(50),

Designation **VARCHAR**(50),

Salary **INT**

)

-- Creating Users table

**CREATE** **TABLE** Users

(

**ID** **INT** **PRIMARY** **KEY** **IDENTITY**(1,1),

UserName **VARCHAR**(50),

UserPassword **VARCHAR**(50)

)

-- Creating RoleMaster Table

**CREATE** **TABLE** RoleMaster

(

**ID** **INT** **PRIMARY** **KEY** **IDENTITY**(1,1),

RollName **VARCHAR**(50)

)

-- Creating User Roles Mapping table

**CREATE** **TABLE** UserRolesMapping

(

**ID** **INT** **PRIMARY** **KEY**,

UserID **INT** NOT **NULL**,

RoleID **INT** NOT **NULL**,

)

-- Adding Foreign KeyS

**ALTER** **TABLE** UserRolesMapping

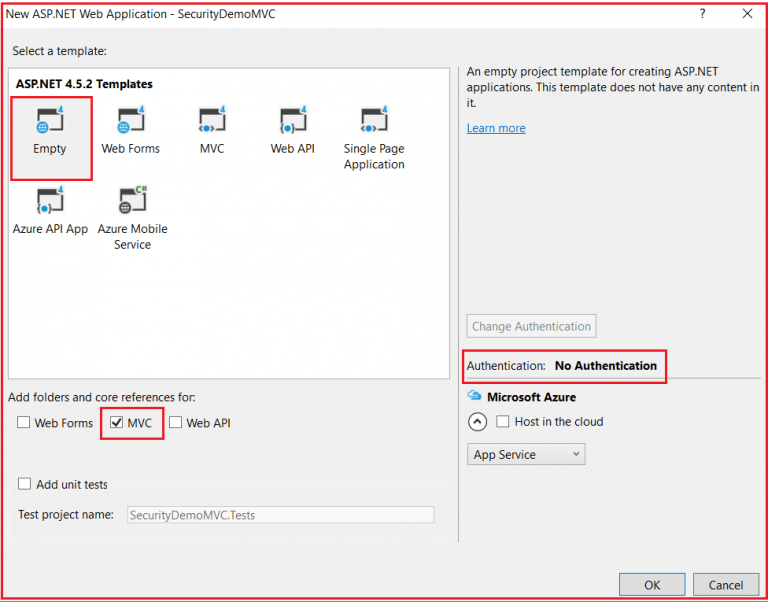
**ADD** **FOREIGN** **KEY** (UserID) **REFERENCES** Users(**ID**);

**ALTER** **TABLE** UserRolesMapping

**ADD** **FOREIGN** **KEY** (RoleID) **REFERENCES** RoleMaster(**ID**);

**Creating a new Empty ASP.NET MVC Application:**

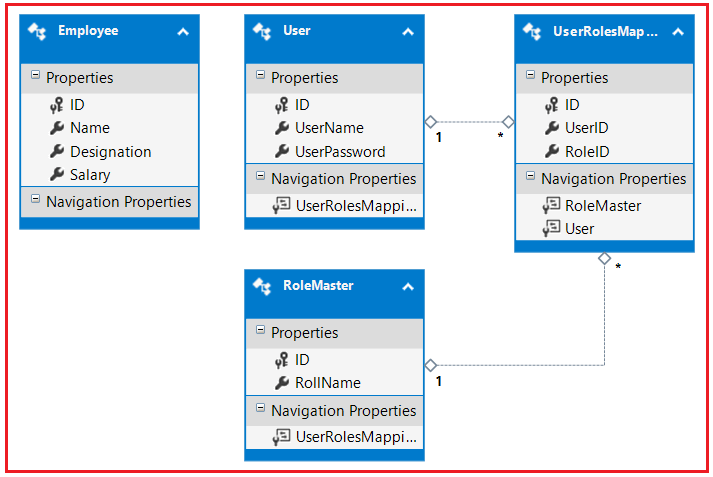
Create a new Empty ASP.NET MVC application with the name **SecurityDemoMVC**. Please select the authentication type as no authentication as shown in the below image.



Once you click on the **OK** button then it will create the project. Here we select the **Empty MVC Project template** as we are going to do everything from scratch. So that it will be easy for you to understand the concept.

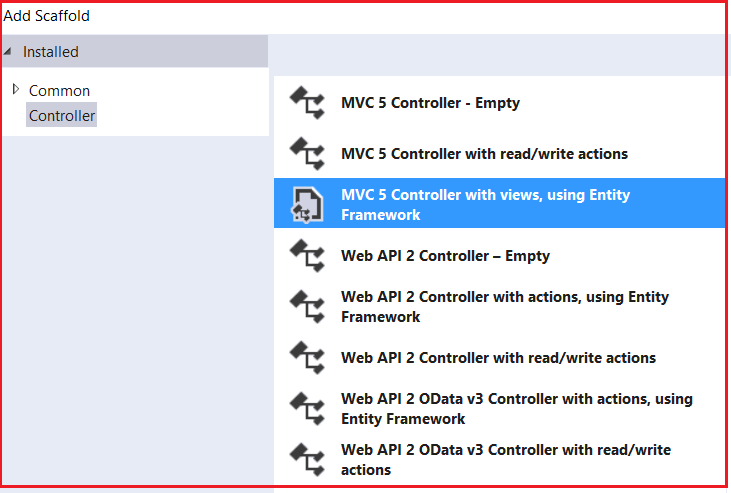
**Creating the ADO.NET Entity Data Model:**

Here we need to use the Entity Framework Database First Approach to create the Entities (i.e. **Employee, Users, RoleMaster and UserRolesMapping**) from the **MVC\_DB** database that we created in our previous step. Here I provided the context class name as **EmployeeDBContext**. Once the **EDMX** file is created build the solution. In this article, we will discuss how to use the Employee and Users entities and in the next article, we are going to how to use RoleMaster and UserRoleMapping entities. The EDMX file should looks as shown below.

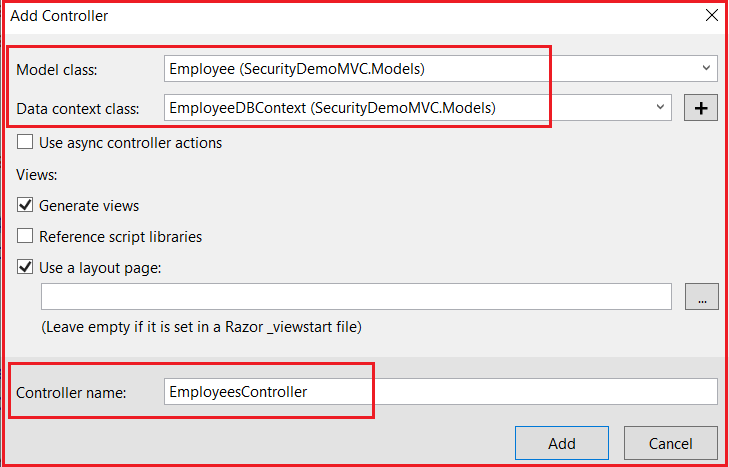


**Creating Employees Controller:**

Here we need to select the **MVC 5 Controller with Views, using Entity framework** option to create the controller as shown below.



After selecting the above controller, click on the **Add** button which will open the following popup for providing the required information to create the controller with necessary actions and related views.



As you can see in the above image, you need to select the **Model class as Employee** and the **Context class as EmployeeDBContext**. Provide the **controller name as EmployeesController** and then click on the **Add** button which will create the EmployeesController.

Now the employee controller is created with the required action methods and views to perform the CRUD operation against the Employee entity. Run the application and test by yourself. Here we are not going to focus on how it performs the CRUD operation rather we are going to focus on how to implement the **Forms Authentication in ASP.NET MVC Application**.

Now, the above application is accessible to each and everyone without any restrictions. Now we need to provide security to this application. So the user with proper credentials can only be able to access our application. To achieve this we are going to use the **Forms Authentication**.

**Implementing Forms Authentication in ASP.NET MVC:**

The **Forms Authentication** is available in **System.Web.Security** namespace. In order to implement the Forms Authentication in the ASP.NET MVC application, we need to do the following three things

1. Set the **Authentication Mode as Forms** in the web.config file
2. We need to use **FormsAuthentication.SetAuthCookie** for login
3. Again we need to use **FormAuthentication.SignOut** for logout

We are going to create another controller with the name **Accounts** to manage the user Signup, Sign in and log out functionalities.

**Creating Accounts Controller:**

Here you need to select the **MVC5 Controller – Empty** option and provide the name as **AccountsController**. Now open the Accounts Controller and add one action method with the name Login. Later we will implement this Login Action Method. Once you created the Login Action Method open the web.config file and add the following code within the **System.Web** section.

**<authentication** mode="Forms"**>**

**<forms** loginUrl="Accounts/Login"**></forms>**

**</authentication>**

Basically, what we are doing here is, whenever a user accesses to our webpages without login, we just navigating that user to the login page. The authentication **mode** stats that which type of authentication we are using, here we are using Forms authentication. The **login URL** is nothing but the URL to which the unauthenticated user navigates and in this case, this is nothing but the Login action method of the Accounts Controller.

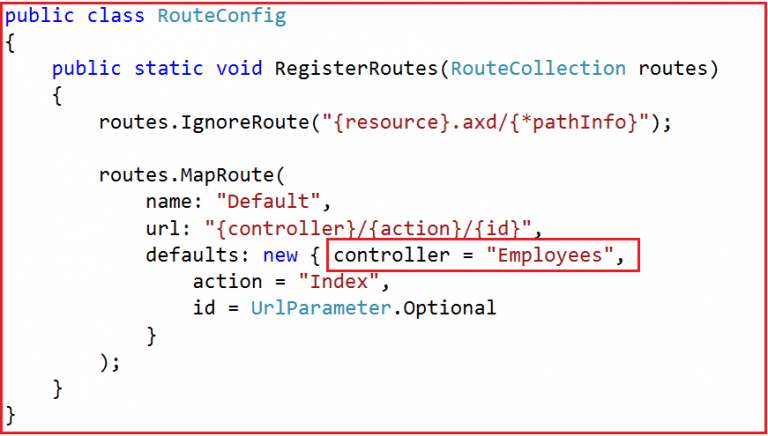
**Using Authorize Attribute:**

The Authorize Attribute is the built-in attribute provided by MVC which is basically used to authenticate a user. The action methods which you don’t want any anonymous user to access decorate those methods with Authorize Attribute. In our example, we don’t want to allow anonymous access to any of our action methods. So here I decorate the Authorize Attribute at the Controller level as shown below which will be applied to all the action methods of Employees Controller.



**Modifying the RouteConfig class:**

Within the Route Config file, we need to set the **Employees** controller as the default controller for our application as shown below.



**Creating the UserModel Model:**

Now we need to create a model within the Models folder within the name **UserModel** and then copy and paste the following code in it.

**namespace** *SecurityDemoMVC.Models*

**{**

**public** **class** UserModel

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string UserName **{** **get**; **set**; **}**

**public** string UserPassword **{** **get**; **set**; **}**

**}**

**}**

We created the above UserModel class with three properties. If you remember the Users model also contains the above three properties.

**Implementing the Login Action Method:**

Please add the following two methods within the Accounts Controller.

**public** ActionResult Login**()**

**{**

**return** View**()**;

**}**

**[**HttpPost**]**

**public** ActionResult Login**(**UserModel model**)**

**{**

**using** **(**EmployeeDBContext context = new EmployeeDBContext**())**

**{**

**bool** IsValidUser = context.Users.Any**(**user =**>** user.UserName.ToLower**()** ==

model.UserName.ToLower**()** && user.UserPassword == model.UserPassword**)**;

**if** **(**IsValidUser**)**

**{**

FormsAuthentication.SetAuthCookie**(**model.UserName, **false)**;

**return** RedirectToAction**(**"Index", "Employees"**)**;

**}**

ModelState.AddModelError**(**"", "invalid Username or Password"**)**;

**return** View**()**;

**}**

**}**

As you can see in the Post method, first we validate the user and if the validation success then we call the **SetAuthCookie** method of **FormsAuthentication** class and then navigate the user to the Index method of Employees Controller. The **FormsAuthentication** class is available in **System.Web.Security** namespace. We are passing the username as the first parameter to the **SetAuthCookie** method.

**Creating the Login View:**

Create the Login view and then copy and paste the following code in it.

@model SecurityDemoMVC.Models.UserModel

@{

ViewBag.Title = "Login";

}

**<h2>**Login**</h2>**

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

**<div** class="form-horizontal"**>**

**<hr** **/>**

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

**<div** class="form-group"**>**

@Html.LabelFor(model => model.UserName, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.TextBoxFor(model => model.UserName, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.UserName, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.Label("Password", htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.TextBoxFor(model => model.UserPassword, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.UserPassword, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

**<div** class="col-md-offset-2 col-md-10"**>**

**<input** type="submit" value="Login" class="btn btn-default" **/>**

**</div>**

**</div>**

**</div>**

}

**<div>**

@Html.ActionLink("Click here to Signup", "Signup")

**</div>**

**Implementing the Signup Page:**

Please add the following two methods in Account Controller which are going to perform the Signup functionality.

**public** ActionResult Signup**()**

**{**

**return** View**()**;

**}**

**[**HttpPost**]**

**public** ActionResult Signup**(**User model**)**

**{**

**using** **(**EmployeeDBContext context = new EmployeeDBContext**())**

**{**

context.Users.Add**(**model**)**;

context.SaveChanges**()**;

**}**

**return** RedirectToAction**(**"Login"**)**;

**}**

As you can see, the above action methods are very straight forward. The Post action method is basically used to add the user into the Users table and then navigate to the Login action method.

**Creating the Signup View:**

Please add the Signup view and then copy and paste the following code in it.

@model SecurityDemoMVC.Models.User

@{

ViewBag.Title = "Signup";

}

**<h2>**Signup**</h2>**

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

**<div** class="form-horizontal"**>**

**<hr** **/>**

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

**<div** class="form-group"**>**

@Html.LabelFor(model => model.UserName, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.TextBoxFor(model => model.UserName, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.UserName, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.Label("Password", htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.PasswordFor(model => model.UserPassword, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.UserPassword, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

**<div** class="col-md-offset-2 col-md-10"**>**

**<input** type="submit" value="Signup" class="btn btn-default" **/>**

**</div>**

**</div>**

**</div>**

}

**<div>**

@Html.ActionLink("Click here to Login", "Login")

**</div>**

**Implementing the Logout functionality:**

Please add the following action method within the Accounts Controller which will Logout the user.

**public** ActionResult Logout**()**

**{**

FormsAuthentication.SignOut**()**;

**return** RedirectToAction**(**"Login"**)**;

**}**

As you can see, we call the **SignOut** method of **FormsAuthentication** method to log out the user and then we navigate the user to the Login action method.

**Modifying the Layout View:**

Please modify the \_layout view as shown below.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** charset="utf-8" **/>**

**<meta** name="viewport" content="width=device-width, initial-scale=1.0"**>**

**<title>**@ViewBag.Title - My ASP.NET Application**</title>**

**<link** href="~/Content/Site.css" rel="stylesheet" type="text/css" **/>**

**<link** href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" **/>**

**<script** src="~/Scripts/modernizr-2.6.2.js"**></script>**

**</head>**

**<body>**

**<div** class="navbar navbar-inverse navbar-fixed-top"**>**

**<div** class="container"**>**

**<div** class="navbar-header"**>**

**<button** type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse"**>**

**<span** class="icon-bar"**></span>**

**<span** class="icon-bar"**></span>**

**<span** class="icon-bar"**></span>**

**</button>**

@Html.ActionLink("Employee Portal", "Index", "Employees", new { area = "" }, new { @class = "navbar-brand" })

**</div>**

**<div** class="navbar-collapse collapse"**>**

**<ul** class="nav navbar-nav"**>**

@if (User.Identity.IsAuthenticated)

{

**<li>**@Html.ActionLink("Get Employee List", "Index", "Employees")**</li>**

**<li>**@Html.ActionLink("Add Employee", "Create", "Employees")**</li>**

**<li><a>**Hello - @User.Identity.Name**</a></li>**

**<li>**@Html.ActionLink("Logout", "Logout", "Accounts")**</li>**

}

else

{

**<li>**@Html.ActionLink("Login", "Login", "Accounts")**</li>**

}

**</ul>**

**</div>**

**</div>**

**</div>**

**<div** class="container body-content"**>**

@RenderBody()

**<hr** **/>**

**<footer>**

**<p>**&copy; @DateTime.Now.Year - My ASP.NET Application**</p>**

**</footer>**

**</div>**

**<script** src="~/Scripts/jquery-1.10.2.min.js"**></script>**

**<script** src="~/Scripts/bootstrap.min.js"**></script>**

**</body>**

**</html>**

**Note:** The **User.Identity.IsAuthenticated** property returns true if the user is authenticated and the **User.Identity.Name** property returns the name of the authenticated user which we provide to the **SetAuthCookie** method which we implement in the **Login** Post method of Accounts Controller.

That’s it. We have done the implementation with the Forms Authentication in ASP.NET MVC Application. Now run the application and test everything is working as expected. In the next article, I am going to discuss the [**Role-Based Authentication in ASP.NET MVC**](https://dotnettutorials.net/lesson/role-based-authentication-in-mvc/) application. I hope you understood how to implement **FormsAuthentication in ASP.NET MVC** Application.

**Role-Based Authentication in ASP.NET MVC**

**Role-Based Authentication in ASP.NET MVC**

In this article, I am going to discuss how to implement **Role-Based Authentication in the ASP.NET MVC** application. I strongly recommended reading our previous article before proceeding to this article as it is a continuation part of our previous article. In our previous article, we discussed how to implement [**Forms Authentication in ASP.NET MVC**](https://dotnettutorials.net/lesson/forms-authentication-in-mvc/) as well as we also created the required database tables. As part of this article, we are going to discuss the following things in detail.

1. **What are the Roles?**
2. **What is the need for Role-Based Authentication?**
3. **How to implement Role-Based Authentication?**

**What are the Roles?**

Roles are nothing but the permissions given to a particular user to access some resources. So in some other words, we can say that, once a user is authenticated then what are the resources the user can access are determined by his roles. A single user can have multiple roles and Roles plays an important part in providing security to the system. For example, Admin, Customer, Accountant, etc.

**SQL Script:**

In order to understand the Roles, let add some data into the tables. Please use the below SQL Script to insert some test data to Employee, Users, RoleMaster, and UserRolesMapping table.

-- Inserting data into Employee table

**INSERT** **INTO** Employee **VALUES**('Anurag', 'Software Engineer', 10000)

**INSERT** **INTO** Employee **VALUES**('Preety', 'Tester', 20000)

**INSERT** **INTO** Employee **VALUES**('Priyanka', 'Software Engineer', 20000)

**INSERT** **INTO** Employee **VALUES**('Ramesh', 'Team Lead', 10000)

**INSERT** **INTO** Employee **VALUES**('Santosh', 'Tester', 15000)

-- Inserting data into Users table

**INSERT** **INTO** Users **VALUES**('Admin','admin')

**INSERT** **INTO** Users **VALUES**('User','user')

**INSERT** **INTO** Users **VALUES**('Customer','customer')

-- Inserting data into Role Master table

**INSERT** **INTO** RoleMaster **VALUES**('Admin')

**INSERT** **INTO** RoleMaster **VALUES**('User')

**INSERT** **INTO** RoleMaster **VALUES**('Customer')

-- Inserting data into User Roll Mapping table

**INSERT** **INTO** UserRolesMapping **VALUES**(1, 1, 1)

**INSERT** **INTO** UserRolesMapping **VALUES**(2, 1, 2)

**INSERT** **INTO** UserRolesMapping **VALUES**(3, 1, 3)

**INSERT** **INTO** UserRolesMapping **VALUES**(4, 2, 2)

**INSERT** **INTO** UserRolesMapping **VALUES**(5, 3, 3)

As you can see, the user with id 1 having three roles whiles the user with id 2 and 3 having only one role.

**Creating the Role Provider:**

Create a class file with the name **UsersRoleProvider** within the Models folder and then copy and paste the following code. This class implements the **RoleProvider** class. If you go to the definition of **RoleProvider** class then you can see it is an **abstract class**. As it is an abstract class we need to implement all the methods of that class. The **RoleProvider** class belongs to **System.Web.Security** namespace.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**using** *System.Web;*

**using** *System.Web.Security;*

**namespace** *SecurityDemoMVC.Models*

**{**

**public** **class** UsersRoleProvider : RoleProvider

**{**

**public** **override** string ApplicationName

**{**

**get**

**{**

**throw** new NotImplementedException**()**;

**}**

**set**

**{**

**throw** new NotImplementedException**()**;

**}**

**}**

**public** **override** **void** AddUsersToRoles**(**string**[]** usernames, string**[]** roleNames**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **override** **void** CreateRole**(**string roleName**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **override** **bool** DeleteRole**(**string roleName, **bool** throwOnPopulatedRole**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **override** string**[]** FindUsersInRole**(**string roleName, string usernameToMatch**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **override** string**[]** GetAllRoles**()**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **override** string**[]** GetRolesForUser**(**string username**)**

**{**

**using** **(**EmployeeDBContext context = new EmployeeDBContext**())**

**{**

var userRoles = **(from** user in context.Users

**join** roleMapping in context.UserRolesMappings

**on** user.ID **equals** roleMapping.UserID

**join** role in context.RoleMasters

**on** roleMapping.RoleID **equals** role.ID

**where** user.UserName == username

**select** role.RollName**)**.ToArray**()**;

**return** userRoles;

**}**

**}**

**public** **override** string**[]** GetUsersInRole**(**string roleName**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **override** **bool** IsUserInRole**(**string username, string roleName**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **override** **void** RemoveUsersFromRoles**(**string**[]** usernames, string**[]** roleNames**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **override** **bool** RoleExists**(**string roleName**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**}**

**}**

In the above class, we only modify the implementation of the **GetRolesForUser** method. This method takes the Username as an input parameter and based on the username we need to fetch the User Roles as an array and return that array.

**Configuring Role Provider in the web.config file:**

Add the following code within the system.web section of your web.config file.

**<roleManager** defaultProvider="usersRoleProvider" enabled="true" **>**

**<providers>**

**<clear/>**

**<add** name="usersRoleProvider" type="SecurityDemoMVC.Models.UsersRoleProvider"**/>**

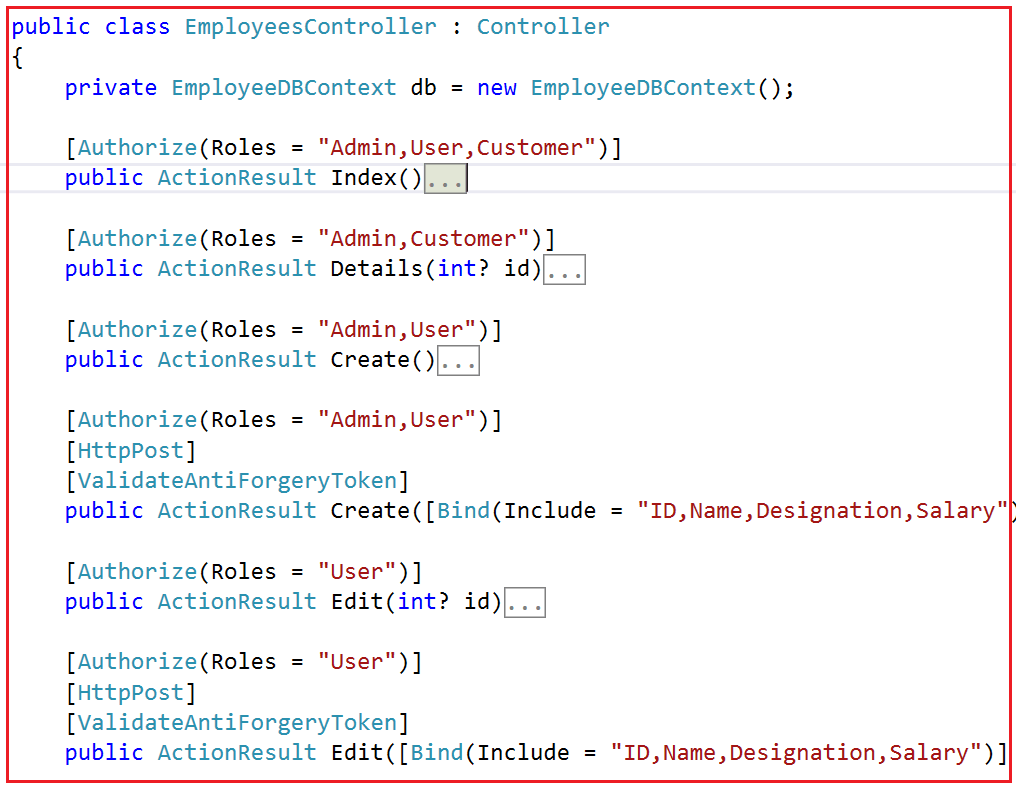
**</providers>**

**</roleManager>**

Basically here we are adding our Role Providers. Before adding the Role Providers first we clear all roles. The name you can give anything but the type value is going to be the full name of your Role Provider i.e. including the namespace. Here you can add any number of Role Providers. You have to provide the default provider which is going to be used as default in the default provider parameter of role manager and you need to enable it by setting the value to true of enabled property.

**Modifying the Employees Controller:**

Please modify the Authorize attribute to include Roles as shown below.



First, we remove the Authorize attribute from the Controller Level and applied it at the action method level. Here you can pass multiple roles separated by a comma. As per your business requirement set the Roles and test by yourself.

**Role-Based Menus in MVC**

**Role-Based Menus in MVC**

In this article, I am going to discuss how to implement **Role-Based Menus in MVC** Application. I strongly recommended you to reads our previous two articles before proceeding to this article as it is a continuation part of our previous two articles. In our previous two articles, we discussed how to implement [**Forms Authentication in MVC**](https://dotnettutorials.net/lesson/forms-authentication-in-mvc/) and how to implement [**Role-Based Authentication in MVC**](https://dotnettutorials.net/lesson/role-based-authentication-in-mvc/) application.

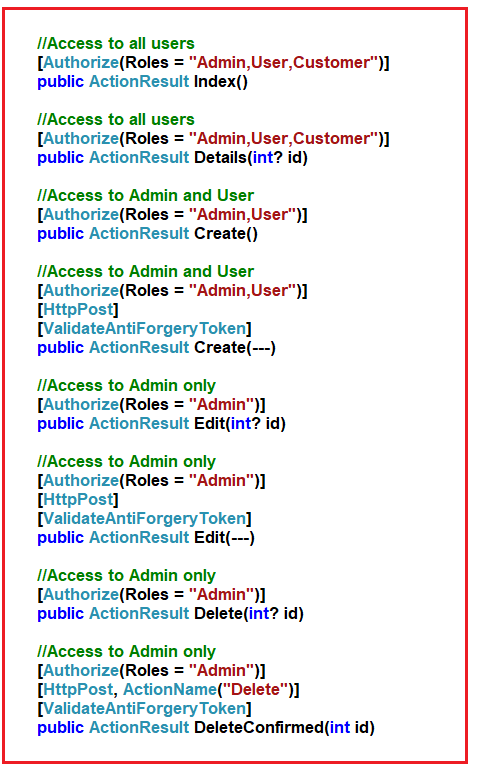
**Business Requirement:**

Our business requirement is we need to display Menus based on the Roles provided to the user. In our previous article, we created three different types of roles such as Admin, User, and Customer, and then assigned these to different types of users.

The Employee Controller Contains 4 views such as Index, Details, Create, Update and Delete. Our requirement is we need to show all menus or links to the users whose role is Admin. Along the same line, we need to show the Create and Index and Details menus or link to the users whose role is User. The users having the role Customer can only see the Index and Details view.

**Let us see how to implement this:**

First of all, modify the Controller class as shown below. Here we are applying Roles to individual action methods. This is important from the security point of view. This is because sometimes we may directly access the views using some tools like Postman and Fiddler.



**Modifying the \_Layout.cshtml file:**

Please modify the layout file as shown below to implement role-based menus.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** charset="utf-8" **/>**

**<meta** name="viewport" content="width=device-width, initial-scale=1.0"**>**

**<title>**@ViewBag.Title - My ASP.NET Application**</title>**

**<link** href="~/Content/Site.css" rel="stylesheet" type="text/css" **/>**

**<link** href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" **/>**

**<script** src="~/Scripts/modernizr-2.6.2.js"**></script>**

**</head>**

**<body>**

**<div** class="navbar navbar-inverse navbar-fixed-top"**>**

**<div** class="container"**>**

**<div** class="navbar-header"**>**

**<button** type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse"**>**

**<span** class="icon-bar"**></span>**

**<span** class="icon-bar"**></span>**

**<span** class="icon-bar"**></span>**

**</button>**

@Html.ActionLink("Employee Portal", "Index", "Employees", new { area = "" }, new { @class = "navbar-brand" })

**</div>**

**<div** class="navbar-collapse collapse"**>**

**<ul** class="nav navbar-nav"**>**

@if (User.Identity.IsAuthenticated)

{

**<li>**@Html.ActionLink("Get List", "Index", "Employees")**</li>**

if (User.IsInRole("Admin") || User.IsInRole("User"))

{

**<li>**@Html.ActionLink("Create", "Create", "Employees")**</li>**

}

**<li><a>**Hello - @User.Identity.Name**</a></li>**

**<li>**@Html.ActionLink("Logout", "Logout", "Accounts")**</li>**

}

else

{

**<li>**@Html.ActionLink("Login", "Login", "Accounts")**</li>**

}

**</ul>**

**</div>**

**</div>**

**</div>**

**<div** class="container body-content"**>**

@RenderBody()

**<hr** **/>**

**<footer>**

**<p>**&copy; @DateTime.Now.Year - My ASP.NET Application**</p>**

**</footer>**

**</div>**

**<script** src="~/Scripts/jquery-1.10.2.min.js"**></script>**

**<script** src="~/Scripts/bootstrap.min.js"**></script>**

**</body>**

**</html>**

**Note:** The **User.IsInRole()** method takes the Role name as input and returns true or false based on the logged-in user role.

**Modifying the Index View:**

Let us do the same thing within the index view where we show the buttons for details, edit and delete. So, modify the Index view of Employees controller as shown below.

@model IEnumerable**<SecurityDemoMVC.Models.Employee>**

@{

ViewBag.Title = "Index";

}

**<h2>**Index**</h2>**

**<p>**

@Html.ActionLink("Create New", "Create")

**</p>**

**<table** class="table"**>**

**<tr>**

**<th>**

@Html.DisplayNameFor(model => model.Name)

**</th>**

**<th>**

@Html.DisplayNameFor(model => model.Designation)

**</th>**

**<th>**

@Html.DisplayNameFor(model => model.Salary)

**</th>**

**<th></th>**

**</tr>**

@foreach (var item in Model) {

**<tr>**

**<td>**

@Html.DisplayFor(modelItem => item.Name)

**</td>**

**<td>**

@Html.DisplayFor(modelItem => item.Designation)

**</td>**

**<td>**

@Html.DisplayFor(modelItem => item.Salary)

**</td>**

**<td>**

@if (User.IsInRole("Admin"))

{

@Html.ActionLink("Edit", "Edit", new { id = item.ID }) **<text>**|**</text>**

@Html.ActionLink("Delete", "Delete", new { id = item.ID }) **<text>**|**</text>**

}

@Html.ActionLink("Details", "Details", new { id=item.ID })

**</td>**

**</tr>**

}

**</table>**

That’s it. We are done with our implementation. Now run the application and see everything is working as expected.