

- [Slashdot](#)
- [StumbleUpon](#)
- [Technorati](#)
- [Twitter](#)
- [Windows Live](#)
- [Yahoo Buzz](#)
- [FriendFeed](#)

•

Securing ASP.NET MVC Applications with ASP.NET Identity

- 0.25
- 0.50
- 0.75
- 1.00
- 1.25
- 1.50
- 1.75
- 2.00
- 2.25
- 2.50
- 2.75
- 3.00
- 3.25
- 3.50
- 3.75
- 4.00
- 4.25
- 4.50
- 4.75
- 5.00

Posted by [Bipin Joshi](#) on August 15th, 2014

-

•

Tweet

Like 0

Share

G+

•

WEBINAR: On-demand webcast How to Boost Database Development Productivity on Linux, Docker, and Kubernetes with Microsoft SQL Server 2017


REGISTER >

Introduction

ASP.NET offers Forms Authentication as one of the authentication schemes. Developers often use Forms Authentication in combination with membership, roles and profile features to provide security to their web applications. Over the years the needs of authentication schemes used in a web application have changed. To take into account these changing trends Microsoft has released ASP.NET Identity - a new way to authenticate users of your web application. This article describes how ASP.NET Identity can be used in an ASP.NET MVC application from the ground up.

Overview of ASP.NET Identity

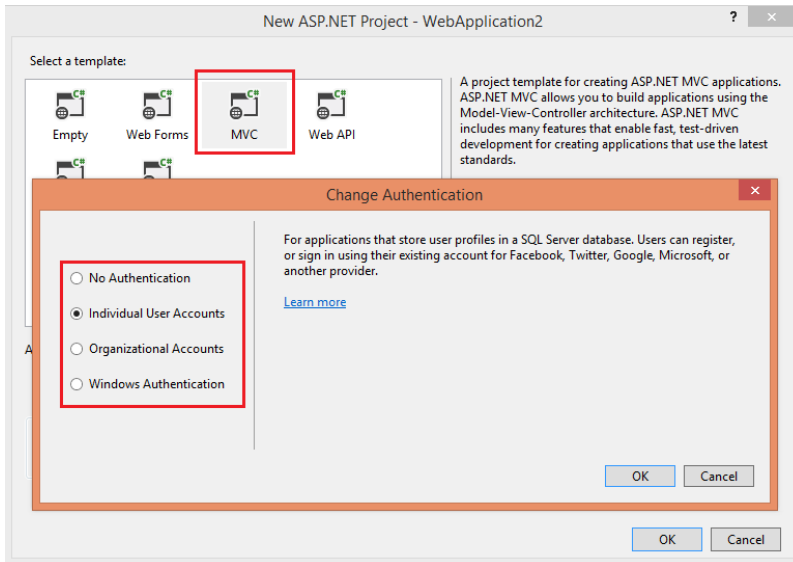
Developers often use Forms Authentication combined with Membership, Roles and Profile features to secure their web applications. Over the years these features have become inadequate to deal with changing trends of web application security. For example, many modern web sites use Facebook or OAuth based external user credentials. ASP.NET membership has no inbuilt way to deal with these situations. There are many other limitations of the membership system such as rigid database structure and complex object model. We won't go into the details of these limitations in this article. You may read the "Extra Reading Links" given at the end of this article.



Open Compute Platforms Power Software-Driven Packet Flow Visibility, Part 2

Download Now

ASP.NET Identity is a new authentication system that is intended to replace the existing membership system of ASP.NET. ASP.NET Identity is an OWIN (Open Web Interface for .NET) based library. Visual Studio 2013 project templates allow you to use ASP.NET Identity for securing the web application being created. Have a look at the following figure that shows the project template dialog of Visual Studio 2013.



Project Template Dialog of Visual Studio 2013

When you select MVC project template you will see the Change Authentication button enabled. Clicking on the button will open the Change Authentication dialog as shown above. The default selection of "Individual User Accounts" indicates that user account information will be stored in the application database (that means users won't use any external / OAuth based logins).

If you create an MVC project with this default selection, you will find that the project template includes AccountController and associated views for registering new users as well as for authenticating users. You will also find reference to a few OWIN assemblies added for you along with an OWIN startup class. Many beginners find this automatically added code difficult to understand. That is why this article shows you how to implement ASP.NET Identity in an empty MVC project from the ground up. Remember that this article discusses only the local user accounts. Any discussion of external logins (such as Facebook or any OAuth compatible system) is beyond the scope of this article.

Before you actually start the development of the example web application, it is important to familiarize yourself with the parts of ASP.NET Identity. Knowing these parts will let you code and use them easily in your web applications.

There are six important pieces of ASP.NET Identity system as far as local user accounts are concerned. They are briefly discussed below:

- User
- Role
- User Manager
- Role Manager
- Authentication Manager
- Entity Framework DbContext

A User object represents a user of the system. The basic authentication details such as user ID and password as well as profile information of a user make a User object. ASP.NET Identity comes with the IdentityUser class that captures basic authentication information. If you also need to capture profile information, you can create a custom class that inherits from the **IdentityUser** base class. This class is analogous to the MembershipUser class of the ASP.NET membership system.

A Role object represents a user role. At a minimum a role has a name with which it is identified in the system. The **IdentityRole** class of ASP.NET Identity provides this basic role. If you wish to add some more pieces to the role (say description of a role) then you can create a custom class that inherits from the IdentityRole base class.

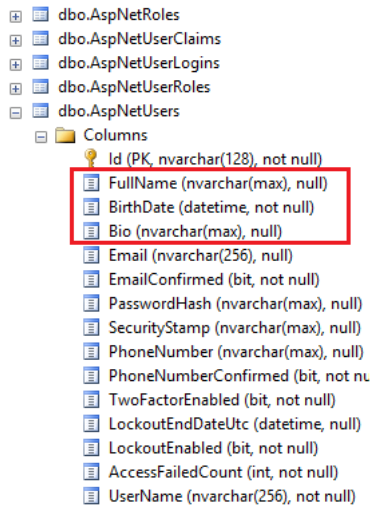
A User Manager is a class that allows you to manage users. Creating user accounts, removing user accounts, changing passwords, adding / removing users to a role and such tasks can be performed using a user manager. ASP.NET Identity comes with the **UserManager** class that can be used for this purpose. This class is analogous to the Membership intrinsic object of the ASP.NET membership system.

A Role Manager is a class that allows you to manage roles. Creating a role, removing a role, checking whether a role exists in the system and such tasks can be performed using a role manager. ASP.NET Identity provides the **RoleManager** class that can be used for this purpose. This class is analogous to the Roles intrinsic object of the ASP.NET membership system.

All the classes mentioned above deal with users and roles respectively. These classes by themselves don't perform any authentication. Authenticating a user - signing in and signing out a user - is the responsibility of Authentication Manager. The local user accounts can use cookie based authentication similar to Forms Authentication. ASP.NET Identity provides the **IAuthenticationManager** interface that represents an authentication manager. An authentication manager is analogous to the FormsAuthentication class of ASP.NET.

One important aspect of ASP.NET Identity is that the database table schema is not rigidly fixed as in the case of the ASP.NET membership system. ASP.NET Identity uses Entity Framework Code First approach to generate the table schema based on the user and role objects. That means for each piece of user profile a

separate column is created in the database table. The following figure shows the sample tables created by ASP.NET Identity in a database.



Sample Tables Created by ASP.NET Identity in a Database

As you can see, all the tables that begin with "AspNet" are created by ASP.NET Identity. Notice that the AspNetUsers table that stores user information also contains profile information such as FullName, BirthDate and Bio as separate columns.

By default a separate database is created in the App_Data folder and all the above tables are created in that database. However, you can use an existing database for storing this information. If you decide to do so, the above tables will be created in the database you specify. To accomplish this you can create a custom DbContext class that inherits from the **IdentityDbContext** base class. While there is no analogy for this part in the ASP.NET membership, recollect that you used the aspnet_regsql.exe tool to configure a database to have the required tables. You then used web.config to configure membership, role and profile providers.

Note:

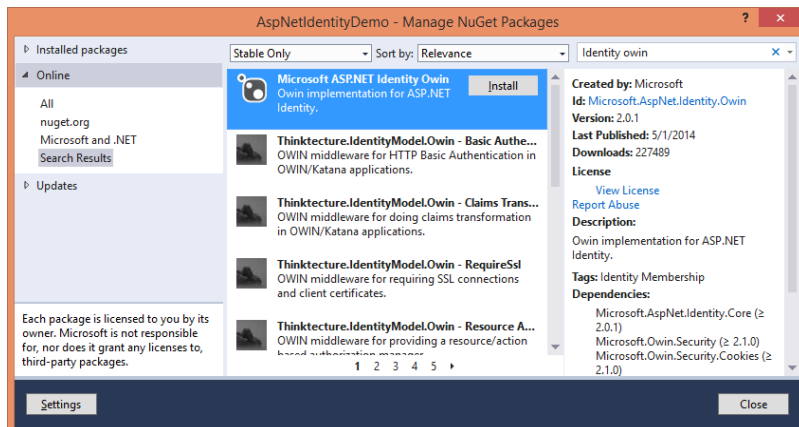
There are many interfaces that are implemented by the above mentioned classes. Since this article aims at quickly showing you how to implement security based on ASP.NET Identity from the ground up, these interfaces are not discussed here.

Using ASP.NET Identity

Now that you have some idea about ASP.NET Identity, let's build an application step-by-step that shows how to implement security using ASP.NET Identity. Begin by creating a new ASP.NET Web Application using Visual Studio 2013. Make sure to select the Empty project template and check the MVC references checkbox.

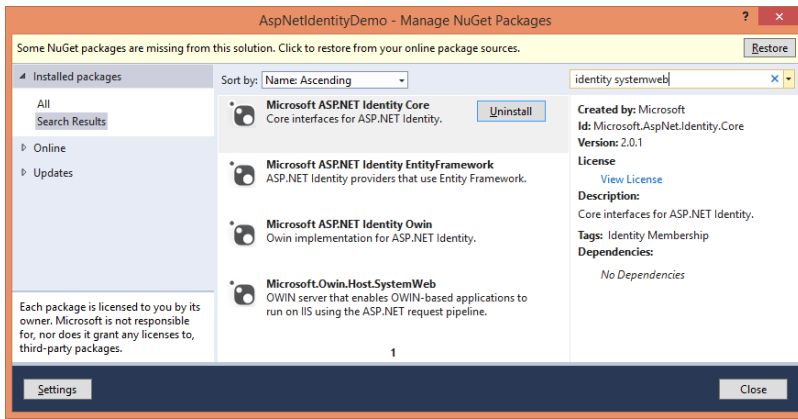
Get Required NuGet Packages

The first step is to obtain the latest NuGet packages for ASP.NET Identity. Right click on the References folder and select the Manage NuGet Packages option. Search for "ASP.NET Identity" in the Manage NuGet Packages dialog. The following figure shows the NuGet package to be added.



The NuGet Package to be Added

This will add references to various OWIN and ASP.NET Identity assemblies. You can have a look at them by expanding the References folder or by checking the Installed packages in the same dialog (see below).



Manage NuGet Packages

Creating Custom DbContext, User and Role

In this section you will create three POCOs that represent a custom DbContext, a custom user and a custom role respectively. So, add three POCOs to your application as shown below:

```

1. public class MyIdentityDbContext : IdentityDbContext<MyIdentityUser>
2. {
3.     public MyIdentityDbContext() : base("connectionstring")
4.     {
5.     }
6. }
7. }
8.
9. public class MyIdentityUser : IdentityUser
10. {
11.     public string FullName { get; set; }
12.     public DateTime BirthDate { get; set; }
13.     public string Bio { get; set; }
14. }
15.
16. public class MyIdentityRole:IdentityRole
17. {
18.     public MyIdentityRole()
19.     {
20.     }
21. }
22.
23. public MyIdentityRole(string roleName,string description) : base(roleName)
24. {
25.     this.Description = description;
26. }
27.
28.     public string Description { get; set; }
29. }

```

The MyIdentityDbContext class inherits from the IdentityDbContext base class and specifies a generic type of MyIdentityUser. This way the underlying system can determine the table schema of the tables. Notice that the MyIdentityDbContext class constructor passes a connection string name - connectionstring - to the base class. This connection string is defined in web.config as follows:

```

1. <connectionStrings>
2.     <add name="connectionstring" connectionString="data source=.;initial catalog=Northwind;integrated security=true" providerName="System.Data.SqlClient" />
3. </connectionStrings>

```

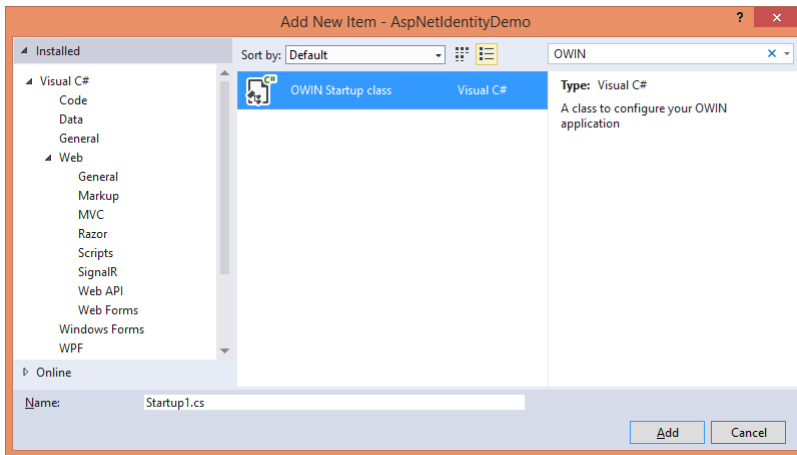
Note that the above connection string specifies the target database as Northwind so that all the required tables will be created inside the existing database (Northwind in this case).

The MyIdentityUser class inherits from the IdentityUser base class and adds three properties - FullName, BirthDate and Bio. These properties make the profile information of the user. The IdentityUser base class provides several properties such as UserName, Email and PasswordHash.

The MyIdentityRole class inherits from the IdentityRole base class and adds the Description property.

Adding an OWIN Startup Class

Now add an OWIN startup class in the App_Start folder using the OWIN Startup Class template. The following figure shows this template in the Add New Item dialog.



Add New Item

Change the name of the class file to Startup.cs and click Add. Then write the following code in the Startup class.

```

1. using Owin;
2. using Microsoft.Owin;
3. using Microsoft.Owin.Security.Cookies;
4. using Microsoft.AspNet.Identity;
5.
6. [assembly: OwinStartup(typeof(AspNetIdentityDemo.App_Start.Startup))]
7.
8. namespace AspNetIdentityDemo.App_Start
9. {
10.     public class Startup
11.     {
12.         public void Configuration(IAppBuilder app)
13.         {
14.             CookieAuthenticationOptions options = new CookieAuthenticationOptions();
15.             options.AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie;
16.             options.LoginPath = new PathString("/account/login");
17.             app.UseCookieAuthentication(options);
18.         }
19.     }
20. }

```

Ensure that the namespaces as listed above are imported before you write any code. The assembly level attribute - OwinStartup - specifies an OWIN startup class. In this case it is AspNetIdentityDemo.App_Start.Startup. The Startup class consists of Configuration() method and receives a parameter of type IAppBuilder. We won't go into the details of OWIN here. Suffice it to say that Configuration() method configures the authentication scheme. In this case the code uses cookie based authentication. The login page of the application is set to /account/login. You will create the Account controller in the following sections.

Creating View Model Classes

In this section you will create model classes need by the Account controller. In all you need four model classes - Register, Login, ChangePassword and ChangeProfile. These model classes are POCOs with some data annotations added to them. They are shown below:

```

1. public class Register
2. {
3.     [Required]
4.     public string UserName { get; set; }
5.     [Required]
6.     public string Password { get; set; }
7.     [Required]
8.     [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
9.     public string ConfirmPassword { get; set; }
10.    [Required]
11.    [EmailAddress]
12.    public string Email { get; set; }
13.    public string FullName { get; set; }
14.    public DateTime BirthDate { get; set; }
15.    public string Bio { get; set; }
16. }

1. public class Login
2. {
3.     [Required]
4.     [Display(Name = "User name")]
5.     public string UserName { get; set; }
6.     [Required]
7.     [DataType(DataType.Password)]
8.     [Display(Name = "Password")]
9.     public string Password { get; set; }
10.    [Display(Name = "Remember me?")]
11.    public bool RememberMe { get; set; }
12. }

1. public class ChangePassword
2. {
3.     [Required]
4.     public string OldPassword { get; set; }
5.     [Required]
6.     [StringLength(40, MinimumLength = 6, ErrorMessage = "Password must be between 6-40 characters.")]
7.     public string NewPassword { get; set; }
8.     [Compare("NewPassword", ErrorMessage = "The new password and confirmation password do not match.")]
9.     public string ConfirmNewPassword { get; set; }
10. }

1. public class ChangeProfile
2. {
3.     public string FullName { get; set; }

```

```

4.     public DateTime BirthDate { get; set; }
5.     public string Bio { get; set; }
6. }

```

The Register, Login, ChangePassword and ChangeProfile model classes are quite straightforward and hence we won't go into any detailed explanation. Create these classes and make sure to place them in the Models folder.

Creating AccountController

In this section you will create the AccountController - the controller where all the security related magic happens. The Account controller contains nine action methods and a constructor. The names and purpose of each of these methods is listed in the following table for your quick reference.

Method	Description
AccountController()	This is the constructor of the Account controller and initializes UserManager and RoleManager for further use.
Register()	The Register() action method returns Register view to the user.
Register(Register model)	This version of Register() is called when the user submits the registration form and does the job of creating a user account.
Login()	The Login() action method returns Login view to the user.
Login(Login model)	This version of Login() is called with the user submits the login form and does the job of authenticating a user.
ChangePassword()	The ChangePassword() method returns ChangePassword view to the user.
ChangePassword(ChangePassword model)	This version of ChangePassword() is called when the user submits the change password form and does the job of changing the user password.
ChangeProfile()	The ChangeProfile() method returns ChangeProfile view to the user.
ChangeProfile(ChangeProfile model)	This version of ChangeProfile() is called when the user submits the change profile form and does the job of changing the user profile information.
Logout()	The Logout() method is called when a logged-in user clicks Logout button from any of the views and does the job of removing the authenticating cookie.

Now that you have some idea about the AccountController, let's write these methods one by one. So, add the AccountController controller class in the Controllers folder and write its constructor as shown below:

```

1. private UserManager<MyIdentityUser> userManager;
2. private RoleManager<MyIdentityRole> roleManager;
3.
4. public AccountController()
5. {
6.     MyIdentityDbContext db = new MyIdentityDbContext();
7.
8.     UserStore<MyIdentityUser> userStore = new UserStore<MyIdentityUser>(db);
9.     userManager = new UserManager<MyIdentityUser>(userStore);
10.
11.     RoleStore<MyIdentityRole> roleStore = new RoleStore<MyIdentityRole>(db);
12.     roleManager = new RoleManager<MyIdentityRole>(roleStore);
13.
14. }

```

The above code declares two member variables inside the AccountController class - one of type UserManager and the other of type RoleManager. Notice that while declaring these variables, the generic type of MyIdentityUser and MyIdentityRole is specified.

The AccountController() constructor creates an instance of our custom DbContext class and passes it to the constructor of UserStore and RoleStore classes. The UserStore and RoleStore instances are further passed to the constructors of UserManager and RoleManager classes respectively. The UserStore and RoleStore classes basically perform database storage and retrieval tasks.

Next, add both the Register() methods as shown below:

```

1. public ActionResult Register()
2. {
3.     return View();
4. }
5.
6. [HttpPost]
7. [ValidateAntiForgeryToken]
8. public ActionResult Register(Register model)
9. {
10.    if (ModelState.IsValid)
11.    {
12.        MyIdentityUser user = new MyIdentityUser();
13.
14.        user.UserName = model.UserName;
15.        user.Email = model.Email;
16.        user.FullName = model.FullName;
17.        user.BirthDate = model.BirthDate;
18.        user.Bio = model.Bio;
19.
20.        IdentityResult result = userManager.Create(user, model.Password);
21.
22.        if (result.Succeeded)
23.        {
24.            userManager.AddToRole(user.Id, "Administrator");
25.            return RedirectToAction("Login", "Account");
26.        }
27.        else
28.        {
29.            ModelState.AddModelError("UserName", "Error while creating the user!");
30.        }
31.    }
32.    return View(model);
33. }

```

The POST version of Register() method creates an instance of MyIdentityUser class and sets its properties to the corresponding properties of the Register model class. It then attempts to create a new user account by calling Create() method on the UserManager object. If IdentityResult.Succeeded returns true, it indicates that a user account has been successfully created. If so the newly created user is assigned an Administrator role (in a more realistic situation you will have a separate user-role management page where users are assigned roles. Here, for the sake of simplicity you are assigning a role at the time of registration itself.). The user is then redirected to the login page.

If there is some error while creating a user an error message is added to the ModelState dictionary and Register view is displayed again with the error messages.

Now, add Login() action methods as shown below:

```

1. public ActionResult Login(string returnUrl)
2. {
3.     ViewBag.ReturnUrl = returnUrl;
4.     return View();
5. }
6.
7. [HttpPost]
8. [ValidateAntiForgeryToken]
9. public ActionResult Login(Login model, string returnUrl)
10. {
11.     if (ModelState.IsValid)
12.     {
13.
14.         MyIdentityUser user = userManager.Find(model.UserName, model.Password);
15.         if (user != null)
16.         {
17.             IAuthenticationManager authenticationManager = HttpContext.GetOwinContext().Authentication;
18.             authenticationManager.SignOut(DefaultAuthenticationTypes.ExternalCookie);
19.             ClaimsIdentity identity = userManager.CreateIdentity(user, DefaultAuthenticationTypes.ApplicationCookie);
20.             AuthenticationProperties props = new AuthenticationProperties();
21.             props.IsPersistent = model.RememberMe;
22.             authenticationManager.SignIn(props, identity);
23.             if (Url.IsLocalUrl(returnUrl))
24.             {
25.                 return Redirect(returnUrl);
26.             }
27.             else
28.             {
29.                 return RedirectToAction("Index", "Home");
30.             }
31.         }
32.         else
33.         {
34.             ModelState.AddModelError("", "Invalid username or password.");
35.         }
36.     }
37.     return View(model);
38. }

```

If a user tries to access a URL without logging in, he is redirected automatically to the login page. While doing so the system passes the URL of the page originally requested by the user in a returnUrl query string parameter. This URL is received in the Login() methods as returnUrl parameter. The first Login() method simply passes the returnUrl to the view so that it can be resent upon form submission.

The second Login() method validates whether user name and password are correct. This is done by finding a user with a specified user name and password using the UserManager object. If a user is found that indicates that user name and password are matching. If so, the code retrieves an authentication manager by calling HttpContext.GetOwinContext().Authentication and stores in a variable of type IAuthenticationManager. The user is logged out from any external logins by calling SignOut() method. Then the ClaimsIdentity object is created by calling CreateIdentity() method on the User Manager. The SignIn() method accepts this ClaimsIdentity and does the job of issuing an authentication cookie that indicates that a user is successfully logged in to the system. The AuthenticationProperties object specifies the IsPersistent property to indicate whether the authentication cookie should be a persistent cookie (true) or not (false).

Next, add ChangePassword() methods as shown below:

```

1. [Authorize]
2. public ActionResult ChangePassword()
3. {
4.     return View();
5. }
6.
7. [HttpPost]
8. [Authorize]
9. [ValidateAntiForgeryToken]
10. public ActionResult ChangePassword(ChangePassword model)
11. {
12.     if (ModelState.IsValid)
13.     {
14.         MyIdentityUser user = userManager.FindByName(HttpContext.User.Identity.Name);
15.         IdentityResult result = userManager.ChangePassword(user.Id, model.OldPassword, model.NewPassword);
16.         if (result.Succeeded)
17.         {
18.             IAuthenticationManager authenticationManager = HttpContext.GetOwinContext().Authentication;
19.             authenticationManager.SignOut();
20.             return RedirectToAction("Login", "Account");
21.         }
22.         else
23.         {
24.             ModelState.AddModelError("", "Error while changing the password.");
25.         }
26.     }
27.     return View(model);
28. }

```

The second ChangePassword() method retrieves the current user name using HttpContext.User.Identity.Name. Based on this value, FindByName() method of the UserManager is called to retrieve the associated MyIdentityUser. Then ChangePassword() method is called on the UserManager by passing the Id of the user, old password and the new password. If the password change operation is successful the user is signed out of the system and is redirected to the login page.

Now, add ChangeProfile() methods as shown below:

```

1. [Authorize]
2. public ActionResult ChangeProfile()

```

```

3. {
4.     MyIdentityUser user = userManager.FindByName(HttpContext.User.Identity.Name);
5.     ChangeProfile model = new ChangeProfile();
6.     model.FullName = user.FullName;
7.     model.BirthDate = user.BirthDate;
8.     model.Bio = user.Bio;
9.     return View(model);
10. }
11.
12. [HttpPost]
13. [Authorize]
14. [ValidateAntiForgeryToken]
15. public ActionResult ChangeProfile(ChangeProfile model)
16. {
17.     if (ModelState.IsValid)
18.     {
19.         MyIdentityUser user = userManager.FindByName(HttpContext.User.Identity.Name);
20.         user.FullName = model.FullName;
21.         user.BirthDate = model.BirthDate;
22.         user.Bio = model.Bio;
23.         IdentityResult result = userManager.Update(user);
24.         if(result.Succeeded)
25.         {
26.             ViewBag.Message = "Profile updated successfully.";
27.         }
28.         else
29.         {
30.             ModelState.AddModelError("", "Error while saving profile.");
31.         }
32.     }
33.     return View(model);
34. }

```

The first `ChangeProfile()` method retrieves `MyIdentityUser` using the `FindByName()` method of the `UserManager`. The current user's profile details are filled in `ChangeProfile` model and passed to the `ChangeProfile` view. The second `ChangeProfile()` method receives the modified user information. It then retrieves the current user using the `FindByName()` method of `UserManager`. The existing user information such as `FullName`, `BirthDate` and `Bio` is updated with the new information and to persist the changes, `Update()` method of `UserManager` is called.

Now add the final action method - `Logout()` - as shown below:

```

1. [HttpPost]
2. [Authorize]
3. [ValidateAntiForgeryToken]
4. public ActionResult Logout()
5. {
6.     IAuthenticationManager authenticationManager = HttpContext.GetOwinContext().Authentication;
7.     authenticationManager.SignOut();
8.     return RedirectToAction("Login", "Account");
9. }

```

The `Logout()` method retrieves the authentication manager using `HttpContext.GetOwinContext().Authentication` property and calls its `SignOut()` method. Doing so will delete the authentication cookie. The user is then redirected to the login page.

This completes the `AccountController`.

Creating Register, Login, ChangePassword and ChangeProfile Views

You will need to create four views - `Register.cshtml`, `Login.cshtml`, `ChangePassword.cshtml` and `ChangeProfile.cshtml` - for the corresponding action methods of the `AccountController`. We will not go into the details of these views here because they are quite straight forward. You can grab them from the code download of this article.

Creating HomeController

In this section you will create `HomeController` - the controller whose action methods are to be secured. So, add `HomeController` to the `Controllers` folder and modify its `Index()` action method as shown below:

```

1. [Authorize]
2. public ActionResult Index()
3. {
4.     MyIdentityDbContext db = new MyIdentityDbContext();
5.
6.     UserStore<MyIdentityUser> userStore = new UserStore<MyIdentityUser>(db);
7.     UserManager<MyIdentityUser> userManager = new UserManager<MyIdentityUser>(userStore);
8.
9.     MyIdentityUser user = userManager.FindByName(HttpContext.User.Identity.Name);
10.
11.     NorthwindEntities northwindDb = new NorthwindEntities();
12.     List<Customer> model = null;
13.
14.     if(userManager.IsInRole(user.Id, "Administrator"))
15.     {
16.         model = northwindDb.Customers.ToList();
17.     }
18.
19.     if(userManager.IsInRole(user.Id, "Operator"))
20.     {
21.         model = northwindDb.Customers.Where(c => c.Country == "USA").ToList();
22.     }
23.
24.     ViewBag.FullName = user.FullName;
25.
26.     return View(model);
27. }

```

Notice that the `Index()` action method is decorated with `[Authorize]` attribute because we wish to secure this action method. Inside, it creates `MyIdentityDbContext`, `UserStore` and `UserManager` as before. It then calls `FindByName()` method of the `UserManager` to retrieve the currently logged-in user. To check whether a user belongs to the `Administrator` role or not the `IsInRole()` method of `UserManager` is used. Based on the outcome of `IsInRole()` method all customers are retrieved or

only the customers belonging to the USA are retrieved (it is assumed that you have created the EF data model for the Customers table of the Northwind database). This data is passed to the Index view. The User's FullName is also passed to the Index view through the FullName ViewBag property.

Creating Index View

The Index view is a simple view that displays a welcome message to the user using the FullName ViewBag property. It also displays a table of Customer records. We won't discuss the Index view here. You can get it from the source code download of this article.

Creating Roles in the System

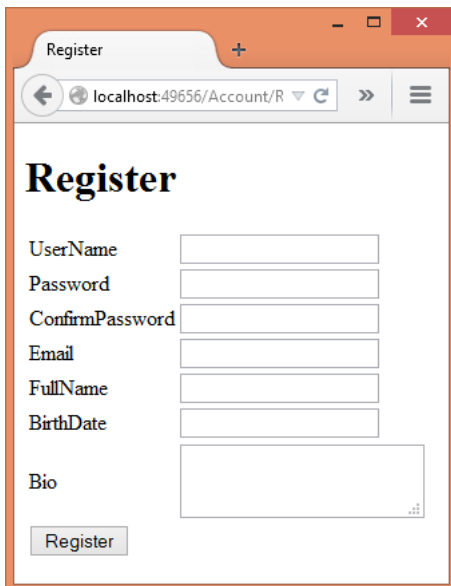
Although we used Administrator and Operator roles in our code, we haven't defined them yet. This task is to be done inside the Application_Start event of Global.asax. The following code shows how this can be done:

```
1. MyIdentityDbContext db = new MyIdentityDbContext();
2. RoleStore<MyIdentityRole> roleStore = new RoleStore<MyIdentityRole>(db);
3. RoleManager<MyIdentityRole> roleManager = new RoleManager<MyIdentityRole>(roleStore);
4.
5. if (!roleManager.RoleExists("Administrator"))
6. {
7.     MyIdentityRole newRole = new MyIdentityRole("Administrator", "Administrators can add, edit and delete data.");
8.     roleManager.Create(newRole);
9. }
10.
11. if (!roleManager.RoleExists("Operator"))
12. {
13.     MyIdentityRole newRole = new MyIdentityRole("Operator", "Operators can only add or edit data.");
14.     roleManager.Create(newRole);
15. }
```

The above code creates a RoleStore and a RoleManager. The RoleExists() method of RoleManager is used to determine if a role already exists. If the role doesn't exist, a new MyIdentityRole instance is created by passing a role name and its description. The role is created in the system by calling the Create() method of the RoleManager and passing the MyIdentityRole object as a parameter.

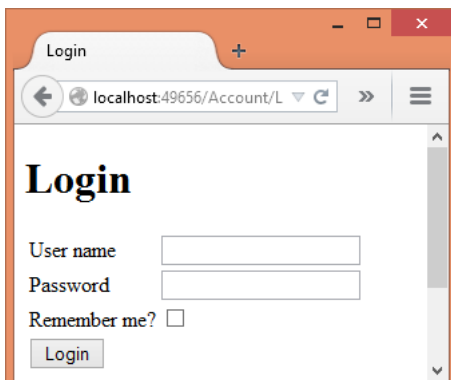
Testing the Application

Now that you have completed all the parts of the sample application, it's time to test the application. Be on the Index view and press F5 to run the application. You will find that you are automatically taken to the login page (/account/login). Click on the Register a new user link from the page so as to display the registration page as shown below:



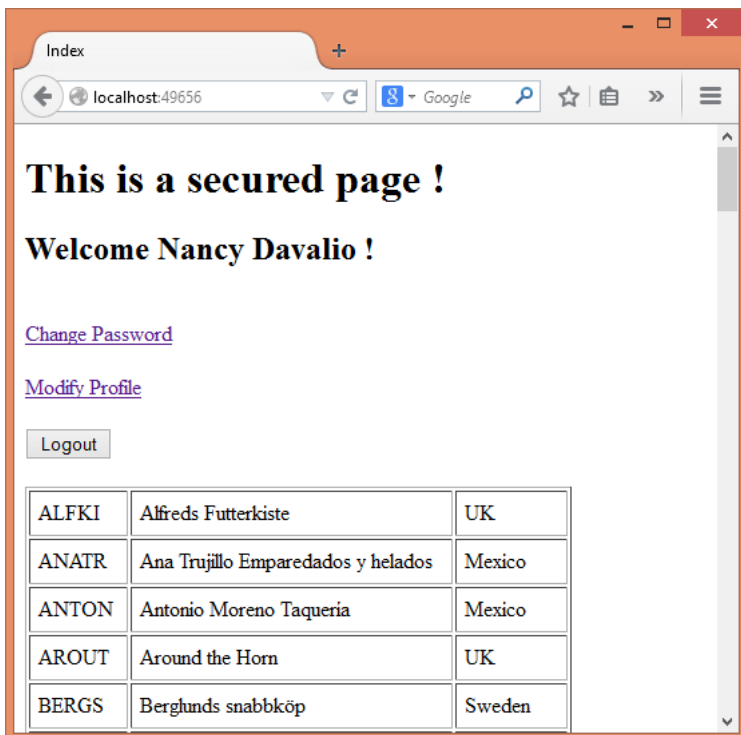
Registration Page

Enter all the registration information and click on the Register button to create a new user. After successful user creation you will be taken to the login page (see below).



Login Page

Enter user name and password and click on the Login button. You will be shown the Index view with a welcome message and Customer data from the Northwind database:



Welcome Message and Customer Data

Go ahead and also test Change Password and Modify Profile functionality.

Summary

ASP.NET Identity offers a new way of authenticating users of your web applications. It is more flexible and powerful than the ASP.NET membership system. This article showed how to implement security using ASP.NET Identity in an ASP.NET MVC application from the ground up. The example developed in this article covers user registration, login, logout, change password and change profile functionality. Once you master these basics you can go ahead and add more advanced features such as external logins and two factor authentication.

Extra Reading Links

- [Introduction to ASP.NET Identity](#)
- [MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on](#)

Related Articles

- [Preventing Cross Site Scripting Attacks in ASP.NET MVC 4](#)
- [Test Driven Development in Asp.Net MVC Architecture](#)
- [Introduction to ASP.NET vNext](#)
- [Overview of OWIN and Katana](#)

Downloads

- [AspNetIdentity-Code.zip](#)

Comments

- **Individual User Account Authentication For OAuth and Others**

Posted by *Dheeraj* on 09/27/2016 03:49pm

Hi, I was just going through your article and after that I was going through security article presented in <http://asp.net/mvc> site. Microsoft site mentions that we use the Individual User Account Authentication type of authentication to use external authentications like facebook, google, outlook others and oauth. But according to your article, it mentions that we can't use Individual User Account Authentication type. Can you please update so that it will be misleading users. Thank You, Dheeraj

[Reply](#)

- **mr**

Posted by *emmanuel* on 09/20/2016 02:37am

nice article... but please the northwind at the connectionstring and the one used in the home controller where is coming from and how do i initialize mine. am relatively new to Asp.net thanks

[Reply](#)

- **Missing *.sln file**