

ASP.NET

ASP.NET Identity 2.0: Implementing Group-Based Permissions Management




 JOHN ATTEN  AUGUST 10, 2014  24

Image by Thomas Hawk | [Some Rights Reserved](#)

Earlier this year we looked at [Implementing Group-Based Permissions Management](#) using the ASP.NET Identity 1.0 framework. The objective of that project was to gain a little more granular control of application authorization, by treating the now-familiar Identity Role as more of a “permission” which could be

granted to members of a group.

With the release of Identity 2.0 in March 2014 came some breaking changes, and a significant expansion of features and complexity. Identity 2.0 is now a full-grown authorization and authentication system for ASP.NET. However, the added features came at a price. There is a lot to get your mind around to start using Identity 2.0 effectively.

In previous articles, we have covered some of the basics:

- [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](#)
- [ASP.NET Identity 2.0 Extending Identity Models and Using Integer Keys Instead of Strings](#)
- [ASP.NET Identity 2.0: Extensible Template Projects](#)

The code we used to implement Group-based permissions under Identity 1.0 breaks in moving to Identity 2.0. Simply too much has changed between the two versions to make a clean, easy upgrade. In this article, we will revisit the Group-Based Permission idea, and implement the concept under the Identity 2.0 framework.

- [Getting Started – Clone a Handy Project Template](#)
- [Adding the Group Models](#)
- [Override OnModelCreating in the DbContext Class](#)
- [Investigation: Building a Consistent Asynchronous Model Architecture](#)
- [Mimicking EntityStore – Building the GroupStoreBase Class](#)
- [Building the Primary ApplicationGroupStore Class](#)
- [Managing Complex Relationships – The ApplicationGroupManager Class](#)
- [Adding a GroupViewModel](#)
- [Update the EditUserViewModel](#)
- [Building the GroupsAdminController](#)
- [Modify the UsersAdminController](#)
- [Adding Views for the GroupsAdminController](#)
- [Modify the Identity.Config File and the DbInitializer](#)
- [Some Thoughts on Authorization and Security](#)

A Quick Review

Under the familiar ASP.NET Identity structure, Users are assigned one or more Roles. Traditionally, access to certain application functionality is governed, through the use of the [Authorize] attribute, by restricting access to certain

controllers or controller methods to members of certain roles. This is effective as far as it goes, but does not lend itself to efficient management of more granular access permissions.

The Group-Based Permissions project attempts to find a middle ground between the complexities of a full-blown, Claims-based authorization scheme, and the simplicity (and limitations) of the authorization offered by Identity out of the box. Here, we will implement another familiar idea – Users are assigned to one or more Groups. Groups are granted set of permissions corresponding to the various authorizations required by group members to perform their function.

I discussed the overall concept, and security concerns in the previous article, so I won't rehash all that here. For reference, the following topics might be worth a quick visit:

- [Granular Management of Authorization Permissions – The Principle of Least Privilege](#)
- [Some Thoughts About Authorization Management and Your Website](#)
- [Limitations of Application Authorization Under Identity](#)

In this article, we will implement a similar structure using Identity 2.0.

Getting Started – Clone a Handy Project Template

We're going to start with a [handy, ready-to-customize project template](#) based on the Identity Samples project created by the Identity team. I've used what we've learned in the past few posts to create a ready to extend project which can be cloned from my Github account (and hopefully soon, from Nuget!).

- [Easily Extensible Project Template on Github](#)

Or, Clone the Finished Source for this Project

Or, if you like, you can clone the source for the completed Group Permissions project, also on Github:

- [Group Permissions Project – Full Source on Github](#)

If you are starting with the template project and following along, it's probably best to [rename the directory and project files](#) to reflect the current effort. if you do this, make sure to also update the Web.Config file as well. You may want to update the connection string so that when the back-end database is created, the name reflects the Group Permissions application.

You MUST update the *appSettings* => `owin:AppStartup` element so that the startup assembly name matches the name you provide in *Project* => *Properties* => *Assembly Name*. In my case, I set my connection string and `owin:appStartup` as

follows:

Update Web.Config Connection String and owin:appStartup elements:

```
<connectionStrings>
  <add name="DefaultConnection"
    connectionString="Data Source=(LocalDb)\v11.0;
    Initial Catalog=AspNetIdentity2GroupPermissions-5;
    Integrated Security=SSPI"
    providerName="System.Data.SqlClient" />
</connectionStrings>
<appSettings>
  <add key="owin:AppStartup" value="IdentitySample.Startup,AspNetIdentitySample" />
  <add key="webpages:Version" value="3.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings></configuration>
```

Adding the Group Models

Let's get right to it. We can start by adding some new models to the *Models => IdentityModels.cs* file. We will be defining three different classes here:

ApplicationGroup, which is the core Group model, as well as two additional classes which map **ApplicationUser** and **ApplicationRole** as collections within **ApplicationGroup**. Add the following to the *IdentityModels.cs* file:

The ApplicationGroup and Related Classes:

```
public class ApplicationGroup
```

```
{
    public ApplicationGroup()
    {
        this.Id = Guid.NewGuid().ToString();
        this.ApplicationRoles = new List<ApplicationGroupRole>();
        this.ApplicationUsers = new List<ApplicationUserGroup>();
    }

    public ApplicationGroup(string name)
        : this()
    {
        this.Name = name;
    }

    public ApplicationGroup(string name, string description)
        : this(name)
    {
        this.Description = description;
    }

    [Key]
    public string Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public virtual ICollection<ApplicationGroupRole> ApplicationRoles { get; }
    public virtual ICollection<ApplicationUserGroup> ApplicationUsers { get; }
}

public class ApplicationUserGroup
{
    public string ApplicationUserId { get; set; }
    public string ApplicationGroupId { get; set; }
}

public class ApplicationGroupRole
{
    public string ApplicationGroupId { get; set; }
    public string ApplicationRoleId { get; set; }
}
```

```
}
```

Groups have a many-to-many relationship with both Users, and Roles. One user can belong to zero or more Groups, and one group can have zero or more users. Likewise with roles. A single role can be assigned to zero or more groups, and a single Group can have zero or more roles.

When we need to manage this type of relationship using EntityFramework, we can create what are essentially mapping classes, in this case, `ApplicationUserGroup`, and `ApplicationGroupRole`. The way we are doing this is similar to the structure used by the Identity team in defining Users, Roles, and UserRoles. For example, our `ApplicationUser` class is derived from `IdentityUser`, which defines a Roles property. Note that the Roles property of `IdentityUser` is not a collection of `IdentityRole` object, but instead a collection of `IdentityUserRole` objects. The difference being, the `IdentityUserRole` class defines only a `UserId` property, and a `RoleId` property.

We are doing the same thing here. We need to allow EF to manage the many-to-many relationships we described above by adding mapping classes between the collections defined on `ApplicationGroup`, and the domain objects involved in each relationship.

Override OnModelCreating in the DbContext Class

EntityFramework will not figure out our many-to-many relationships on its own, nor will it determine the proper table structures to create in our database. We need to help things along by overriding the `OnModelCreating` method in the `ApplicationDbContext` class. Also, we need to add `ApplicationGroups` as a property on our `DbContext` so that we can access our Groups from within our application. Update the `ApplicationDbContext` class as follows:

Update ApplicationDbContext and Override OnModelCreating:

```
public class ApplicationDbContext
    : IdentityDbContext<ApplicationUser, ApplicationRole,
    string, ApplicationUserLogin, ApplicationUserRole, ApplicationUserClaim>
{
    public ApplicationDbContext()
        : base("DefaultConnection")
    {
    }

    static ApplicationDbContext()
    {
        Database.SetInitializer<ApplicationDbContext>(new ApplicationDbInitializer());
    }

    public static ApplicationDbContext Create()
    {
        return new ApplicationDbContext();
    }

    // Add the ApplicationGroups property:
    public virtual IDbSet<ApplicationGroup> ApplicationGroups { get; set; }

    // Override OnModelsCreating:
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
    }
```

```
// Make sure to call the base method first:
base.OnModelCreating(modelBuilder);

// Map Users to Groups:
modelBuilder.Entity<ApplicationGroup>()
    .HasMany<ApplicationUserGroup>((ApplicationGroup g) => g.Applic
    .WithRequired()
    .HasForeignKey<string>((ApplicationUserGroup ag) => ag.Applica
modelBuilder.Entity<ApplicationUserGroup>()
    .HasKey((ApplicationUserGroup r) =>
        new
        {
            ApplicationUserId = r.ApplicationUserId,
            ApplicationGroupId = r.ApplicationGroupId
        }).ToTable("ApplicationUserGroups");

// Map Roles to Groups:
modelBuilder.Entity<ApplicationGroup>()
    .HasMany<ApplicationGroupRole>((ApplicationGroup g) => g.Applic
    .WithRequired()
    .HasForeignKey<string>((ApplicationGroupRole ap) => ap.Applica
modelBuilder.Entity<ApplicationGroupRole>().HasKey((ApplicationGro
    new
    {
        ApplicationRoleId = gr.ApplicationRoleId,
        ApplicationGroupId = gr.ApplicationGroupId
    }).ToTable("ApplicationGroupRoles");
}
}
```

With these humble beginnings in place, let's run the project, and see if everything works the way we expect.

Running the Project and Confirming Database

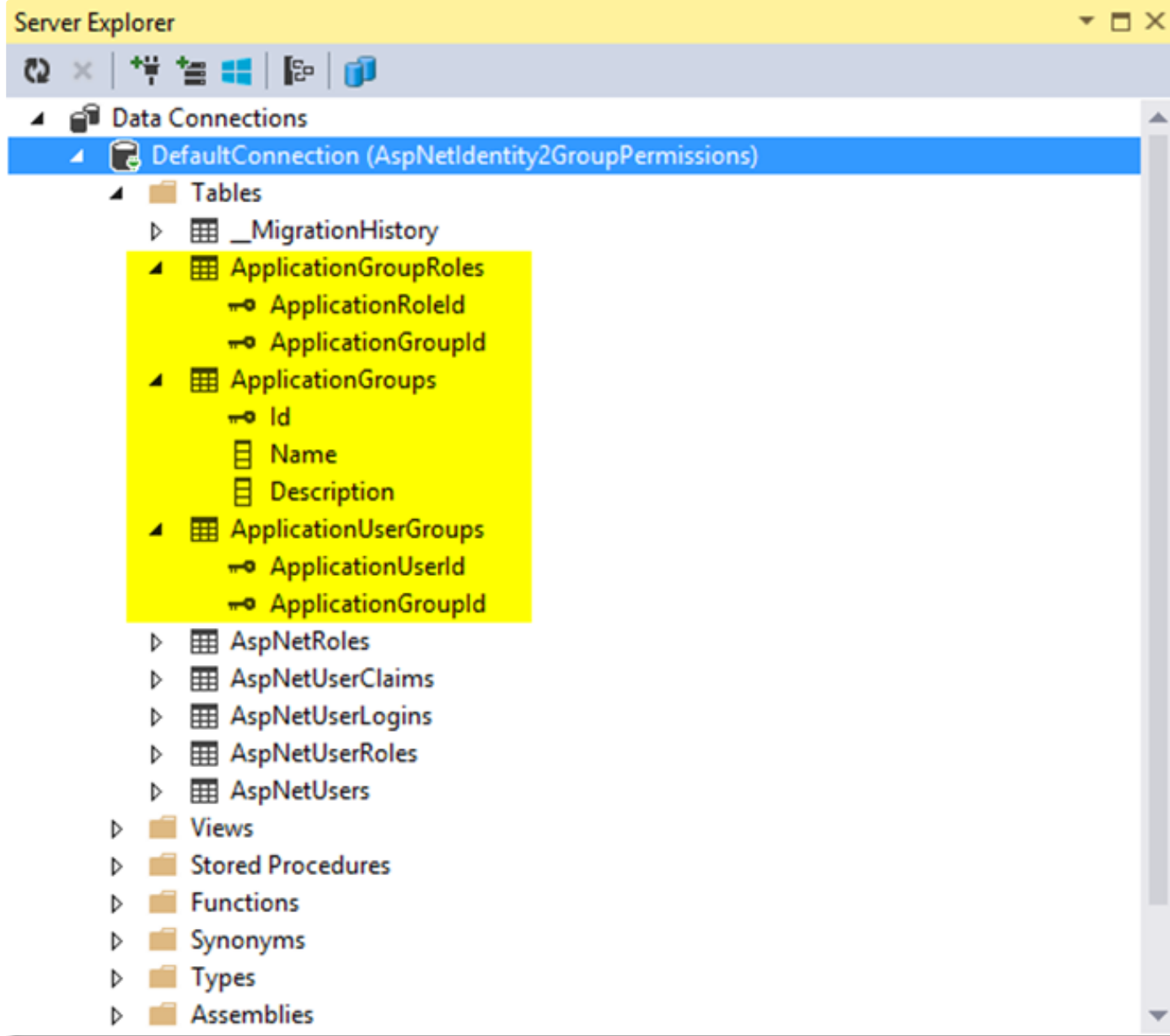
Creation

If we run the project now, we will be greeted by the standard MVC page. Recall that with the Entity Framework Code-First model, database creation will occur at first data access. In other words, as things stand, we need to log in.

To this point, we haven't added any explicit functionality to our front-end site – when we log in, there will be no evidence that our underlying model has changed. We simply want to see if the site starts up properly, and that the database and table we expect are in fact created.

Once we have run the project and logged in, we should be able to stop, and use the Visual Studio Server Explorer to see what our database tables look like. We should see something like the following:

Server Explorer with Additional Tables for Groups:



We see from the above that our `ApplicationGroup` and related classes are now represented by tables in our back-end database, along with the expected columns and primary keys. So far, so good!

Investigation: Building a Consistent Asynchronous Model Architecture

APS.NET Identity 2.0 offers a fully async model architecture. We are going to do our best to follow the conventions established by the Identity team in building up our Groups management structure, using similar abstractions to create a Group store, and Group Manager with fully async methods. In other words, perhaps we can take a look at how the Identity 2.0 team built the basic UserStore and RoleStore abstractions (including the async methods) and simply model up our own GroupStore along the same lines.

If we take a close look at the structure used by the Identity team to build up the basic `UserStore` and `RoleStore` classes within the Identity framework, we find that each are composed around an instance of a class called `EntityStore<TEntity>`, which wraps the most basic behaviors expected of a persistence store.

For example, if we look inside the `RoleStore<TRole, TKey, TUserRole>` class defined as part of the Identity 2.0 framework, we find the following:

Decomposing the RoleStore Class:

```
public class RoleStore<TRole, TKey, TUserRole> :  
    IQueryableRoleStore<TRole, TKey>, IRoleStore<TRole, TKey>, IDisposable  
    where TRole : IdentityRole<TKey, TUserRole>, new()  
    where TUserRole : IdentityUserRole<TKey>, new()
```

```
{
    private bool _disposed;
    private EntityStore<TRole> _roleStore;

    public DbContext Context { get; private set; }
    public bool DisposeContext {get; set; }

    public IQueryable<TRole> Roles
    {
        get
        {
            return this._roleStore.EntitySet;
        }
    }

    public RoleStore(DbContext context)
    {
        if (context == null)
        {
            throw new ArgumentNullException("context");
        }
        this.Context = context;
        this._roleStore = new EntityStore<TRole>(context);
    }

    public virtual async Task CreateAsync(TRole role)
    {
        this.ThrowIfDisposed();
        if (role == null)
        {
            throw new ArgumentNullException("role");
        }
        this._roleStore.Create(role);
        TaskExtensions.CultureAwaiter<int> cultureAwaiter =
            this.Context.SaveChangesAsync().WithCurrentCulture<int>();
        await cultureAwaiter;
    }

    public virtual async Task DeleteAsync(TRole role)
```

```

{
    this.ThrowIfDisposed();
    if (role == null)
    {
        throw new ArgumentNullException("role");
    }
    this._roleStore.Delete(role);
    TaskExtensions.CultureAwaiter<int> cultureAwaiter =
        this.Context.SaveChangesAsync().WithCurrentCulture<int>();
    await cultureAwaiter;
}

public void Dispose()
{
    this.Dispose(true);
    GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
    if (this.DisposeContext && disposing && this.Context != null)
    {
        this.Context.Dispose();
    }
    this._disposed = true;
    this.Context = null;
    this._roleStore = null;
}

public Task<TRole> FindByIdAsync(TKey roleId)
{
    this.ThrowIfDisposed();
    return this._roleStore.GetByIdAsync(roleId);
}

public Task<TRole> FindByNameAsync(string roleName)
{
    this.ThrowIfDisposed();
    return QueryableExtensions

```

```

        .FirstOrDefaultAsync<TRole>(this._roleStore.EntitySet,
            (TRole u) => u.Name.ToUpper() == roleName.ToUpper());
    }

    private void ThrowIfDisposed()
    {
        if (this._disposed)
        {
            throw new ObjectDisposedException(this.GetType().Name);
        }
    }

    public virtual async Task UpdateAsync(TRole role)
    {
        this.ThrowIfDisposed();
        if (role == null)
        {
            throw new ArgumentNullException("role");
        }
        this._roleStore.Update(role);
        TaskExtensions.CultureAwaiter<int> cultureAwaiter =
            this.Context.SaveChangesAsync().WithCurrentCulture<int>();
        await cultureAwaiter;
    }
}

```

The code above is interesting, and we will be looking more closely in a bit. For the moment, notice the highlighted item. `RoleStore` wraps an instance of `EntityStore<TRole>`. If we follow our noses a little further, we find a definition for `EntityStore` as well:

The EntityStore Class from Identity 2.0 Framework:

```

internal class EntityStore<TEntity>

```



```
where TEntity : class
{
    public DbContext Context { get; private set; }
    public DbSet<TEntity> DbSet { get; private set; }
    public IQueryable<TEntity> EntitySet
    {
        get
        {
            return this.DbEntitySet;
        }
    }
    public EntityStore(DbContext context)
    {
        this.Context = context;
        this.DbEntitySet = context.Set<TEntity>();
    }
    public void Create(TEntity entity)
    {
        this.DbEntitySet.Add(entity);
    }
    public void Delete(TEntity entity)
    {
        this.DbEntitySet.Remove(entity);
    }
    public virtual Task<TEntity> GetByIdAsync(object id)
    {
        return this.DbEntitySet.FindAsync(new object[] { id });
    }
    public virtual void Update(TEntity entity)
    {
        if (entity != null)
        {
            this.Context.Entry<TEntity>(entity).State = EntityState.Modified;
        }
    }
}
```

This code is also of great interest, despite its simplicity. Unfortunately, we cannot directly use the `EntityStore` class within our project. Note the internal modifier in the class declaration – this means `EntityStore` is only available to classes within the `Microsoft.AspNet.Identity.EntityFramework` assembly. In other words, we can't consume `EntityStore` in order to build up our own `GroupStore` implementation. Instead, we will take the time-honored approach of stealing/copying.

Building an Asynchronous GroupStore

We are going to apply the same conventions used by the Identity team in building out a `GroupStore` class, and then, in similar fashion, wrap a `GroupManager` class around THAT, much the same as Identity Framework wraps a `RoleManager` class around an instance of `RoleStore`.

But first, we need to deal with the `EntityStore` problem. In order to properly mimic the structure used to build up `RoleStore` and `UserStore`, we need to basically create our own implementation of `EntityStore`. In our case, we don't need a generically-typed class – we can create a non-generic implementation specific to our needs.

Mimicking EntityStore – Building the GroupStoreBase Class

We can basically steal most of the code from the `EntityStore<TEntity>` class shown above, and adapt it to suit our needs by removing the generic type arguments for the class itself, and by passing non-generic arguments where needed. Add a class named `GroupStoreBase` to the Models folder of your project, and then use the following code for the class itself. First, you will need the following using statements at the top of your code file:

Required Assembly References for the GroupStoreBase Class:

```
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;
```

The GroupStoreBase Class:

```
public class GroupStoreBase
{
    public DbContext Context { get; private set; }
    public DbSet<ApplicationGroup> DbEntitySet { get; private set; }

    public IQueryable<ApplicationGroup> EntitySet
    {
        get
        {
            return this.DbEntitySet;
        }
    }

    public GroupStoreBase(DbContext context)
    {
        this.Context = context;
    }
}
```

```
        this.DbEntitySet = context.Set<ApplicationGroup>();
    }

    public void Create(ApplicationGroup entity)
    {
        this.DbEntitySet.Add(entity);
    }

    public void Delete(ApplicationGroup entity)
    {
        this.DbEntitySet.Remove(entity);
    }

    public virtual Task<ApplicationGroup> GetByIdAsync(object id)
    {
        return this.DbEntitySet.FindAsync(new object[] { id });
    }

    public virtual ApplicationGroup GetById(object id)
    {
        return this.DbEntitySet.Find(new object[] { id });
    }

    public virtual void Update(ApplicationGroup entity)
    {
        if (entity != null)
        {
            this.Context.Entry<ApplicationGroup>(entity).State = EntityState
        }
    }
}
```

Note the structure here. the `GroupStoreBase` provides methods for working with a `DbSet<ApplicationGroup>`, but performs no persistence to the backside database directly. Persistence will be controlled by the next class in our structure, `ApplicationGroupStore`.

Building the Primary ApplicationGroupStore Class

In following the pattern used by `UserStore` and `RoleStore`, we will now build out a `GroupStore` class, which will be composed around our new `GroupStoreBase` class. Add another class to the models folder named `ApplicationGroupStore`, and add the following code:

The ApplicationGroupStore Class:

```
public class ApplicationGroupStore : IDisposable
{
    private bool _disposed;
    private GroupStoreBase _groupStore;

    public ApplicationGroupStore(DbContext context)
    {
        if (context == null)
        {
            throw new ArgumentNullException("context");
        }
        this.Context = context;
        this._groupStore = new GroupStoreBase(context);
    }
}
```

```
public IQueryable<ApplicationGroup> Groups
{
    get
    {
        return this._groupStore.EntitySet;
    }
}
```

```
public DbContext Context
{
    get;
    private set;
}
```

```
public virtual void Create(ApplicationGroup group)
{
    this.ThrowIfDisposed();
    if (group == null)
    {
        throw new ArgumentNullException("group");
    }
    this._groupStore.Create(group);
    this.Context.SaveChanges();
}
```

```
public virtual async Task CreateAsync(ApplicationGroup group)
{
    this.ThrowIfDisposed();
    if (group == null)
    {
        throw new ArgumentNullException("group");
    }
    this._groupStore.Create(group);
    await this.Context.SaveChangesAsync();
}
```

```
public virtual async Task DeleteAsync(ApplicationGroup group)
{
    this.ThrowIfDisposed();
    if (group == null)
    {
        throw new ArgumentNullException("group");
    }
    this._groupStore.Delete(group);
    await this.Context.SaveChangesAsync();
}
```

```
public virtual void Delete(ApplicationGroup group)
{
    this.ThrowIfDisposed();
    if (group == null)
    {
        throw new ArgumentNullException("group");
    }
    this._groupStore.Delete(group);
    this.Context.SaveChanges();
}
```

```
public Task<ApplicationGroup> FindByIdAsync(string roleId)
{
    this.ThrowIfDisposed();
    return this._groupStore.GetByIdAsync(roleId);
}
```

```
public ApplicationGroup FindById(string roleId)
{
    this.ThrowIfDisposed();
    return this._groupStore.GetById(roleId);
}
```

```
public Task<ApplicationGroup> FindByNameAsync(string groupName)
{
    this.ThrowIfDisposed();
    return QueryableExtensions
        .FirstOrDefaultAsync<ApplicationGroup>(this._groupStore.Entity
            (ApplicationGroup u) => u.Name.ToUpper() == groupName.ToUp
        }
}
```

```
public virtual async Task UpdateAsync(ApplicationGroup group)
{
    this.ThrowIfDisposed();
    if (group == null)
    {
        throw new ArgumentNullException("group");
    }
    this._groupStore.Update(group);
    await this.Context.SaveChangesAsync();
}
```

```
public virtual void Update(ApplicationGroup group)
{
    this.ThrowIfDisposed();
    if (group == null)
    {
        throw new ArgumentNullException("group");
    }
    this._groupStore.Update(group);
    this.Context.SaveChanges();
}
```

```
// DISPOSE STUFF: =====
```

```
public bool DisposeContext
{
    get;
    set;
}
```



```

    }

    private void ThrowIfDisposed()
    {
        if (this._disposed)
        {
            throw new ObjectDisposedException(this.GetType().Name);
        }
    }

    public void Dispose()
    {
        this.Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (this.DisposeContext && disposing && this.Context != null)
        {
            this.Context.Dispose();
        }
        this._disposed = true;
        this.Context = null;
        this._groupStore = null;
    }
}

```

Some things to note about the `ApplicationGroupStore` class. First, notice that this class takes care of all the actual persistence to the backing store, by virtue of calls to `.SaveChanges()` or `.SaveChangesAsync()`. Also, we have provided both an async and a synchronous implementation for each of the methods.

However, we're not quite done yet. While the `ApplicationGroupStore` manages the basics of persistence for Groups, what it does NOT do is handle the complexities introduced by the relationships between Groups, Users, and Roles. Here, we can perform the basic CRUD operations on our groups, but we have no control over the relationships between the classes.

This becomes the job of the `ApplicationGroupManager` class.

Managing Complex Relationships – The ApplicationGroupManager Class

The relationships between Users, Groups, and Roles, for the sake of our application, are more complex than they might first appear.

Our Users-Groups-Roles structure is actually performing a bit of an illusion. It may appear, when we are done, that Roles will “belong” to groups, and that Users, by virtue of membership in a particular group, gain access to the Roles of that Group. However, what is really going on is that, when a User is assigned to a particular group, our application is then adding that user to each Role within the Group.

This is a subtle, but important distinction.

Let's assume we have an existing Group with two Roles assigned –

“CanEditAccount” and “CanViewAccount.” Let’s further assume that there are three users in this group. Finally, let’s say we want to add another (already existing) Role to this group – “CanDeleteAccount.” What needs to happen?

1. We assign the role to the group
2. We need to add each member or the group to the new role

On the face of it, that is relatively straightforward. However, each User can belong to more than one group. Also, a Role can be assigned to more than one group. What if we want to remove a Role from a group?

1. Remove each User in the Group from the Role, except when they are also a member of another group which also has that same role
2. Remove the Role from the Group

This is a little more complicated. A similar situation arises if we wish to remove a User from a Group:

1. Remove the User from all Roles in the Group, except when the user also belongs to another Group with the same role
2. Remove the User from the Group

And so on. In order to get the predictable, intuitive behavior from our application

which will be expected by the end user, there is more going on than meets the eye.

It will be the job of the `ApplicationGroupManager` to handle these types of problems for us, and afford a clear API against which our controllers can work between the user and the backside data.

We have created the `GroupStore` class to handle the basic persistence of Group data, and we have the `ApplicationUserManager` and `ApplicationRoleManager` classes to handle the relationships between Users, Roles, and persistence. For the most part, the `ApplicationGroupManager's` job will mainly be governing the interactions of these three stores, and occasionally the `DbContext` directly.

We will accomplish this by defining an API which, similar to the Identity base classes `userManager` and `RoleManager`, affords us the intuitive methods we need to deal with Group-based Role management. We will provide both synchronous and asynchronous implementations.

Add another class to the *Models* folder, and name it `ApplicationGroupManager`. You will need the following `using` statements at the top of your code file:

Required Assembly References for `ApplicationGroupManager`:

```
using Microsoft.AspNet.Identity;  
using Microsoft.AspNet.Identity.Owin;  
using System;  
using System.Collections.Generic;
```

```
using System.Data.Entity;  
using System.Linq;  
using System.Threading.Tasks;  
using System.Web;
```

Now, add the following code:

The ApplicationGroupManager Class:

```
public class ApplicationGroupManager  
{  
    private ApplicationGroupStore _groupStore;  
    private ApplicationDbContext _db;  
    private ApplicationUserManager _userManager;  
    private ApplicationRoleManager _roleManager;  
  
    public ApplicationGroupManager()  
    {  
        _db = HttpContext.Current  
            .GetOwinContext().Get<ApplicationDbContext>();  
        _userManager = HttpContext.Current  
            .GetOwinContext().GetUserManager<ApplicationUserManager>();  
        _roleManager = HttpContext.Current  
            .GetOwinContext().Get<ApplicationRoleManager>();  
        _groupStore = new ApplicationGroupStore(_db);  
    }  
  
    public IQueryable<ApplicationGroup> Groups  
    {  
        get  
        {  
            return _groupStore.Groups;  
        }  
    }  
}
```

```
public async Task<IdentityResult> CreateGroupAsync(ApplicationGroup group)
{
    await _groupStore.CreateAsync(group);
    return IdentityResult.Success;
}
```

```
public IdentityResult CreateGroup(ApplicationGroup group)
{
    _groupStore.Create(group);
    return IdentityResult.Success;
}
```

```
public IdentityResult SetGroupRoles(string groupId, params string[] roleNames)
{
    // Clear all the roles associated with this group:
    var thisGroup = this.FindById(groupId);
    thisGroup.ApplicationRoles.Clear();
    _db.SaveChanges();

    // Add the new roles passed in:
    var newRoles = _roleManager.Roles.Where(r => roleNames.Any(n => n == r.Name));
    foreach(var role in newRoles)
    {
        thisGroup.ApplicationRoles.Add(new ApplicationGroupRole
        {
            ApplicationGroupId = groupId,
            ApplicationRoleId = role.Id
        });
    }
    _db.SaveChanges();

    // Reset the roles for all affected users:
    foreach(var groupUser in thisGroup.ApplicationUsers)
    {
        this.RefreshUserGroupRoles(groupUser.ApplicationUserId);
    }
    return IdentityResult.Success;
}
```

```

    }

    public async Task<IdentityResult> SetGroupRolesAsync(
        string groupId, params string[] roleNames)
    {
        // Clear all the roles associated with this group:
        var thisGroup = await this.FindByIdAsync(groupId);
        thisGroup.ApplicationRoles.Clear();
        await _db.SaveChangesAsync();

        // Add the new roles passed in:
        var newRoles = _roleManager.Roles
            .Where(r => roleNames.Any(n => n == r.Name));
        foreach (var role in newRoles)
        {
            thisGroup.ApplicationRoles.Add(new ApplicationGroupRole
            {
                ApplicationGroupId = groupId,
                ApplicationRoleId = role.Id
            });
        }
        await _db.SaveChangesAsync();

        // Reset the roles for all affected users:
        foreach (var groupUser in thisGroup.ApplicationUsers)
        {
            await this.RefreshUserGroupRolesAsync(groupUser.ApplicationUser);
        }
        return IdentityResult.Success;
    }
}

```

```

    public async Task<IdentityResult> SetUserGroupsAsync(
        string userId, params string[] groupIds)
    {
        // Clear current group membership:
        var currentGroups = await this.GetUserGroupsAsync(userId);
        foreach (var group in currentGroups)
        {
            await this.RemoveUserFromGroupAsync(userId, group.Id);
        }
    }
}

```

```

{
    group.ApplicationUsers
        .Remove(group.ApplicationUsers
            .FirstOrDefault(gr => gr.ApplicationUserId == userId
        ));
}
await _db.SaveChangesAsync();

// Add the user to the new groups:
foreach (string groupId in groupIds)
{
    var newGroup = await this.FindByIdAsync(groupId);
    newGroup.ApplicationUsers.Add(new ApplicationUserGroup
    {
        ApplicationUserId = userId,
        ApplicationGroupId = groupId
    });
}
await _db.SaveChangesAsync();

await this.RefreshUserGroupRolesAsync(userId);
return IdentityResult.Success;
}

```

```

public IdentityResult SetUserGroups(string userId, params string[] groupIds)
{
    // Clear current group membership:
    var currentGroups = this.GetUserGroups(userId);
    foreach(var group in currentGroups)
    {
        group.ApplicationUsers
            .Remove(group.ApplicationUsers
                .FirstOrDefault(gr => gr.ApplicationUserId == userId
            ));
    }
    _db.SaveChanges();

    // Add the user to the new groups:

```



```

foreach(string groupId in groupIds)
{
    var newGroup = this.FindById(groupId);
    newGroup.ApplicationUsers.Add(new ApplicationUserGroup
    {
        ApplicationUserId = userId,
        ApplicationGroupId = groupId
    });
}
_db.SaveChanges();

this.RefreshUserGroupRoles(userId);
return IdentityResult.Success;
}

public IdentityResult RefreshUserGroupRoles(string userId)
{
    var user = _userManager.FindById(userId);
    if(user == null)
    {
        throw new ArgumentNullException("User");
    }
    // Remove user from previous roles:
    var oldUserRoles = _userManager.GetRoles(userId);
    if(oldUserRoles.Count > 0)
    {
        _userManager.RemoveFromRoles(userId, oldUserRoles.ToArray());
    }

    // Find the roles this user is entitled to from group membership:
    var newGroupRoles = this.GetUserGroupRoles(userId);

    // Get the damn role names:
    var allRoles = _roleManager.Roles.ToList();
    var addTheseRoles = allRoles
        .Where(r => newGroupRoles.Any(gr => gr.ApplicationRoleId == r.Id)
    );
    var roleNames = addTheseRoles.Select(n => n.Name).ToArray();
}

```

```

        // Add the user to the proper roles
        _userManager.AddToRoles(userId, roleNames);

        return IdentityResult.Success;
    }

    public async Task<IdentityResult> RefreshUserGroupRolesAsync(string userId)
    {
        var user = await _userManager.FindByIdAsync(userId);
        if (user == null)
        {
            throw new ArgumentNullException("User");
        }
        // Remove user from previous roles:
        var oldUserRoles = await _userManager.GetRolesAsync(userId);
        if (oldUserRoles.Count > 0)
        {
            await _userManager.RemoveFromRolesAsync(userId, oldUserRoles.ToArray());
        }

        // Find the roles this user is entitled to from group membership:
        var newGroupRoles = await this.GetUserGroupRolesAsync(userId);

        // Get the damn role names:
        var allRoles = await _roleManager.Roles.ToListAsync();
        var addTheseRoles = allRoles
            .Where(r => newGroupRoles.Any(gr => gr.ApplicationRoleId == r.Id))
            .ToList();
        var roleNames = addTheseRoles.Select(n => n.Name).ToArray();

        // Add the user to the proper roles
        await _userManager.AddToRolesAsync(userId, roleNames);

        return IdentityResult.Success;
    }

```

```
public async Task<IdentityResult> DeleteGroupAsync(string groupId)
{
    var group = await this.FindByIdAsync(groupId);
    if (group == null)
    {
        throw new ArgumentNullException("User");
    }

    var currentGroupMembers = (await this.GetGroupUsersAsync(groupId))
    // remove the roles from the group:
    group.ApplicationRoles.Clear();

    // Remove all the users:
    group.ApplicationUsers.Clear();

    // Remove the group itself:
    _db.ApplicationGroups.Remove(group);

    await _db.SaveChangesAsync();

    // Reset all the user roles:
    foreach (var user in currentGroupMembers)
    {
        await this.RefreshUserGroupRolesAsync(user.Id);
    }
    return IdentityResult.Success;
}
```

```
public IdentityResult DeleteGroup(string groupId)
{
    var group = this.FindById(groupId);
    if (group == null)
    {
        throw new ArgumentNullException("User");
    }

    var currentGroupMembers = this.GetGroupUsers(groupId).ToList();
    // remove the roles from the group:
```

```
group.ApplicationRoles.Clear();

// Remove all the users:
group.ApplicationUsers.Clear();

// Remove the group itself:
_db.ApplicationGroups.Remove(group);

_db.SaveChanges();

// Reset all the user roles:
foreach(var user in currentGroupMembers)
{
    this.RefreshUserGroupRoles(user.Id);
}
return IdentityResult.Success;
}

public async Task<IdentityResult> UpdateGroupAsync(ApplicationGroup group)
{
    await _groupStore.UpdateAsync(group);
    foreach (var groupUser in group.ApplicationUsers)
    {
        await this.RefreshUserGroupRolesAsync(groupUser.ApplicationUserId);
    }
    return IdentityResult.Success;
}

public IdentityResult UpdateGroup(ApplicationGroup group)
{
    _groupStore.Update(group);
    foreach(var groupUser in group.ApplicationUsers)
    {
        this.RefreshUserGroupRoles(groupUser.ApplicationUserId);
    }
    return IdentityResult.Success;
}
```

```
public IdentityResult ClearUserGroups(string userId)
{
    return this.SetUserGroups(userId, new string[] { });
}
```

```
public async Task<IdentityResult> ClearUserGroupsAsync(string userId)
{
    return await this.SetUserGroupsAsync(userId, new string[] { });
}
```

```
public async Task<IEnumerable<ApplicationGroup>> GetUserGroupsAsync(st
{
    var result = new List<ApplicationGroup>();
    var userGroups = (from g in this.Groups
                      where g.ApplicationUsers
                          .Any(u => u.ApplicationUserId == userId)
                      select g).ToListAsync();
    return await userGroups;
}
```

```
public IEnumerable<ApplicationGroup> GetUserGroups(string userId)
{
    var result = new List<ApplicationGroup>();
    var userGroups = (from g in this.Groups
                      where g.ApplicationUsers
                          .Any(u => u.ApplicationUserId == userId)
                      select g).ToList();
    return userGroups;
}
```

```
public async Task<IEnumerable<ApplicationRole>> GetGroupRolesAsync(
    string groupId)
{

```

```

        var grp = await _db.ApplicationGroups
            .FirstOrDefaultAsync(g => g.Id == groupId);
        var roles = await _roleManager.Roles.ToListAsync();
        var groupRoles = (from r in roles
                           where grp.ApplicationRoles
                               .Any(ap => ap.ApplicationRoleId == r.Id)
                           select r).ToList();

        return groupRoles;
    }

```

```

public IEnumerable<ApplicationRole> GetGroupRoles(string groupId)
{
    var grp = _db.ApplicationGroups.FirstOrDefault(g => g.Id == groupId);
    var roles = _roleManager.Roles.ToList();
    var groupRoles = from r in roles
                       where grp.ApplicationRoles
                           .Any(ap => ap.ApplicationRoleId == r.Id)
                       select r;

    return groupRoles;
}

```

```

public IEnumerable<ApplicationUser> GetGroupUsers(string groupId)
{
    var group = this.FindById(groupId);
    var users = new List<ApplicationUser>();
    foreach (var groupUser in group.ApplicationUsers)
    {
        var user = _db.Users.Find(groupUser.ApplicationUserId);
        users.Add(user);
    }
    return users;
}

```

```

public async Task<IEnumerable<ApplicationUser>> GetGroupUsersAsync(string groupId)
{
    var group = await this.FindByIdAsync(groupId);

```

```

        var users = new List<ApplicationUser>();
        foreach (var groupUser in group.ApplicationUsers)
        {
            var user = await _db.Users
                .FirstOrDefaultAsync(u => u.Id == groupUser.ApplicationUser.Id);
            users.Add(user);
        }
        return users;
    }
}

```

```

public IEnumerable<ApplicationGroupRole> GetUserGroupRoles(string userId)
{
    var userGroups = this.GetUserGroups(userId);
    var userGroupRoles = new List<ApplicationGroupRole>();
    foreach(var group in userGroups)
    {
        userGroupRoles.AddRange(group.ApplicationRoles.ToArray());
    }
    return userGroupRoles;
}

```

```

public async Task<IEnumerable<ApplicationGroupRole>> GetUserGroupRolesAsync(
    string userId)
{
    var userGroups = await this.GetUserGroupsAsync(userId);
    var userGroupRoles = new List<ApplicationGroupRole>();
    foreach (var group in userGroups)
    {
        userGroupRoles.AddRange(group.ApplicationRoles.ToArray());
    }
    return userGroupRoles;
}

```

```

public async Task<ApplicationGroup> FindByIdAsync(string id)
{
    return await _groupStore.FindByIdAsync(id);
}

```

```
}

public ApplicationGroup FindById(string id)
{
    return _groupStore.FindById(id);
}
}
```

As we can see from the above, there's a lot of code there. However, much of it is due to duplication between synchronous and asynchronous method implementations. With the above, we now have an API to work against from our controllers, and we can now get down to brass tacks, and add the Groups functionality to our site.

We've kept things as simple as possible in terms of adding and removing Users from Groups, Roles to and from Groups, and such. As you can see, each time we change the groups a user belongs to, we change all the group assignments at once, by calling `SetUserGroups()` and passing in an array of Group Id's. Similarly, we assign all of the Roles to a group in one shot by calling `SetGroupRoles()` and again, passing in an array of Role names, representing all of the roles assigned to a particular group.

We do these in this manner because, when we modify a User's Group membership, we need to essentially re-set all of the User's Roles anyway. In like manner, when we modify the Roles assigned to a particular group, we need to refresh the Roles assigned to every user within that group.

This is also handy because, when we receive the User and/or Role selections made by the user from either of the Admin Views, we get them as an array anyway. We'll see this more closely in a bit.

Adding a GroupViewModel

We will need a view model for passing group data between various controller methods and their associated Views. in the *Models => AdminViewModel.cs* file, add the following class:

The GroupViewModel Class:

```
public class GroupViewModel
{
    public GroupViewModel()
    {
        this.UsersList = new List<SelectListItem>();
        this.PermissionsList = new List<SelectListItem>();
    }
    [Required(AllowEmptyStrings = false)]
    public string Id { get; set; }
    [Required(AllowEmptyStrings = false)]
    public string Name { get; set; }
    public string Description { get; set; }
    public ICollection<SelectListItem> UsersList { get; set; }
    public ICollection<SelectListItem> RolesList { get; set; }
}
```

Notice here that we are passing `ICollection<SelectListItem>` to represent Users and Roles assigned to a Group. This way, we can pass a list of user names

or or role names out to a view, allow the user to select one or more items from the list, and then process the selection choices made when the form data is submitted back to the controller via the HTTP POST method.

Update the EditUserViewModel

While we have the *AdminViewModel.cs* file open, let's also modify the `EditUserViewModel`, and add a collection property for Groups:

Add a GroupsList Property to EditUserViewModel:

```
public class EditUserViewModel
{
    public EditUserViewModel()
    {
        this.RolesList = new List<SelectListItem>();
        this.GroupsList = new List<SelectListItem>();
    }
    public string Id { get; set; }
    [Required(AllowEmptyStrings = false)]
    [Display(Name = "Email")]
    [EmailAddress]
    public string Email { get; set; }

    // We will still use this, so leave it here:
    public ICollection<SelectListItem> RolesList { get; set; }

    // Add a GroupsList Property:
    public ICollection<SelectListItem> GroupsList { get; set; }
}
```

We will keep the `RolesList` collection as well. Even though we won't be assigning Roles directly to Users anymore, we may want to create a View which allows us to see what Roles a user has by virtue of membership in various Groups. This way, we can use the same ViewModel.

Now that we have our ViewModel all tuned up, let's add a `GroupsAdminController`.

Building the GroupsAdminController

Similar to the existing `UserAdminController` and the `RolesAdminController`, we need to provide a Controller to work with our new Groups functionality.

We need to add a controller in the *Controllers* directory. Instead of using the *Context Menu => Add Controller* method, just add a class, named `GroupsAdminController`, and add the following code:

Adding the GroupsAdminController:

```
public class GroupsAdminController : Controller
{
    private ApplicationDbContext db = new ApplicationDbContext();

    private ApplicationGroupManager _groupManager;
    public ApplicationGroupManager GroupManager
    {
        get
        {
```

```

        return _groupManager ?? new ApplicationGroupManager();
    }
    private set
    {
        _groupManager = value;
    }
}

private ApplicationRoleManager _roleManager;
public ApplicationRoleManager RoleManager
{
    get
    {
        return _roleManager ?? HttpContext.GetOwinContext()
            .Get<ApplicationRoleManager>();
    }
    private set
    {
        _roleManager = value;
    }
}

public ActionResult Index()
{
    return View(this.GroupManager.Groups.ToList());
}

public async Task<ActionResult> Details(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    ApplicationGroup applicationgroup =
        await this.GroupManager.Groups.FirstOrDefaultAsync(g => g.Id == id);
    if (applicationgroup == null)
    {

```

```

        return HttpNotFound();
    }
    var groupRoles = this.GroupManager.GetGroupRoles(applicationgroup.Id);
    string[] RoleNames = groupRoles.Select(p => p.Name).ToArray();
    ViewBag.RolesList = RoleNames;
    ViewBag.RolesCount = RoleNames.Count();
    return View(applicationgroup);
}

```

```

public ActionResult Create()
{
    //Get a SelectList of Roles to choose from in the View:
    ViewBag.RolesList = new SelectList(
        this.RoleManager.Roles.ToList(), "Id", "Name");
    return View();
}

```

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create(
    [Bind(Include = "Name,Description")] ApplicationGroup applicationgroup,
    params string[] selectedRoles)
{
    if (ModelState.IsValid)
    {
        // Create the new Group:
        var result = await this.GroupManager.CreateGroupAsync(applicationgroup);
        if (result.Succeeded)
        {
            selectedRoles = selectedRoles ?? new string[] { };

            // Add the roles selected:
            await this.GroupManager
                .SetGroupRolesAsync(applicationgroup.Id, selectedRoles);
        }
        return RedirectToAction("Index");
    }
}

```

```

// Otherwise, start over:
ViewBag.RoleId = new SelectList(
    this.RoleManager.Roles.ToList(), "Id", "Name");
return View(applicationgroup);
}

public async Task<ActionResult> Edit(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    ApplicationGroup applicationgroup = await this.GroupManager.FindBy
    if (applicationgroup == null)
    {
        return HttpNotFound();
    }

    // Get a list, not a DbSet or queryable:
    var allRoles = await this.RoleManager.Roles.ToListAsync();
    var groupRoles = await this.GroupManager.GetGroupRolesAsync(id);

    var model = new GroupViewModel()
    {
        Id = applicationgroup.Id,
        Name = applicationgroup.Name,
        Description = applicationgroup.Description
    };

    // load the roles/Roles for selection in the form:
    foreach (var Role in allRoles)
    {
        var listItem = new SelectListItem()
        {
            Text = Role.Name,
            Value = Role.Id,
            Selected = groupRoles.Any(g => g.Id == Role.Id)
        }
    }
}

```

```

        };
        model.RolesList.Add(listItem);
    }
    return View(model);
}

```

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit(
    [Bind(Include = "Id,Name,Description")] GroupViewModel model,
    params string[] selectedRoles)
{
    var group = await this.GroupManager.FindByIdAsync(model.Id);
    if (group == null)
    {
        return HttpNotFound();
    }
    if (ModelState.IsValid)
    {
        group.Name = model.Name;
        group.Description = model.Description;
        await this.GroupManager.UpdateGroupAsync(group);

        selectedRoles = selectedRoles ?? new string[] { };
        await this.GroupManager.SetGroupRolesAsync(group.Id, selectedRoles);
        return RedirectToAction("Index");
    }
    return View(model);
}

```

```

public async Task<ActionResult> Delete(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    ApplicationGroup applicationgroup = await this.GroupManager.FindBy

```

```

        if (applicationgroup == null)
        {
            return HttpNotFound();
        }
        return View(applicationgroup);
    }

    [HttpPost, ActionName("Delete")]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> DeleteConfirmed(string id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        ApplicationGroup applicationgroup = await this.GroupManager.FindBy
        await this.GroupManager.DeleteGroupAsync(id);
        return RedirectToAction("Index");
    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            db.Dispose();
        }
        base.Dispose(disposing);
    }
}

```

As we can see, we use the **GroupsAdminController** to manage the creation of groups, and the assignment of roles to various groups. Now, we need to modify the **UsersAdminController** so that instead of assigning Users to Roles directly,

we are instead assigning Users to Groups, through which they gain access to the roles assigned to each group.

Modify the UsersAdminController

We need to make a few adjustments to the `UsersAdminController` to reflect the manner in which we are managing Groups and Roles. As mentioned previously, we are now assigning users to Groups instead of directly to roles, and our `UsersAdminController` needs to reflect this.

First off, we need to add an instance of `ApplicationGroupManager` to the controller. Next, we need to update all of our controller methods to consume Groups instead of Roles. When we create a new User, we want the View to include a list of available Groups to which the User might be assigned. When we Edit an existing User, we want the option to modify group assignments. When we Delete a user, we need to make sure the corresponding Group relationships are deleted as well.

The Following is the updated code for the entire `UsersAdminController`. It is easier to copy the entire thing in than to wade through it piece by piece. Then you can eyeball things, and fairly easily understand what is going on in each controller method.

The Modified UsersAdminController:

```
[Authorize(Roles = "Admin")]
public class UsersAdminController : Controller
{
    public UsersAdminController()
    {
    }

    public UsersAdminController(ApplicationUserManager userManager,
        ApplicationRoleManager roleManager)
    {
        UserManager = userManager;
        RoleManager = roleManager;
    }

    private ApplicationUserManager _userManager;
    public ApplicationUserManager UserManager
    {
        get
        {
            return _userManager ?? HttpContext.GetOwinContext()
                .GetUserManager<ApplicationUserManager>();
        }
        private set
        {
            _userManager = value;
        }
    }

    // Add the Group Manager (NOTE: only access through the public
    // Property, not by the instance variable!)
    private ApplicationGroupManager _groupManager;
    public ApplicationGroupManager GroupManager
    {
        get
        {
            return _groupManager ?? new ApplicationGroupManager();
        }
        private set
    }
}
```

```
    {
        _groupManager = value;
    }
}

private ApplicationRoleManager _roleManager;
public ApplicationRoleManager RoleManager
{
    get
    {
        return _roleManager ?? HttpContext.GetOwinContext()
            .Get<ApplicationRoleManager>();
    }
    private set
    {
        _roleManager = value;
    }
}

public async Task<ActionResult> Index()
{
    return View(await UserManager.Users.ToListAsync());
}

public async Task<ActionResult> Details(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var user = await UserManager.FindByIdAsync(id);

    // Show the groups the user belongs to:
    var userGroups = await this.GroupManager.GetUserGroupsAsync(id);
    ViewBag.GroupNames = userGroups.Select(u => u.Name).ToList();
    return View(user);
}
```

```

public ActionResult Create()
{
    // Show a list of available groups:
    ViewBag.GroupsList =
        new SelectList(this.GroupManager.Groups, "Id", "Name");
    return View();
}

```

```

[HttpPost]
public async Task<ActionResult> Create(RegisterViewModel userViewModel,
    params string[] selectedGroups)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser
        {
            UserName = userViewModel.Email,
            Email = userViewModel.Email
        };
        var adminresult = await UserManager
            .CreateAsync(user, userViewModel.Password);

        //Add User to the selected Groups
        if (adminresult.Succeeded)
        {
            if (selectedGroups != null)
            {
                selectedGroups = selectedGroups ?? new string[] { };
                await this.GroupManager
                    .SetUserGroupsAsync(user.Id, selectedGroups);
            }
            return RedirectToAction("Index");
        }
    }
    ViewBag.Groups = new SelectList(
        await RoleManager.Roles.ToListAsync(), "Id", "Name");
}

```

```
        return View();
    }

    public async Task<ActionResult> Edit(string id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        var user = await UserManager.FindByIdAsync(id);
        if (user == null)
        {
            return HttpNotFound();
        }

        // Display a list of available Groups:
        var allGroups = this.GroupManager.Groups;
        var userGroups = await this.GroupManager.GetUserGroupsAsync(id);

        var model = new EditUserViewModel()
        {
            Id = user.Id,
            Email = user.Email
        };

        foreach (var group in allGroups)
        {
            var listItem = new SelectListItem()
            {
                Text = group.Name,
                Value = group.Id,
                Selected = userGroups.Any(g => g.Id == group.Id)
            };
            model.GroupsList.Add(listItem);
        }
        return View(model);
    }
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit(
    [Bind(Include = "Email,Id")] EditUserViewModel editUser,
    params string[] selectedGroups)
{
    if (ModelState.IsValid)
    {
        var user = await UserManager.FindByIdAsync(editUser.Id);
        if (user == null)
        {
            return HttpNotFound();
        }

        // Update the User:
        user.UserName = editUser.Email;
        user.Email = editUser.Email;
        await this.UserManager.UpdateAsync(user);

        // Update the Groups:
        selectedGroups = selectedGroups ?? new string[] { };
        await this.GroupManager.SetUserGroupsAsync(user.Id, selectedGr
        return RedirectToAction("Index");
    }
    ModelState.AddModelError("", "Something failed.");
    return View();
}
```

```
public async Task<ActionResult> Delete(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var user = await UserManager.FindByIdAsync(id);
    if (user == null)
    {
```

```
        return HttpNotFound();
    }
    return View(user);
}

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<ActionResult> DeleteConfirmed(string id)
{
    if (ModelState.IsValid)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }

        var user = await UserManager.FindByIdAsync(id);
        if (user == null)
        {
            return HttpNotFound();
        }

        // Remove all the User Group references:
        await this.GroupManager.ClearUserGroupsAsync(id);

        // Then Delete the User:
        var result = await UserManager.DeleteAsync(user);
        if (!result.Succeeded)
        {
            ModelState.AddModelError("", result.Errors.First());
            return View();
        }
        return RedirectToAction("Index");
    }
    return View();
}
```

Now that we have all of our Controllers in place, we need to add and/or update some Views.

Add Groups Admin as a Menu Item in the Main Layout View

In order to access our new Groups, we will need to add a menu item to *Views => Shared => _Layout.cshtml*. Since we only want Admin users to access menu item, we want to tuck it in with the other links to Admin-type Views. Modify the *_Layout.cshtml* file as follows (I've only included the relevant section of the View template below):

Add Groups Admin as a Link in the Main Layout View:

```
// Other View Code before...:

<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>

  @if (Request.IsAuthenticated && User.IsInRole("Admin")) {
    <li>@Html.ActionLink("RolesAdmin", "Index", "RolesAdmin")</li>
    <li>@Html.ActionLink("UsersAdmin", "Index", "UsersAdmin")</li>
    <li>@Html.ActionLink("GroupsAdmin", "Index", "GroupsAdmin")</li>
  }
</ul>
@Html.Partial("_LoginPartial")
```



```
// ...Other View Code After...
```

Adding Views for the GroupsAdminController

We're going to step through this fairly quickly, since there isn't a lot of discussion needed around View template code. Obviously, we need a view for each of our GroupsAdminController action methods.

Since Visual Studio won't generate quite what we need using the Add View Context Menu command, we'll do this manually.

Add a folder to the Views directory named *GroupsAdmin*. Now add the following Views:

The GroupsAdmin Index.cshtml View:

```
@model IEnumerable<IdentitySample.Models.ApplicationGroup>
@{
    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.Description)
        </th>
    </th></th>
</tr>

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

```

The GroupsAdmin Create.chnl View:

```

@model IdentitySample.Models.ApplicationGroup

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">

```

```

<h4>ApplicationGroup</h4>
<hr />
@Html.ValidationSummary(true)
<div class="form-group">
    @Html.LabelFor(model => model.Name, new { @class = "control-label" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.Name)
        @Html.ValidationMessageFor(model => model.Name)
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.Description, new { @class = "control-label" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.Description)
        @Html.ValidationMessageFor(model => model.Description)
    </div>
</div>
<div class="form-group">
    <label class="col-md-2 control-label">
        Select Group Roles
    </label>
    <div class="col-md-10">
        @foreach (var item in (SelectList)ViewBag.RolesList)
        {
            <div>
                <input type="checkbox" name="SelectedRoles" value="@item.Value" />
                @Html.Label(item.Text, new { @class = "control-label" })
            </div>
        }
    </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 GroupsAdmin Edit.cshtml View:

```

@model IdentitySample.Models.GroupViewModel
@{
    ViewBag.Title = "Edit";
}
<h2>Edit</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>ApplicationGroup</h4>
        <hr />
        @Html.ValidationSummary(true)
        @Html.HiddenFor(model => model.Id)

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

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

```

```

        <div class="col-md-10">
            @Html.EditorFor(model => model.Description)
            @Html.ValidationMessageFor(model => model.Description)
        </div>
    </div>
    <div class="form-group">
        @Html.Label("Permissions", new { @class = "control-label col-m
        <span class=" col-md-10">
            @foreach (var item in Model.RolesList)
            {
                <div>
                    <input type="checkbox" name="selectedRoles" value=
                    @Html.Label(item.Text, new { @class = "control-lab
                </div>
            }
        </span>
    </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 GroupsAdmin Details.cshtml View:

```
@model IdentitySample.Models.ApplicationGroup
```

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

<h2>Details</h2>

<div>
    <h4>ApplicationGroup</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>

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

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

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

    </dl>
</div>

<h4>List of permissions granted this group</h4>
@if (ViewBag.PermissionsCount == 0)
{
    <hr />
    <p>No users found in this role.</p>
}

<table class="table">
```

```

@foreach (var item in ViewBag.RolesList)
{
    <tr>
        <td>
            @item
        </td>
    </tr>
}
</table>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
    @Html.ActionLink("Back to List", "Index")
</p>

```

The GroupsAdmin Delete.cshtml View:

```

@model IdentitySample.Models.ApplicationGroup

@{
    ViewBag.Title = "Delete";
}

<h2>Delete</h2>

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

```

Updating the UserAdmin Views

We need to make a few minor changes to the UserAdmin Views. In the existing project, the UserAdmin Create and Edit Views allow us to assign Roles to the user. Instead, we want to assign the User to one or more Groups. Update the Create and Edit Views as follows:

Pay close attention when modifying the view code, and the names of Viewbag properties matter.

The Modified UserAdmin Create.cshtml View:

```

@model IdentitySample.Models.RegisterViewModel
@{

```



```

        ViewBag.Title = "Create";
    }

<h2>@ViewBag.Title.</h2>

@using (Html.BeginForm("Create", "UsersAdmin", FormMethod.Post, new { @class = "form" })
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-error" })
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <label class="col-md-2 control-label">
            Select User Groups
        </label>
        <div class="col-md-10">
            @foreach (var item in (SelectList)ViewBag.GroupsList)
            {
                <div>
                    <input type="checkbox" name="selectedGroups" value="@item.Value" />
                    @Html.Label(item.Text, new { @class = "control-label" })
                </div>
            }
        </div>
    </div>
}

```

```

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

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

```

The highlighted area calls out the primary impacted code.

Next, we will similarly modify the *UsersAdmin => Edit.cshtml* file.

The Modified UsersAdmin Edit.cshtml View:

```

@model IdentitySample.Models.EditUserViewModel

@{
    ViewBag.Title = "Edit";
}

<h2>Edit.</h2>

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

    <div class="form-horizontal">
        <h4>Edit User Form.</h4>

```

```

<hr />
@Html.ValidationSummary(true)
@Html.HiddenFor(model => model.Id)

<div class="form-group">
    @Html.LabelFor(model => model.Email, new { @class = "control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        @Html.ValidationMessageFor(model => model.Email)
    </div>
</div>
<div class="form-group">
    @Html.Label("This User belongs to the following Groups", new { @class = "control-label" })
    <span class="col-md-10">
        @foreach (var item in Model.GroupsList)
        {
            <div>
                <input type="checkbox" name="selectedGroups" value="@item.Id" />
                @Html.Label(item.Text, new { @class = "control-label" })
            </div>
        }
    </span>
</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 UsersAdmin *Details.cshtml* View displays a list of Roles assigned to the currently selected User. We will instead display a list of Groups:

The Modified UsersAdmin Details.cshtml View:

```
@model IdentitySample.Models.ApplicationUser
@{
    ViewBag.Title = "Details";
}

<h2>Details.</h2>

<div>
    <h4>User</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.UserName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.UserName)
        </dd>
    </dl>
</div>
<h4>List of Groups this user belongs to:</h4>
@if (ViewBag.GroupNames.Count == 0)
{
    <hr />
    <p>No Groups found for this user.</p>
}

<table class="table">
    @foreach (var item in ViewBag.GroupNames)
    {
```

```

        <tr>
            <td>
                @item
            </td>
        </tr>
    }
</table>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
    @Html.ActionLink("Back to List", "Index")
</p>

```

Of course, there is more we could do with the Views in this project, and we could add a few more for displaying User effective permissions (the sum permissions a user holds by virtue of membership in the assigned Groups) for example. For now, though, we'll just get things working, and then you can fine-tune things to the needs of your project.

Modify the Identity.Config File and the DbInitializer

In the previous project, when we set up Group-Based Permissions Management under Identity 1.0, we used EF Migrations to perform the database creation and Code-First generation. This time around, we are going to continue using the DbInitializer from the IdentitySamples project.

The `ApplicationDbInitializer` is defined in `App_Start => IdentityConfig.cs`. I have set it up for development purposes to inherit from `DropCreateDatabaseAlways`. However, you can easily change this to `DropCreateDatabaseIfModelChanges`.

Of course, we want our application to run with a basic configuration ready to go. Currently, the DbInitializer sets things up by creating a default User, an Admin Role, and then assigns the default user to that role.

We want to create the same default user, but then also create a default group. Then, we will create the default Admin role and assign it to the default group. Finally, we'll add the user to that group.

Open the *App_Start => Identity.Config* file, and make the following changes to the `InitializeIdentityForEF()` Method in the `ApplicationDbInitializer` class:

The InitializeIdentityForEF Method:

```
public static void InitializeIdentityForEF(ApplicationDbContext db) {
    var userManager = HttpContext.Current
        .GetOwinContext().GetUserManager<ApplicationUserManager>();
    var roleManager = HttpContext.Current
        .GetOwinContext().Get<ApplicationRoleManager>();
    const string name = "admin@example.com";
    const string password = "Admin@123456";
    const string roleName = "Admin";

    //Create Role Admin if it does not exist
    var role = roleManager.FindByName(roleName);
    if (role == null) {
        role = new ApplicationRole(roleName);
        var roleresult = roleManager.Create(role);
    }

    var user = userManager.FindByName(name);
    if (user == null) {
        user = new ApplicationUser
        {
```

```
        UserName = name,  
        Email = name,  
        EmailConfirmed = true  
    };  
    var result = userManager.Create(user, password);  
    result = userManager.SetLockoutEnabled(user.Id, false);  
}  
  
var groupManager = new ApplicationGroupManager();  
var newGroup = new ApplicationGroup("SuperAdmins", "Full Access to All");  
  
groupManager.CreateGroup(newGroup);  
groupManager.SetUserGroups(user.Id, new string[] { newGroup.Id });  
groupManager.SetGroupRoles(newGroup.Id, new string[] { role.Name });  
}
```

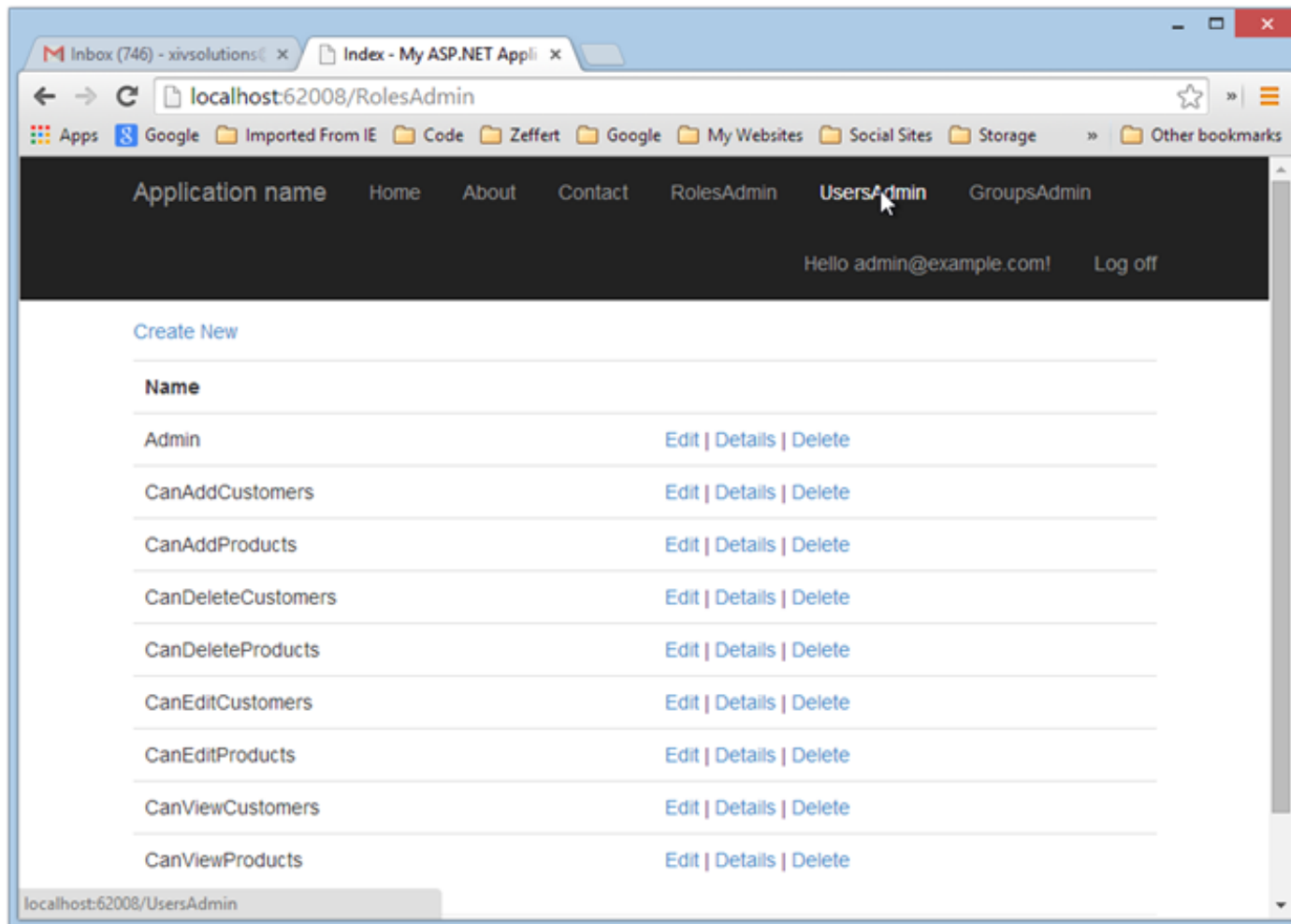
With that, we should be ready to run the application.

Running the Application

Once we've started the application and logged in, we should be able to navigate through the various admin functions. To make things interesting, let's add a few Roles, Groups, and Users and see what's what:

If we have added some additional Roles, Users, and Groups, we begin to see how this might work in the context of a real-world application. Let's take two fictitious departments, Sales and Purchasing. We might have some Roles at a relatively granular level (perhaps, at the level of our basic controller Actions) for each function:

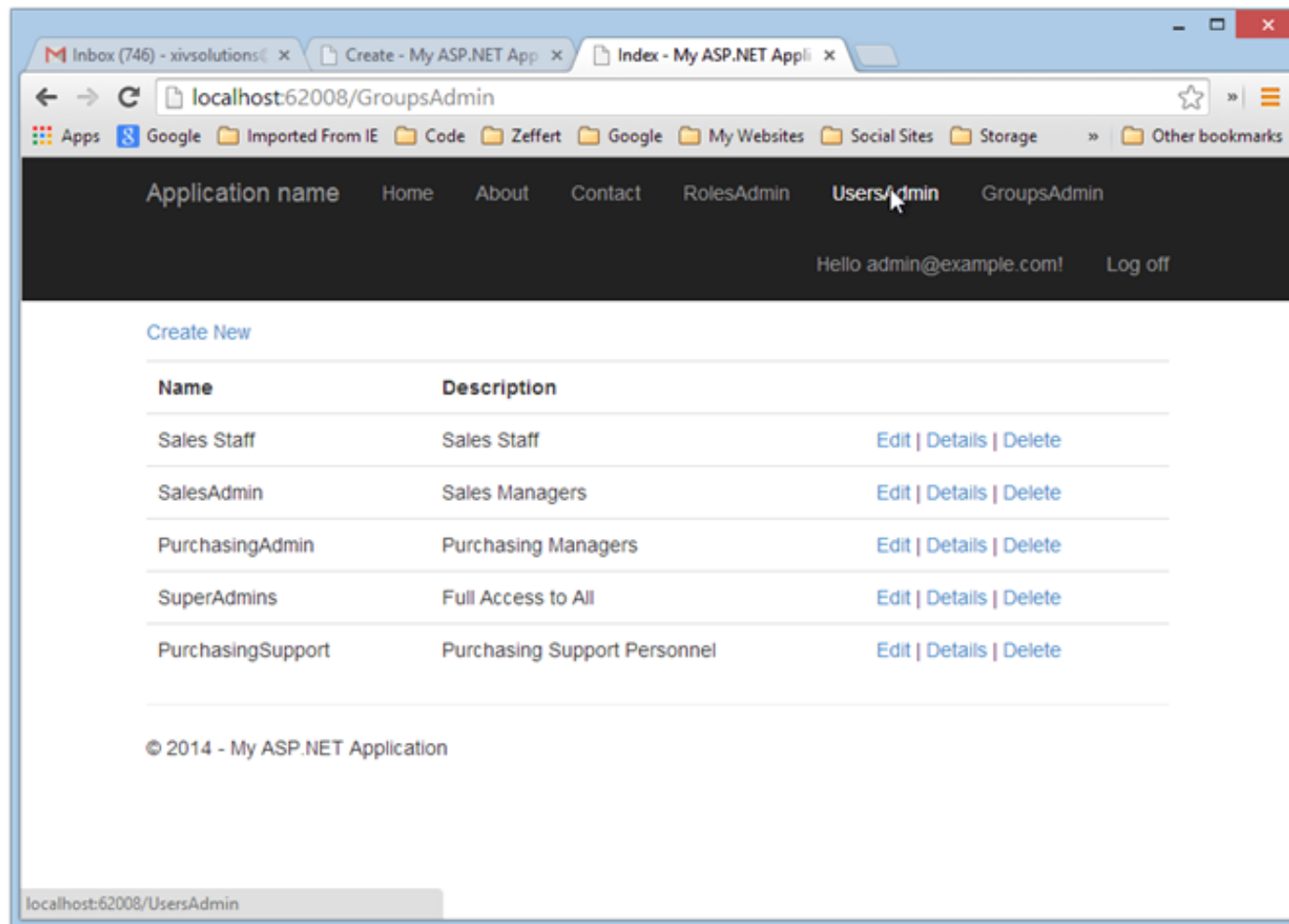
Basic Roles for Sales and Purchasing Departments



Now, this, and our other views could most likely use some help from a designer, or at least some presentation controls which would allow for better groupings, or something. Nonetheless, we see we have added some Roles here which correspond roughly with hypothetical controller actions we might find on a **CustomersController** and a **ProductsController** (OK – we’re simplifying a little for the purpose of this example, but you get the idea).

Now, we might also define some Groups:

Basic Groups for Sales and Purchasing:



Again, if we were worried about design aesthetics here, we might find our Groups management View a little lacking. But you can see that we have defined a few Groups related to the functions of the Sales and Purchasing departments.

Now, if we edit one of these groups, we can assign the appropriate Roles to the group:

Assigning Group Roles When Editing Group:

Inbox (746) - xivolutions x Create - My ASP.NET Ap x Edit - My ASP.NET Applic x Facebook x

localhost:62008/GroupsAdmin/Edit/c9d4adc7-4f53-4439-96d1-c34a4e808c95

Apps Google Imported From IE Code Zeffert Google My Websites Social Sites Storage Other bookmarks

Application name Home About Contact **RolesAdmin** UsersAdmin GroupsAdmin

Hello admin@example.com! Log off

ApplicationGroup

Name

SalesAdmin

Description

Sales Managers

Permissions

- ☐ Admin
- ☒ CanAddCustomers
- ☐ CanAddProducts
- ☒ CanDeleteCustomers
- ☒ CanDeleteProducts
- ☒ CanEditCustomers
- ☐ CanEditProducts
- ☒ CanViewCustomers
- ☒ CanViewProducts

Save

localhost:62008/RolesAdmin

Here, we have decided that users in the SalesAdmin Group should be able to add/edit/view/delete Customer data, as well as view (but not modify) product data.

John Atten



Now, if we save the SalesAdmin Group like this, we can then assign one or more users, who will then get all of the permissions associated with this group.

Creating a New User with Group Assignments:

Inbox (746) - xivolutions x Create - My ASP.NET App x Create - My ASP.NET App x Facebook x

localhost:62008/UsersAdmin/Create

Apps Google Imported From IE Code Zeffert Google My Websites Social Sites Storage Other bookmarks

Application name Home About Contact RolesAdmin UsersAdmin GroupsAdmin

Hello admin@example.com! Log off

Create a new account.

Email
jim@example.com

Password

Confirm password

Select User Group

- ☐ Sales Staff
- ☒ SalesAdmin
- ☐ PurchasingAdmin
- ☐ SuperAdmins
- ☐ PurchasingSupport

Create

© 2014 - My ASP.NET Application

Once we save, Jim will be a member of the SalesAdmin Group, and will have all of the Role permissions we assigned to that group.

Controlling Access with the [Authorize] Attribute

Of course, none of this does us any good if we don't put these granular Role permissions to use.

First off, we probably want to add some access control to our

`GroupsAdminController` itself. We might want to add an `[Authorize]` decoration to the entire controller, similar to the `UserAdminController`, so that only those with the Admin Role can modify Groups.

Beyond that, though, let's expand upon our example above. Consider a hypothetical `CustomersController`. We might decorate the basic controller methods as follows, in keeping with the Roles we defined:

Hypothetical CustomersController:

```
public class CustomerController
{
    [Authorize(Roles = "CanViewCustomers")]
    public async ActionResult Index()
    {
        // Code...
    }

    [Authorize(Roles = "CanAddCustomers")]
    public async ActionResult Create()
    {
        // Code...
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
```

```
[Authorize(Roles = "CanAddCustomers")]
public async Task<ActionResult> Create(SomeArguments)
{
    // Code...
}
```

```
[Authorize(Roles = "CanEditCustomers")]
public async ActionResult Edit(int id)
{
    // Code...
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
[Authorize(Roles = "CanEditCustomers")]
public async Task<ActionResult> Edit(SomeArguments)
{
    // Code...
}
```

```
[Authorize(Roles = "CanDeleteCustomers")]
public async Task<ActionResult> Delete(id)
{
    // Code...
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
[Authorize(Roles = "CanDeleteCustomers")]
public async Task<ActionResult> Delete(SomeArguments)
{
    // Code...
}
```

```
}
```

We can see that we have now implemented some authorization control which corresponds to some of the Roles we defined. The roles themselves are not specific to any one type of user. instead, Roles can be assigned to different Groups. From there, it is a simple matter to add or remove users from one or more groups, depending upon their function and access requirements.

Some Thoughts on Authorization and Security

The concepts in this project are sort of a middle ground between the basic, but highly functional authorization mechanism offered by Identity 2.0 out of the box, and more complex implementations using Claims or Active Directory.

The system illustrated here will afford a more granular control over Roles and Authorization to access and execute code. However, there is a practical limit.

Designing a robust application authorization matrix requires careful thought, and choosing the correct tools (as with most things in development). Planning carefully upfront will go a long ways towards helping you create a solid, maintainable application.

If you get too fine-grained in your Authorizations and Role definitions, you will have a difficult to manage mess. Not going granular enough results in a clunky, limited authorization scheme where you may find yourself giving users too much or too little access.

Errata, Ideas, and Pull Requests

It is entirely possible I have missed something in putting the code together for this article. Also, as noted, there is plenty of room for improvement in the design here. If you find something broken, have an idea to improve the concept, or (Especially) would like to improve the Views with respect to organizing Roles, Groups, etc, please open an issue or send me a Pull Request.

Additional Resources and Items of Interest

- [Complete Source Code for this Project on Github](#)
- [Source Code for the Extensible Template on Which this Project was Built](#)
- [ASP.NET MVC and Identity 2.0: Understanding the Basics](#)
- [ASP.NET Identity 2.0: Customizing Users and Roles](#)
- [Original Identity 1.0 Project: Implementing Group-Based Permissions Management](#)
- [ASP.NET Identity 2.0 Extending Identity Models and Using Integer Keys Instead of Strings](#)
- [Routing Basics in ASP.NET MVC](#)
- [Customizing Routes in ASP.NET MVC](#)



RELATED ARTICLES

C#

Extending C# Listview with
Collapsible Groups (Part I)

C#

Working with Pdf Files in C#
Using PdfBox and IKVM



VIEW COMMENTS



PREVIOUS POST

[ASP.NET Identity 2.0: Extensible Template Projects](#)



NEXT POST

[C#: Building a Useful, Extensible .NET Console Application Template for
Development and Testing](#)

Copyright © John Atten. 2016 • All rights reserved.