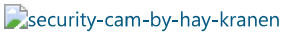
ASP.NET MVC 5 Identity: Extending and Modifying Roles

John Atten, 14 Jul 2014

4.96 (45 votes) Rate this:

In a recent article I took a rather long look at extending the ASP.NET 5 Identity model, adding some custom properties to the basic IdentityUser class, and also some basic role-based identity management. We did not discuss modifying, extending, or working directly with Roles, beyond seeding the d



In a recent article I took a rather long look at **extending the ASP.NET 5 Identity model**, adding some custom properties to the basic `IdentityUser` class, and also some basic role-based identity management. We did not discuss modifying, extending, or working directly with Roles, beyond seeding the database with some very basic roles with which to manage application access.

Extending the basic ASP.NET `IdentityRole` class, and working directly with roles from an administrative perspective, requires some careful consideration, and no small amount of work-around code-wise.

Image by Hay Kranen | Some Rights Reserved

There are two reasons this is so:

- As we saw in the previous article, the ASP.NET team made it fairly easy to extend `IdentityUser`, and also to get some basic Role management happening.
- When roles are used to enforce access restrictions within our application, they are basically hard-coded, usually via the `[Authorize]` attribute. Giving application administrators the ability to add, modify, and delete roles is of limited use if they cannot also modify the access permissions afforded by `[Authorize]`.

The above notwithstanding, sometimes we might wish to add some properties to our basic roles, such as a brief description.

In this post we will see how we can extend the `IdentityRole` class, by adding an additional property. We will also add the basic ability to create, edit, and delete roles, and what all is involved with that, despite the fact that any advantage to adding or removing roles is limited by the hard-coded `[Authorize]` permissions within our application.

In the next post, **ASP.NET MVC 5 Identity: Implementing Group-Based Permissions Management**, look at working around the limitations of the Role/[Authorize] model to create a more finely-grained role-based access control system.

If you are using the Identity 2.0 Framework:

This article focuses on customizing and modifying **version 1.0** of the ASP.NET Identity framework. If you are using the recently released version 2.0, this code in this article won't work. For more information on working with Identity 2.0, see

- ASP.NET Identity 2.0: Understanding the Basics
- ASP.NET Identity 2.0: Customizing Users and Roles
- ASP.NET Identity 2.0: Setting Up Account Validation and Two-Factor Authentication.
- ASP.NET Identity 2.0 Extending Identity Models and Using Integer Keys Instead of Strings

Many of the customizations implemented in this article are included "in the box" with the Identity Samples project. I discuss extending and customizing IdentityUser and IdentityRole in Identity 2.0 in a new article, **ASP.NET Identity 2.0: Customizing Users and Roles**

If you are using the Identity 1.0 Framework:

UPDATE: 2/24/2014 *- Thanks to Code Project user Budoray for catching some typos in the code. The* `EditRoleViewModel` *and* `RoleViewModel` *classes were referenced incorrectly in a number of places, preventing the project from building properly. Fixed!*

Getting Started - Building on Previous Work

We have laid the groundwork for what we will be doing in the previous article on **Extending Identity Accounts**, so we will clone that project and build on top of the work already done. In that project, we:

- Created a restricted, internal access MVC site.
- Removed extraneous code related to social media account log-ins, and other features we don't need.
- Extended the IdentityUser class to include some additional properties, such as first/last names, and email addresses.
- Added the ability to assign users to pre-defined roles which govern access to various functionality within our application.

Clone the Source

You can clone the `original project` and follow along, or you can grab the finished project from my Github repo. To get the original project and build along with this article, clone the source from:

- SOURCE: Role-Based Security Example

If you want to check out the `finished project`, clone the source from:

- SOURCE: Extending ASP.NET Identity Roles

First, a Little Refactoring

In the original project, I had left all of the Identity-related models in the single file created with the default project template. Before getting started here, I pulled each class out into its own code file. Also, we will be re-building our database and migrations.

After cloning the source, I deleted the existing Migration (not the Migrations folder, just the Migration file within named *201311110510410_Init.cs*. We will keep the *Migrations/Configuration.cs* file as we will be building out on that as we go.

If you are following along, note that I also **renamed the project and solution, namespaces, and such**, as I am going to push this project up to Github separately from the original.

There is plenty of room for additional cleanup in this project, but for now, it will do. Let's get started.

Getting Started

Previously, we were able to define the model class `ApplicationUser`, which extended the Identity class `IdentityUser`, run EF Migrations, and with relative ease swap it with `IdentityUser` in all the areas of our application which previously consumed `IdentityUser`. Things are not so simple, however, when it comes to extending `IdentityRole`.

`IdentityRole` forms a core component in the authorization mechanism for an ASP.NET application. For this reason, we might expect the Identity system to be resistant to casual modification of the `IdentityRole` class itself, and perhaps equally importantly, the manner in which the rest of the identity system accepts derivatives. So we need to find a way to accomplish what we wish to achieve without compromising the integrity of the Identity mechanism, or those components downstream which may depend upon an instance of `IdentityRole` to get the job done.

First off, let's take a look at our existing `ApplicationDbContext` class:

The ApplicationDbContext Class:

Hide Copy Code

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext()
        : base("DefaultConnection")
    {
    }
}
```

In the above, we can see we are inheriting from the class `IdentityDbContext<TUser>`, which allows us to specify a custom type, so long as that type is derived from `IdentityUser`. So it appears that the Identity system generally provides a built-in mechanism for extending `IdentityUser`. Is there a similar path for extending `IdentityRole`? Turns out there is. Sort of.

Extending the Identity Role Class

First, of course, we need to create our derived class, `ApplicationRole`. Add the following class to the *Models* folder:

The Application Role Class:

Hide Copy Code

```
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity;
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
namespace AspNetExtendingIdentityRoles.Models
{
    public class ApplicationRole : IdentityRole
    {
        public ApplicationRole() : base() { }
        public ApplicationRole(string name, string description) : base(name)
        {
            this.Description = description;
        }
        public virtual string Description { get; set; }
    }
}
```

As we can see, we have created a derived class and implemented a simple new `Description` property, along with a new overridden constructor. Next, we need modify our `ApplicationDbContext` so that, when we run EF Migrations, our database will reflect the proper modeling. Open the `ApplicationDbContext` class, and add the following override for the `OnModelCreating` method:

Add These Namespaces to the Top of your Code File:

Hide Copy Code

```
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity;
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using System.Data.Entity;
using System;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.ModelConfiguration;
using System.Data.Entity.ModelConfiguration.Configuration;
using System.Data.Entity.Validation;
using System.Globalization;
using System.Linq;
using System.Linq.Expressions;
using System.Reflection;
using System.Runtime.CompilerServices;
```

Then add the following Code to the `ApplicationDbContext` class:

NOTE: *This code is dense and a little messy. Don't work about understanding the details here - just take a high level view and try to grasp the overall idea of how this code is mapping our code objects to database tables. In particular, the manner in which it models the foreign key relationships between tables.*

Overriding the OnModelCreating method:

Hide Shrink Copy Code

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    if (modelBuilder == null)
    {
        throw new ArgumentNullException("modelBuilder");
    }

    // Keep this:
    modelBuilder.Entity<IdentityUser>().ToTable("AspNetUsers");

    // Change TUser to ApplicationUser everywhere else -
    // IdentityUser and ApplicationUser essentially 'share' the AspNetUsers Table in the database:
    EntityTypeConfiguration<ApplicationUser> table =
        modelBuilder.Entity<ApplicationUser>().ToTable("AspNetUsers");

    table.Property((ApplicationUser u) => u.UserName).IsRequired();

    // EF won't let us swap out IdentityUserRole for ApplicationUserRole here:
    modelBuilder.Entity<ApplicationUser>().HasMany<IdentityUserRole>((ApplicationUser u) => u.Roles);
    modelBuilder.Entity<IdentityUserRole>().HasKey((IdentityUserRole r) =>
        new { UserId = r.UserId, RoleId = r.RoleId }).ToTable("AspNetUserRoles");

    // Leave this alone:
    EntityTypeConfiguration<IdentityUserLogin> entityTypeConfiguration =
        modelBuilder.Entity<IdentityUserLogin>().HasKey((IdentityUserLogin l) =>
            new { UserId = l.UserId, LoginProvider = l.LoginProvider, ProviderKey
                = l.ProviderKey }).ToTable("AspNetUserLogins");

    entityTypeConfiguration.HasRequired<IdentityUser>((IdentityUserLogin u) => u.User);
    EntityTypeConfiguration<IdentityUserClaim> table1 =
        modelBuilder.Entity<IdentityUserClaim>().ToTable("AspNetUserClaims");

    table1.HasRequired<IdentityUser>((IdentityUserClaim u) => u.User);

    // Add this, so that IdentityRole can share a table with ApplicationRole:
    modelBuilder.Entity<IdentityRole>().ToTable("AspNetRoles");

    // Change these from IdentityRole to ApplicationRole:
    EntityTypeConfiguration<ApplicationRole> entityTypeConfiguration1 =
        modelBuilder.Entity<ApplicationRole>().ToTable("AspNetRoles");

    entityTypeConfiguration1.Property((ApplicationRole r) => r.Name).IsRequired();
}
```

In the above, we are basically telling Entity Framework how to model our inheritance structure into the database. We can, however, tell EF to model our database in such a way that both of our derived classes can also utilize the same tables, and in fact extend them to include our custom fields. Notice how, in the code above, we first tell the `modelBuilder` to point the `IdentityUser` class at the table "AspNetUsers", and then also tell it to point `ApplicationUser` at the same table? We do the same thing later with `ApplicationRole`.

As you can see, there is actually no getting away from either the `IdentityUser` or `IdentityRole` classes - both are used by the Identity system under the covers. We are simply taking advantage of polymorphism such that, at the level of our application, we are able to use our derived classes, while down below, Identity recognizes them as their base implementations. We will see how this affects our application shortly.

Update the Identity Manager Class

Now, however, we can replace `IdentityRole` with `ApplicationRole` in most of the rest of our application, and begin using our new `Description` property. We will begin with our `IdentityManager` class. I did a little refactoring here while I was at it, so if you are following along, this code will look a little different than what you will find in the original project. Just paste this code in (but make sure your namespaces match!). I also added a few new using's at the top of the code file.

Modified Identity Manager Class:

Hide Shrink Copy Code

```
public class IdentityManager
{
```

```
    // Swap ApplicationRole for IdentityRole:
    RoleManager<ApplicationRole> _roleManager = new RoleManager<ApplicationRole>(
        new RoleStore<ApplicationRole>(new ApplicationDbContext()));

    UserManager<ApplicationUser> _userManager = new UserManager<ApplicationUser>(
        new UserStore<ApplicationUser>(new ApplicationDbContext()));

    ApplicationDbContext _db = new ApplicationDbContext();


    public bool RoleExists(string name)
    {
        return _roleManager.RoleExists(name);
    }


    public bool CreateRole(string name, string description = "")
    {
        // Swap ApplicationRole for IdentityRole:
        var idResult = _roleManager.Create(new ApplicationRole(name, description));
        return idResult.Succeeded;
    }


    public bool CreateUser(ApplicationUser user, string password)
    {
        var idResult = _userManager.Create(user, password);
        return idResult.Succeeded;
    }


    public bool AddUserToRole(string userId, string roleName)
    {
        var idResult = _userManager.AddToRole(userId, roleName);
        return idResult.Succeeded;
    }


    public void ClearUserRoles(string userId)
    {
        var user = _userManager.FindById(userId);
        var currentRoles = new List<IdentityUserRole>();

        currentRoles.AddRange(user.Roles);
        foreach (var role in currentRoles)
        {
            _userManager.RemoveFromRole(userId, role.Role.Name);
        }
    }
}
```

Update the Add Users and Roles Method in Migrations Configuration

We will also want to update our `AddUsersAndRoles()` method, which is called by the `Seed()` method in the Configuration file for EF Migrations. We want to seed the database with Roles which use our new extended properties:

The Updated Add Users and Roles Method:

Hide Shrink Copy Code

```
bool AddUserAndRoles()
{
    bool success = false;
    var idManager = new IdentityManager();

    // Add the Description as an argument:
    success = idManager.CreateRole("Admin", "Global Access");
    if (!success == true) return success;

    // Add the Description as an argument:
    success = idManager.CreateRole("CanEdit", "Edit existing records");
    if (!success == true) return success;

    // Add the Description as an argument:
    success = idManager.CreateRole("User", "Restricted to business domain activity");
    if (!success) return success;

    // While you're at it, change this to your own log-in:
    var newUser = new ApplicationUser()
    {
        UserName = "jatten",
        FirstName = "John",
        LastName = "Atten",
        Email = "jatten@typecastexception.com"
    };

    // Be careful here - you  will need to use a password which will
    // be valid under the password rules for the application,
    // or the process will abort:
    success = idManager.CreateUser(newUser, "Password1");
    if (!success) return success;
```

```
    success = idManager.AddUserToRole(newUser.Id, "Admin");
    if (!success) return success;

    success = idManager.AddUserToRole(newUser.Id, "CanEdit");
    if (!success) return success;

    success = idManager.AddUserToRole(newUser.Id, "User");
    if (!success) return success;

    return success;
}
```

All we really did here was pass an additional argument to the `CreateRole()` method, such the the seed roles will exhibit our new property. Now, we need to make a few adjustments to our `AccountController`, View Models, and Views.

Update the Select Role Editor View Model

In the previous article, we created a `SelectRoleEditorViewModel` which accepted an instance of `IdentityRole` as a constructor argument. We need to modify the code here in order to accommodate any new properties we added when we extended `IdentityRole`. For our example, we just added a single new property, so this is pretty painless:

The Modified SelectRoleEditorViewModel:

Hide Copy Code

```
public class SelectRoleEditorViewModel
{
    public SelectRoleEditorViewModel() { }

    // Update this to accept an argument of type ApplicationRole:
    public SelectRoleEditorViewModel(ApplicationRole role)
    {
        this.RoleName = role.Name;

        // Assign the new Descrption property:
        this.Description = role.Description;
    }


    public bool Selected { get; set; }

    [Required]
    public string RoleName { get; set; }

    // Add the new Description property:
    public string Description { get; set; }
}
```

Update the Corresponding Editor View Model

Recall that, in order to **display the list of roles with checkboxes** as an HTML form from which we can return the selection choices made by the user, we needed to define an `EditorViewModel.cshtml` file which corresponded to our EditorViewModel class. In *Views/Shared/EditorViewModels* open `SelectRoleEditorViewModel.cshtml` and make the following changes:

The Modified SelectRoleEditorViewModel:

Hide Copy Code

```
@model AspNetExtendingIdentityRoles.Models.SelectRoleEditorViewModel
@Html.HiddenFor(model => model.RoleName)
<tr>
    <td style="text-align:center">
        @Html.CheckBoxFor(model => model.Selected)
    </td>
    <td style="padding-right:20px">
        @Html.DisplayFor(model => model.RoleName)
    </td>
    <td style="padding-right:20px">
        @Html.DisplayFor(model => model.Description)
    </td>
</tr>
```

Again, all we needed to do in the above was add a table data element for the new property.

Update the User Roles View

To this point, we actually only have one View which displays our Roles - the `UserRoles` view, where we assign users to one or more Roles within our application. Once again, we really just need to add a table Header element to represent our new `Description` property: The Updated `UserRoles` View:

Hide Shrink Copy Code

```
@model AspNetExtendingIdentityRoles.Models.SelectUserRolesViewModel

@{
    ViewBag.Title = "User Roles";
}
<h2>Roles for user @Html.DisplayFor(model => model.UserName)</h2>
<hr />
```

```
@using (Html.BeginForm("UserRoles", "Account", FormMethod.Post, new { encType = "multipart/form-data", name = "myform" }))
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        @Html.ValidationSummary(true)
        <div class="form-group">
            <div class="col-md-10">
                @Html.HiddenFor(model => model.UserName)
            </div>
        </div>

        <h4>Select Role Assignments</h4>
        <br />
        <hr />

        <table>
            <tr>
                <th>
                    Select
                </th>
                <th>
                    Role
                </th>
                <th>
                    Description
                </th>
            </tr>

            @Html.EditorFor(model => model.Roles)
        </table>
        <br />
        <hr />

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Do We Really Want to Edit Roles?

Here is where you may want to think things through a little bit. Consider **- the main reason for utilizing Roles within an ASP.NET application is to manage authorization**. We do this by hard-coding access permissions to controllers and/or specific methods through the `[Authorize]` attribute. If we make roles createable/editable/deletable, what can possibly go wrong? Well, a lot. First off, if we deploy our application, then add a role, we really have no way to make use of the new role with respect to the security concerns listed above, without re-compiling and re-deploying our application after adding the new role to whichever methods we hope to secure with it.

The same is true if we change a role name, or worse, delete a role. In fact, an ambitious admin user could lock themselves out of the application altogether! However, there are some cases where we might want to have this functionality anyway. If, for example, you are the developer, and you add some new role permissions via `[Authorize]` to an existing application, it is more convenient to add the new roles to the database through the front-end than by either manually adding to the database, or re-seeding the application using Migrations. Also, if your application is in production, then re-running migrations really isn't an option anyway. In any case, a number of commentors on my previous post expressed the desire to be able to modify or remove roles, so let's plow ahead!

Why Not? Adding the Roles Controller

Once I had everything built out, and I went to use Visual Studio's built-in scaffolding to add a new Roles controller, I ran into some issues. Apparently, by extending `IdentityRole` the way we have, Entity Framework has some trouble scaffolding up a new controller based on the `ApplicationRole` model (EF detects some ambiguity between `IdentityRole` and `ApplicationRole`). I didn't spend too much time wrestling with this - it was easy enough to code up a simple CRUD controller the old-fashioned way, so that's what I did. Here is my hand-rolled `RolesController`, ready for action:

The Roles Controller:

Hide Shrink Copy Code

```
using AspNetExtendingIdentityRoles.Models;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Net;
using System.Web.Mvc;

namespace AspNetExtendingIdentityRoles.Controllers
{
    public class RolesController : Controller
    {
        private ApplicationDbContext _db = new ApplicationDbContext();

        public ActionResult Index()
        {
            var rolesList = new List<RoleViewModel>();
            foreach(var role in _db.Roles)
            {
                var roleModel = new RoleViewModel(role);
```

```
            rolesList.Add(roleModel);
        }
        return View(rolesList);
    }


    [Authorize(Roles = "Admin")]
    public ActionResult Create(string message = "")
    {
        ViewBag.Message = message;
        return View();
    }


    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult Create([Bind(Include =
        "RoleName,Description")]RoleViewModel model)
    {
        string message = "That role name has already been used";
        if (ModelState.IsValid)
        {
            var role = new ApplicationRole(model.RoleName, model.Description);
            var idManager = new IdentityManager();

            if(idManager.RoleExists(model.RoleName))
            {
                return View(message);
            }
            else
            {
                idManager.CreateRole(model.RoleName, model.Description);
                return RedirectToAction("Index", "Account");
            }
        }
        return View();
    }


    [Authorize(Roles = "Admin")]
    public ActionResult Edit(string id)
    {
        // It's actually the Role.Name tucked into the id param:
        var role = _db.Roles.First(r => r.Name == id);
        var roleModel = new EditRoleViewModel(role);
        return View(roleModel);
    }


    [HttpPost]
    [Authorize(Roles = "Admin")]
    public ActionResult Edit([Bind(Include =
        "RoleName,OriginalRoleName,Description")] EditRoleViewModel model)
    {
        if (ModelState.IsValid)
        {
            var role = _db.Roles.First(r => r.Name == model.OriginalRoleName);
            role.Name = model.RoleName;
            role.Description = model.Description;
            _db.Entry(role).State = EntityState.Modified;
            _db.SaveChanges();
            return RedirectToAction("Index");
        }
        return View(model);
    }


    [Authorize(Roles = "Admin")]
    public ActionResult Delete(string id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        var role = _db.Roles.First(r => r.Name == id);
        var model = new RoleViewModel(role);
        if (role == null)
        {
            return HttpNotFound();
        }
        return View(model);
    }


    [Authorize(Roles = "Admin")]
    [HttpPost, ActionName("Delete")]
    public ActionResult DeleteConfirmed(string id)
    {
        var role = _db.Roles.First(r => r.Name == id);
        var idManager = new IdentityManager();
        idManager.DeleteRole(role.Id);
        return RedirectToAction("Index");
    }
  }
}
```

In the above, notice that when we go to delete a role, we make a call out to our `IdentityManager` class to a method named `DeleteRole()`. Why the complexity, John? Why not just delete the role from the datastore directly? There's a reason for that . . .

About Deleting Roles

Think about it. If you have one or more users assigned to a role, **when you delete that role, you want to remove the users from the role first**. Otherwise you will run into **foreign key issues in your database** which will not let you delete the role. So, we need to add a couple important methods to `IdentityManager`.

Adding a Delete Role Method to Identity Manager

Clearly, in order to delete roles, we first need to remove any users from that role first. Then we can delete the role. However, we have to employ a slight hack to do this, because the Identity framework does not actually implement a `RemoveRole()` method out of the box. Oh, it's there - you can find it if you look hard enough. The `RoleStore<IRole>` class actually defines a `DeleteAsync` method. However, it throws a "Not Implemented" exception. Here is how I worked around the issue. Add the following two methods to the `IdentityManager` class:

Adding DeleteRole and RemoveFromRole Methods to Identity Manager Class:

Hide Copy Code

```
public void RemoveFromRole(string userId, string roleName)
{
    _userManager.RemoveFromRole(userId, roleName);
}


public void DeleteRole(string roleId)
{
    var roleUsers = _db.Users.Where(u => u.Roles.Any(r => r.RoleId == roleId));
    var role = _db.Roles.Find(roleId);

    foreach (var user in roleUsers)
    {
        this.RemoveFromRole(user.Id, role.Name);
    }
    _db.Roles.Remove(role);
    _db.SaveChanges();
}
```

First, notice how we pass in a simple `RoleId` instead of an instance or `ApplicationRole`? This is because, in order for the `Remove(role)` method to operate properly, the role passed in as an argument must be from the same `ApplicationDbContext` instance, which would not be the case if we were to pass one in from our controller. Also notice, before calling `_db.Roles.Remove(role)` , we retrieve the collection of users who are role members, and remove them. This solves the foreign key relationship problem, and prevents orphan records in the `AspNetUserRoles` table in our database.

ViewModels and Views for the Roles Controller

Notice in our controller, we make use of two new View Models, the `RoleViewModel`, and the `EditRoleViewModel`. I went ahead and added these to the *AccountViewModels.cs* file. The code is as follows:

The RoleViewModel and EditRoleViewModel Classes:

Hide Copy Code

```
public class RoleViewModel
{
    public string RoleName { get; set; }
    public string Description { get; set; }

    public RoleViewModel() { }
    public RoleViewModel(ApplicationRole role)
    {
        this.RoleName = role.Name;
        this.Description = role.Description;
    }
}


public class EditRoleViewModel
{
    public string OriginalRoleName { get; set; }
    public string RoleName { get; set; }
    public string Description { get; set; }

    public EditRoleViewModel() { }
    public EditRoleViewModel(ApplicationRole role)
    {
        this.OriginalRoleName = role.Name;
        this.RoleName = role.Name;
        this.Description = role.Description;
    }
}
```

Views Used by the Role Controller

The Views used by the `RolesController` were created by simply right-clicking on the associated Controller method and selecting "Add View." I'm including it here for completeness, but there is nothing revolutionary going on here. The code for each follows.

The Index Role View

Displays a list of the Roles, along with Action Links to `Edit` or `Delete`.

Code for the Create Role View:

Hide Shrink Copy Code

```
@model IEnumerable<AspNetExtendingIdentityRoles.Models.RoleViewModel>

@{
    ViewBag.Title = "Application Roles";
}

<h2>Application Roles</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.RoleName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Description)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.RoleName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Description)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id = item.RoleName }) |
            @Html.ActionLink("Delete", "Delete", new { id = item.RoleName })
        </td>
    </tr>
}

</table>
```

The Create Role View

Obviously, affords creation of new Roles.

Code for the Create Role View:

Hide Shrink Copy Code

```
@model AspNetExtendingIdentityRoles.Models.RoleViewModel

@{
    ViewBag.Title = "Create";
}

<h2>Create Role</h2>


@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>RoleViewModel</h4>
        <hr />
        @Html.ValidationSummary(true)

        @if(ViewBag.Message != "")
        {
            <p style="color: red">ViewBag.Message</p>
        }
        <div class="form-group">
            @Html.LabelFor(model => model.RoleName,
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.RoleName)
                @Html.ValidationMessageFor(model => model.RoleName)
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Description,
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Description)
```

```
                @Html.ValidationMessageFor(model => model.Description)
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

The Edit Roles View

For, um, editing Roles . ..

Code for the Edit Roles View:

Hide Shrink Copy Code

```
@model AspNetExtendingIdentityRoles.Models.EditRoleViewModel

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>


@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>EditRoleViewModel</h4>
        <hr />
        @Html.ValidationSummary(true)

        @*Hide the original name away for later:*@
        @Html.HiddenFor(model => model.OriginalRoleName)

        <div class="form-group">
            @Html.LabelFor(model => model.RoleName,
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.RoleName)
                @Html.ValidationMessageFor(model => model.RoleName)
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Description,
                new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Description)
                @Html.ValidationMessageFor(model => model.Description)
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

The Delete Roles View

Code for the Delete Roles View:

Hide Shrink Copy Code

```
@model AspNetExtendingIdentityRoles.Models.RoleViewModel

@{
    ViewBag.Title = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>RoleViewModel</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.RoleName)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.RoleName)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.Description)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Description)
        </dd>

    </dl>

    @using (Html.BeginForm()) {
        @Html.AntiForgeryToken()

        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            @Html.ActionLink("Back to List", "Index")
        </div>
    }
</div>
```

Modify _Layout.cshtml to Add Users and Roles Links

I went ahead and modified the *_Layout.cshtml* file, changed what was the "Admin" link to simply "Users," and added a new "Roles" link as follows:

Modify _Layout.cshtml:

Hide Copy Code

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li>@Html.ActionLink("Home", "Index", "Home")</li>
        <li>@Html.ActionLink("About", "About", "Home")</li>
        <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
        <li>@Html.ActionLink("Users", "Index", "Account")</li>
        <li>@Html.ActionLink("Roles", "Index", "Roles")</li>
    </ul>
    @Html.Partial("_LoginPartial")
</div>
```

Run Migrations and Build out the Database

Ok, you should now be able to add a new migration, run it, and build out the database. If you are new to EF Migrations, you may want to review **Configuring Entity Framework Migrations.** In this case, the cloned project already has migrations enabled, so all we need to do is Build, then type into the Package Manager Console:

Add New Migration (Delete the previous Migration file, or choose a new name)

Hide Copy Code

```
Add-Migration init
```

Then, if all went well with that,

Update The Database:

Hide Copy Code

```
Update-Database
```

Running the Application

If all went well (and I haven't missed anything in this post!) we should be able to run our application, log in, and navigate to the "Users" tab. Then, select the "Roles" link of the single user listed. You should see something similar to this:

The User Roles View: user-roles-view

We can see, our new property is evident.

Some Thoughts in Closing

This was a rather long article to implement some relatively minor functionality. However, we touched on some concepts which may prove helpful if you need just a little more from the new Identity system than is available straight out of the box. Also, I am setting the stage for the next article, where I will look at setting up "Role Groups" to which users may be assigned. In this scenario, we can use the built-in Roles to set pretty granular access permissions within our code, and then assign users to predefined "Groups" of such Role permissions, somewhat mimicking familiar domain permissions.

Pay Attention When Messing with Auth and Security!

I do my best when working with membership or Identity to stay within the bounds and mechanisms set up by the ASP.NET team. **I am far, far from a security expert** (I DID buy a book on it recently, though!), and those people know way more about creating a secure authorization system than I could ever hope to. In the preceding article on extending `IdentityUser` and adding roles to our application, we stayed well within the bounds of the intended uses of Identity.

In this article, I ventured a little further afield, extending a class which it appears the ASP.NET team did not intend to be easily extended. Further, I implemented a means to create, edit, and delete roles at the application user level. I assume there are reasons such functionality was not built-in out of the box. Most likely for the reasons I discussed previously.

When your security and authorization needs become sufficiently complex, it may be time to examine more robust alternatives to the identity system. After all, the Identity system was designed with the primary objective of securing a public-facing web application, with a limited number of user roles and administrative requirements. Additionally, the NET team has recently released the ASP.NET Identity 2 Preview, which holds additional interesting developments.

That said, near as I can tell we have done nothing here to explicitly compromise the security of our application, or the integrity of the identity system. Our `ApplicationRole` class, through inheritance and polymorphism, is consumed properly by the internals of the ASP.NET Identity system, while delivering whatever additional properties we required.

Got Any Thoughts? See Some Improvements?

If you see where I have something wrong, or missed something while moving the code into this article, please do let me know in the comments, or shoot me an email at the address in the "About the Author" sidebar. If you have suggestions for improving on what you see here, please let me know, or submit a Pull Request on Github. I would love to incorporate any good ideas. Watch for the next article on implementing groups and permissions!

Additional Resources and Items of Interest

Identity v2.0:

- ASP.NET MVC and Identity 2.0: Understanding the Basics
- ASP.NET Identity 2.0: Setting Up Account Validation and Two-Factor Authorization
- ASP.NET Identity 2.0: Customizing Users and Roles

Identity 1.0 and Other Items:

- Source Code for this Article on Github
- Extending Identity Accounts and Implementing Role-Based Authentication in ASP.NET MVC 5
- Configuring Db Connection and Code-First Migration for Identity Accounts in ASP.NET MVC 5 and Visual Studio 2013
- ASP.NET MVC Display an HTML Table with Checkboxes to Select Row Items
- Customizing Routes in ASP.NET MVC
- Routing Basics in ASP.NET MVC

John on GoogleCodeProject

License

Share

About the Author

John Atten
Software Developer XIV Solutions
United States

My name is John Atten, and my username on many of my online accounts is xivSolutions. I am Fascinated by all things technology and software development. I work mostly with C#, Javascript/Node.js, Various flavors of databases, and anything else I find interesting. I am always looking for new information, and value your feedback (especially where I got something wrong!)

You may also be interested in...

- Building Reactive Apps
- SAPrefs - Netscape-like Preferences Dialog
- Extending Identity Accounts and Implementing Role-Based Authentication in ASP.NET MVC 5
- Window Tabs (WndTabs) Add-In for DevStudio
- ASP.NET MVC 5: Extending ASP.NET Identity 2.0 Roles and Implementation of Role Based Authorization
- To Heap or not to Heap; That's the Large Object Question?

Comments and Discussions

You must Sign In to use this message board.

Search Comments

First Prev Next

Extending Identity to grant permission on action dynamically by extending authorize attribute
diggudg **12-Jul-15 8:48**

Re: Extending Identity to grant permission on action dynamically by extending authorize attribute
longnights 9-Jan-17 16:59

Working with existing user-object
jakobvjensen **1-Jul-14 10:08**

Re: Working with existing user-object
John Atten 1-Jul-14 13:09

Odd error for Edit user
Terri Morgan **12-May-14 21:20**

Re: Odd error for Edit user
John Atten 13-May-14 2:41

Re: Odd error for Edit user
Terri Morgan 14-May-14 10:44

Re: Odd error for Edit user
Terri Morgan 17-May-14 10:13

gives error at Identity version 2
Mohammad Jamshid Shafiee **11-May-14 21:53**

Re: gives error at Identity version 2
John Atten 12-May-14 2:50

Re: gives error at Identity version 2
jonmjones 10-Jun-14 7:25

Setup directly to existing DB ?? not working
Terri Morgan **11-May-14 19:36**

Re: Setup directly to existing DB ?? not working
John Atten 12-May-14 2:47

Re: Setup directly to existing DB ?? not working
Terri Morgan 12-May-14 10:27

Getting NullReferenceException
AdamW78 **4-Apr-14 10:03**

Re: Getting NullReferenceException
John Atten 4-Apr-14 10:11

Re: Getting NullReferenceException
AdamW78 4-Apr-14 10:40

Question regarding External logins
Yann BURY **4-Apr-14 7:17**

Re: Question regarding External logins
John Atten 4-Apr-14 7:29

Re: Question regarding External logins
Yann BURY 8-Apr-14 4:59

My vote of 5
Hassan Oumar Mahamat **10-Mar-14 1:03**

Extending ApplicationUserRole
siddhant4u **26-Feb-14 1:26**

Re: Extending ApplicationUserRole
John Atten 26-Feb-14 3:04

Re: Extending ApplicationUserRole
siddhant4u 26-Feb-14 4:15

Re: Extending ApplicationUserRole
John Atten 26-Feb-14 4:23

Refresh

1 2 Next »

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.