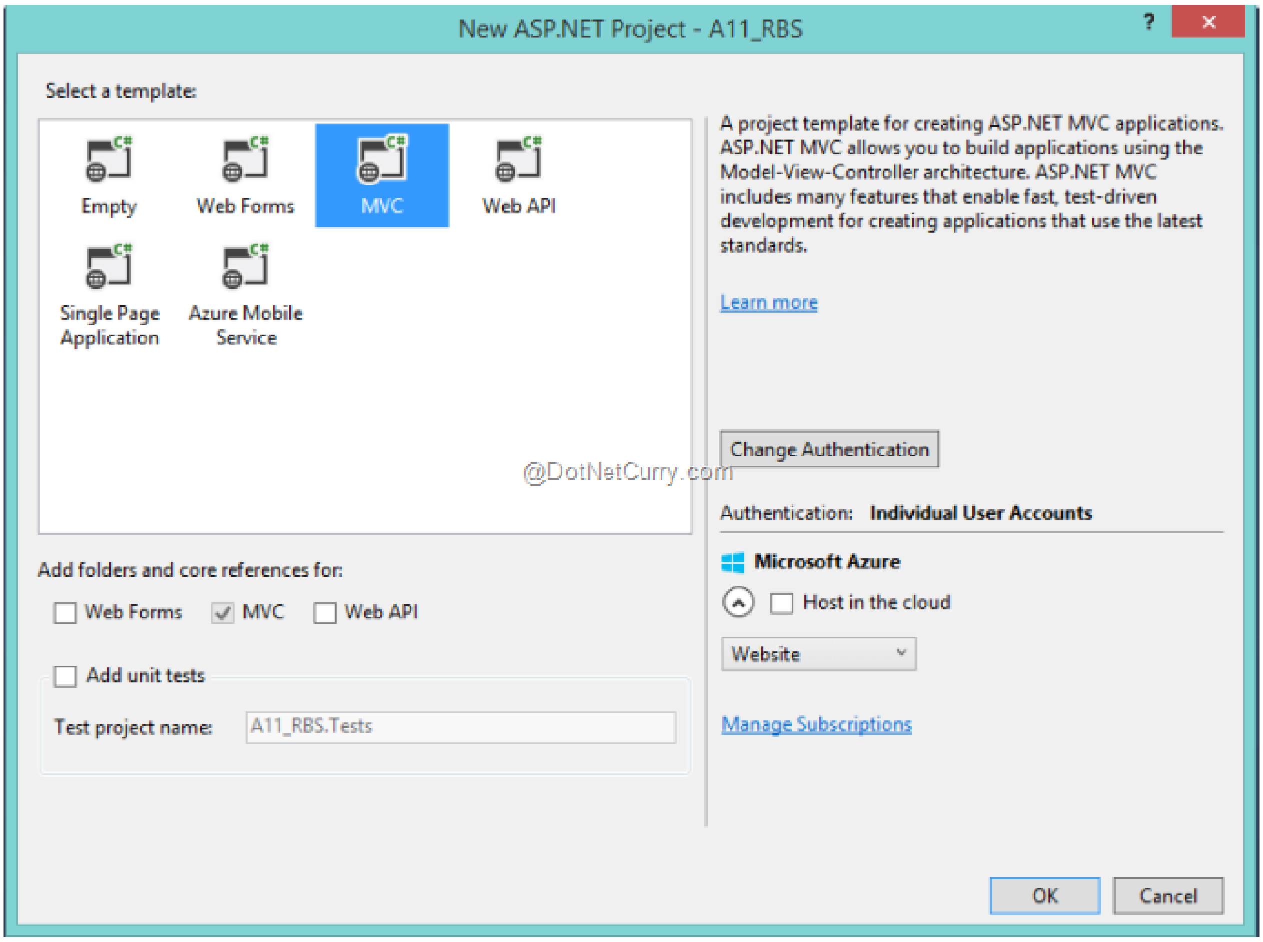
UNIVERSITY of GREENWICH
Alliance with FPT Education

Lab: ASP.NET Identity

Creating ASP.NET MVC Application, Configuring and Creating Roles and Users

Step 1: Open the [Free Visual Studio Community Edition]() and create an MVC application of the name A11_RBS**.** Select a MVC Template as shown below:



A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Authentication: **Individual User Accounts**

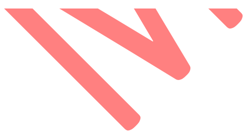
Add folders and core references for:

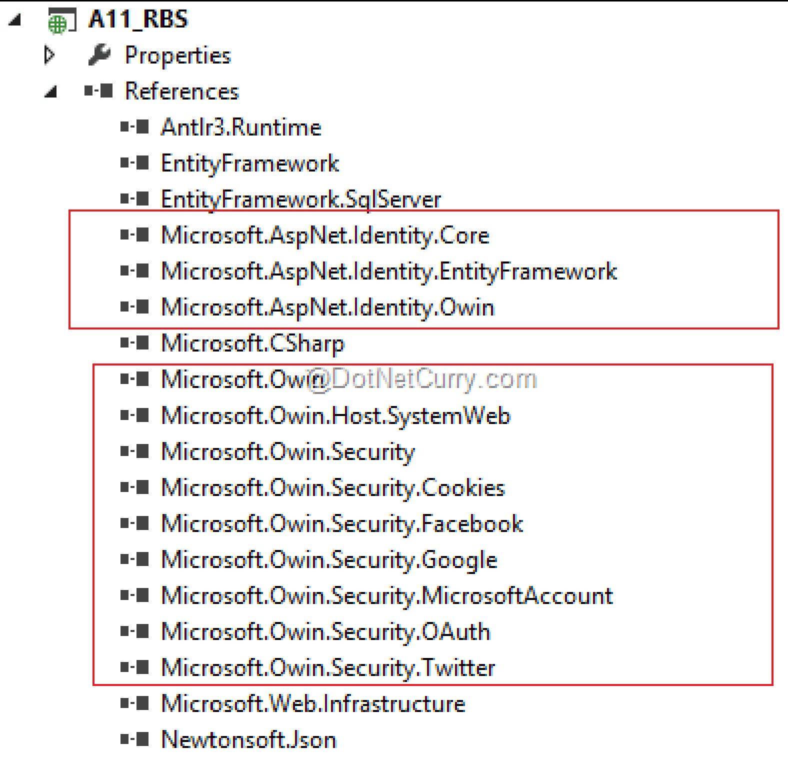
☐ Web Forms ☑ MVC ☐ Web API

☐ Add unit tests

Test project name: A11_RBS.Tests

The MVC project will be created with the ready references for ASP.NET Identity:





```
A11_RBS
    Properties
    References
        Antlr3.Runtime
        EntityFramework
        EntityFramework.SqlServer
        Microsoft.AspNet.Identity.Core
        Microsoft.AspNet.Identity.EntityFramework
        Microsoft.AspNet.Identity.Owin
        Microsoft.CSharp
        Microsoft.Owin
        Microsoft.Owin.Host.SystemWeb
        Microsoft.Owin.Security
        Microsoft.Owin.Security.Cookies
        Microsoft.Owin.Security.Facebook
        Microsoft.Owin.Security.Google
        Microsoft.Owin.Security.MicrosoftAccount
        Microsoft.Owin.Security.OAuth
        Microsoft.Owin.Security.Twitter
        Microsoft.Web.Infrastructure
        Newtonsoft.Json
```

In the Models folder of the project, we have an *IdentityModel.cs* class file. This contains *ApplicationDbContext* class which is used to connect to the database where the Users and Roles Table will be created. The class file *AccountViewModels.cs* contains classes for Login, Register, etc. In this file locate the *RegisterViewModel* class and add the following string property:

```
public string Name { get; set; }
```

This property will be used to assign role to the user when a new user is registered in the application.

Step 2: Now it is time to create a view for creating Roles for the application. To do so, in the Controllers' folder add a new Empty MVC controller of name *RoleController*. In this controller add the following code:

```
using System;
using System.Linq;
using System.Web.Mvc;

using A11_RBS.Models;
using Microsoft.AspNet.Identity.EntityFramework;

namespace A11_RBS.Controllers
{
    public class RoleController : Controller
    {
        ApplicationDbContext context;

        public RoleController()
        {
            context = new ApplicationDbContext();
        }
        public ActionResult Index()
        {
            var Roles = context.Roles.ToList();
            return View(Roles);
        }

        public ActionResult Create()
        {
            var Role = new IdentityRole();
            return View(Role);
        }

        [HttpPost]
        public ActionResult Create(IdentityRole Role)
        {
            context.Roles.Add(Role);
            context.SaveChanges();
            return RedirectToAction("Index");
        }
    }
}
```

Role creation is done using *IdentityRole* class. This class provides properties e.g. Id, Name, etc for creating roles for the applications. Scaffold the Index and Create view, using Index and Create Action method from the RoleController class.

Index.cshtml

```
@model IEnumerable<Microsoft.AspNet.Identity.EntityFramework.IdentityRole>
@{
    ViewBag.Title = "Index";
}

<h2>Available Roles For Application</h2>

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

<style type="text/css">
    #tbrole,.c {
    border:double;
    }
</style>

<table id="tbrole">
    <tr>
        <td class="c">
            Role Name
        </td>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td class="c">
                @item.Name
            </td>
        </tr>
    }
</table>
```

Create.cshtml

```
@model Microsoft.AspNet.Identity.EntityFramework.IdentityRole
@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>

<style type="text/css">
    #tbrole, .c {
        border: double;
    }
</style>


@using (Html.BeginForm())
{
    <table id="tbrole">
        <tr>
            <td class="c">Enter Role Name To be Created:</td>
            <td class="c">
                @Html.EditorFor(m => m.Name)
            </td>
        </tr>
    </table>
    <input type="submit" value="Create Role" />
}
```

Step 3: Open *AccountController.cs* and create an instance of *ApplicationDbContext* class in the AccountController Constructor.

```
ApplicationDbContext context;

public AccountController()
{
    context = new ApplicationDbContext();
}
```

Add the following code in the *Register()* action method:

```
[AllowAnonymous]
public ActionResult Register()
{
    ViewBag.Name = new SelectList(context.Roles.ToList(), "Name", "Name");
    return View();
}
```

This code provides the Roles information to the Register View, so that when a new user is registered with the application, they will be given the desired Role. Open Register.cshtml view in the Account sub folder of Views folder and add the following Html Helper in it above the *Submit* button.

```
<!--Select the Role Type for the User-->
<div class="form-group">
    @Html.Label("Select Your User Type", new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.DropDownList("Name")
    </div>
```

```
</div>
<!--Ends Here-->
```

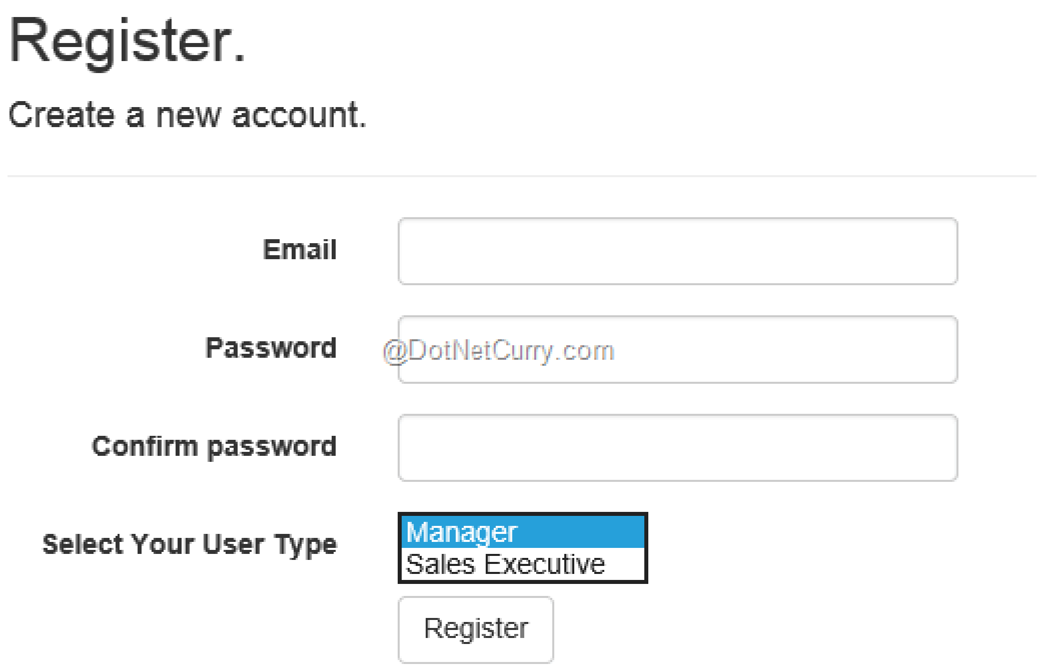
To complete the operation of Assigning Role to the user, change the *Register()* action method with *HttpPost* in the AccountController as shown below:

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {

            //Assign Role to user Here
            await this.UserManager.AddToRoleAsync(user.Id, model.Name);
            //Ends Here


            await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);

            return RedirectToAction("Index", "Home");
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Step 4: In the _Layout.cshtml view of the Shared sub folder of the Views folder, add the following piece of code for showing link for Role in a <div> element with the class *navbar-collapse collapse.*

```
<li>@Html.ActionLink("Role", "Index", "Role")</li>
```

Step 5: Run the Application, navigate to the Create View for the RoleController and create *Manager* and *Sales Executive* roles. Now navigate to the Register Action of the Account Controller and create users with roles:


Register.

Create a new account.

Email

Password

@DotNetCurry.com

Confirm password

Select Your User Type

Manager

Sales Executive

Register

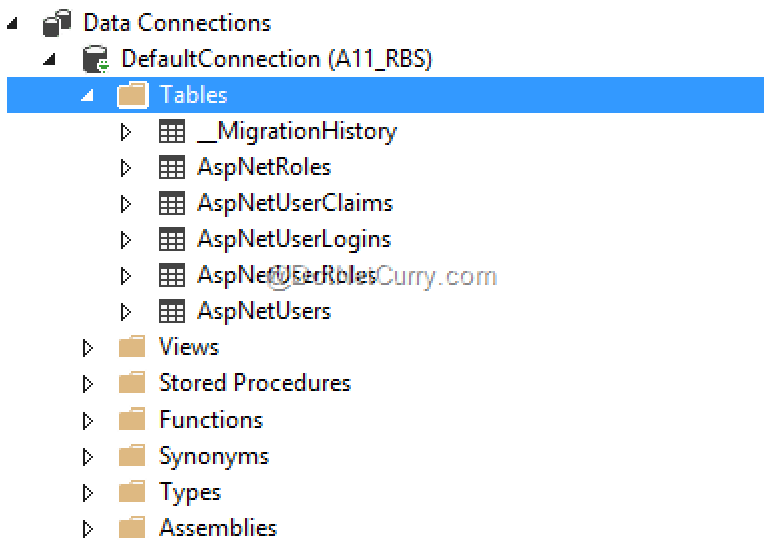
When the Register button is clicked, the user will be registered either as Manager or Sales Executive.

Create Two users:

Mahesh1@user.com - This is Manager Role user

User1.user@user.com - This is Sales Executive Role user

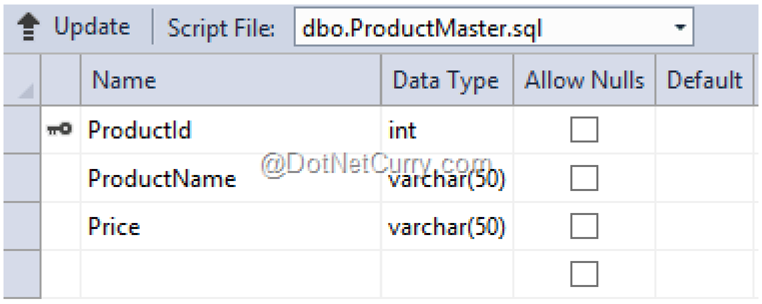
Stop the application and check the server explorer. It will display tables for Users and Roles:


```
◢ Data Connections
  ◢ DefaultConnection (A11_RBS)
    ◢ Tables
      ▷ __MigrationHistory
      ▷ AspNetRoles
      ▷ AspNetUserClaims
      ▷ AspNetUserLogins
      ▷ AspNetUserRoles
      ▷ AspNetUsers
    ▷ Views
    ▷ Stored Procedures
    ▷ Functions
    ▷ Synonyms
    ▷ Types
    ▷ Assemblies
```
@DotNetCurry.com

Using Authorization for controlling Access of the Action methods of controller

Once the roles and users are created and configured, it's time for us to manage them and define access to the application. We will be implementing a simple application of selling products in a Super Market. The product can be added by the Manager Role where as Sales Executive can sell it.

Step 1: In the App_Data folder, add a new Sql Server database of name *SuperMarket*. In this database add the following ProductMaster table:



Update | Script File: dbo.ProductMaster.sql

Name	Data Type	Allow Nulls	Default
ProductId	int	☐	
ProductName	varchar(50)	☐	
Price	varchar(50)	☐	
		☐	

Step 2: In the Models folder, add a new ADO.NET Entity Data Model. In the wizard select SuperMarket.mdf and select ProductMaster table. Complete the wizard to generate table mapping. Build the project.

Step 3: In the Controllers, add a new empty MVC Controller of the name ProductController. Add the action methods in this controller:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

using A11_RBS.Models;

namespace A11_RBS.Controllers
{
    public class ProductController : Controller
    {
        SuperMarketEntities ctx;

        public ProductController()
        {
            ctx = new SuperMarketEntities();
        }

        // GET: Product
        public ActionResult Index()
        {
            var Products = ctx.ProductMasters.ToList();
            return View(Products);
        }


        public ActionResult Create()
        {
            var Product = new ProductMaster();
            return View(Product);
        }

        [HttpPost]
        public ActionResult Create(ProductMaster p)
        {
            ctx.ProductMasters.Add(p);
            ctx.SaveChanges();
            return RedirectToAction("Index");
        }

        public ActionResult SaleProduct()
        {
            ViewBag.Message = "This View is designed for the Sales Executive to Sale Product.";
            return View();
        }
    }
}
```

The ProductController contains Index, Create and SaleProduct action methods. Scaffold Views from these action methods.

Step 4: Since we want to configure the Create Action to Manager Role and SaleProduct to Sales Executive Role, we need to apply *[Authorize (Role="<Role Name>")]* on these methods. But if the user is not authorized to perform a specific action, we need to navigate to an error page. To implement this we need to add the Custom Action filter for Authorization and the Error View.

Step 5: In the Views folder, we have a Shared Subfolder. In this folder add a new View Empty Model of the name *AuthorizeFailed.cshtml* as shown here:

```
@{
    ViewBag.Title = "AuthorizeFailed";
}

<h2>Authorize Failed</h2>

@ViewData["Message"]
```

Step 6: In the project add a new folder called *CustomFilters* and add the class file with following login logic in it:

```
using System.Web.Mvc;

namespace A11_RBS.CustomFilters
{
    public class AuthLogAttribute : AuthorizeAttribute
    {
        public AuthLogAttribute()
        {
            View = "AuthorizeFailed";
        }

        public string View { get; set; }

        public override void OnAuthorization(AuthorizationContext filterContext)
        {
            base.OnAuthorization(filterContext);
            IsUserAuthorized(filterContext);
        }
        private void IsUserAuthorized(AuthorizationContext filterContext)
        {

            if (filterContext.Result == null)
                return;


            if (filterContext.HttpContext.User.Identity.IsAuthenticated)
            {

                var vr = new ViewResult();
                vr.ViewName = View;

                ViewDataDictionary dict = new ViewDataDictionary();
                dict.Add("Message", "Sorry you are not Authorized to Perform this
Action");

                vr.ViewData = dict;

                var result = vr;

                filterContext.Result = result;
            }
        }
    }
}
```

The above custom filter is derived from *AuthorizeAttribute* class and overrides the *OnAuthorization()* method. The *IsUserAuthorized()* helper method checks the user authentication with *AuthorizationContext* class. If the *Result* property returns null, then the Authorization is successful and user can continue the operation; else if the user is authenticated but not authorized, then an Error Page will be returned.

Step 7: Open the ProductController and add the *AuthLog* attribute on Create and SaleProduct action methods:

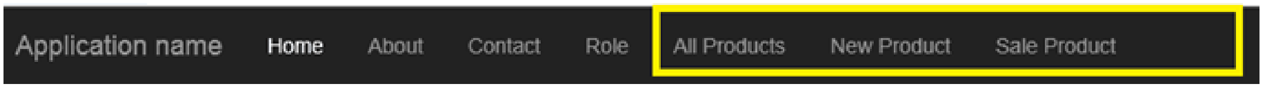
```
[AuthLog(Roles = "Manager")]
public ActionResult Create()
{
    var Product = new ProductMaster();
    return View(Product);
}

[AuthLog(Roles = "Sales Executive")]
public ActionResult SaleProduct()
{
    ViewBag.Message = "This View is designed for the Sales Executive to Sale Product.";
    return View();
}
```

The create method is Authorized to all Manager Role users, while SaleProduct is for Sales Executive.

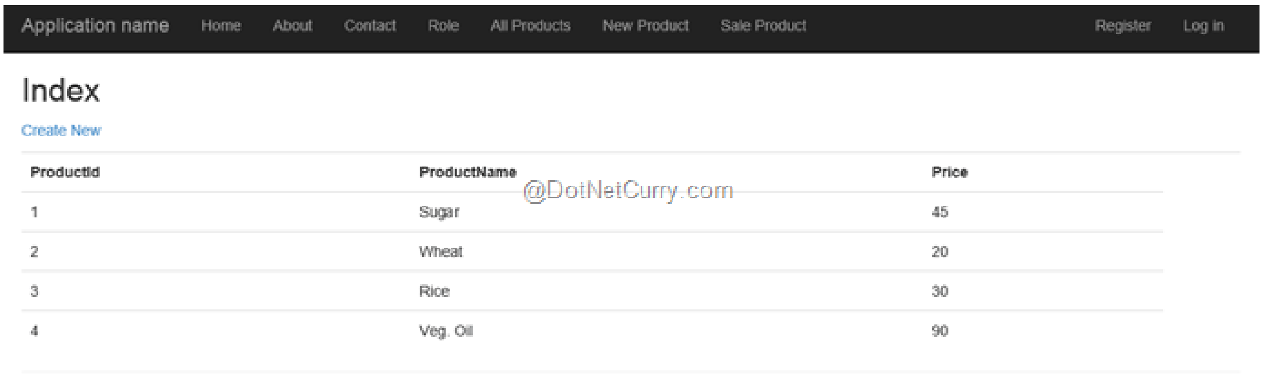
Step 8: Open the _Layout.cshtml from Views/Shared folder path and add the links for All Products, New Product and Sale Product in a <div> with *navbar-collapse collapse* class.

```
<li>@Html.ActionLink("All Products", "Index", "Product")</li>
<li>@Html.ActionLink("New Product", "Create", "Product")</li>
<li>@Html.ActionLink("Sale Product", "SaleProduct", "Product")</li>
```

Run the application and the following page will be displayed:



Click on All Products to bring up an Index View with all products:

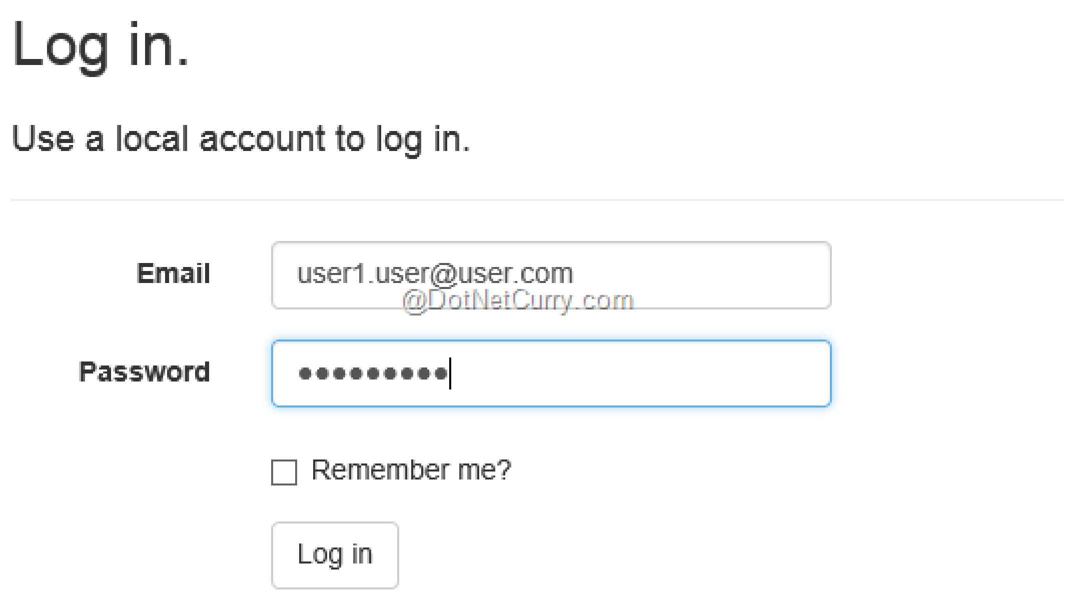


Index

Create New

ProductId	ProductName	Price
1	Sugar	45
2	Wheat	20
3	Rice	30
4	Veg. Oil	90

Click on the *Create New* link or *New Product Link* and a login page gets displayed. Enter Credentials for the Manager User (Mahesh1@user.com) and a Create Product view gets displayed. However instead of the manager, if a Sales Executive role user's credentials (user1.user@user.com) were used, an error page would be displayed:



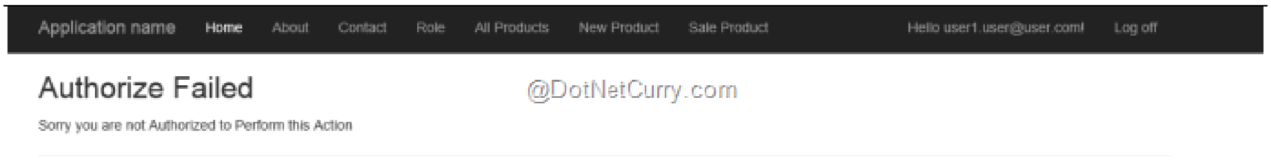
Log in.

Use a local account to log in.

Email: user1.user@user.com

Password: ●●●●●●●●●●

Login with user1.user@use.com but since this is a Sales Executive role, the user cannot create a new product and hence the error page will be displayed:



Authorize Failed

Sorry you are not Authorized to Perform this Action

Conclusion: The IdentityRole class in ASP.NET Identity provides some useful features for creating and managing Roles in an application.