ASP.NET

ASP.NET Identity 2.0 Extending Identity Models and Using Integer Keys Instead of Strings

JOHN ATTEN JULY 13, 2014 24

The ASP.NET Identity framework was released to manufacture on March 20 2014, bringing with it a slew of long-awaited enhancements, delivering a fully-formed authentication and authorization platform to the ASP.NET developer community.

In previous posts, we have taken a broad look at the

structure of the new framework, and how it differs from the 1.0 release. We've also walked through implementing email account confirmation and two-factor authentication, as well as extending the basic User and Role models (which requires a bit more effort than you might think).

- 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

In this post we're going take a deeper look at extending the core set of models afforded by the Identity 2.0 framework, and re-implementing the basic Identity Samples project using integer keys for all of our models, instead of the default string keys which are the default.

Source Code on Github

In the course of this article, we will basically re-implement the Identity Samples project with integer keys. you can clone the **completed source code from my Github repo**. Also, if you find bugs and/or have suggestions, please do open an issue and/or shoot me a pull request!

- Why Does ASP.NET Identity Use String Keys

in the First Place?
- Identity 2.0 Core Classes use Generic Type Arguments
- Implementing Integer Keys Using Identity 2.0 and the Identity Samples Project
- Re-Engineering the Basic Identity Models
- Cookie Authentication Configuration
- Update Admin View Models
- Update Controller Method Parameter Arguments
- Add Integer Type Argument to GetUserId() Calls
- Update Roles Admin Views
- A Note on Security

Why Does ASP.NET Identity Use String Keys in the First Place?

A popular, and somewhat confounding question is "why did the Identity team choose string keys as the default for the Identity framework models? Many of us who grew up using databases tend towards easy, auto-incrementing integers as database primary keys, because it's easy, and at least in theory, there are some performance advantages with respect to table indexes and such.

The decision of the Identity Team to use strings as keys is best summarized in a Stack Overflow answer by Rick Anderson, writer for ASP.NET at Microsoft:

> 1. The Identity runtime prefers strings for the user ID because we don't want to be in the business of figuring out proper serialization of the user IDs (we use strings for claims as well for the same reason), e.g. all (or most) of the Identity interfaces refer to user ID as a string.
> 2. People that customize the persistence layer, e.g. the entity types, can choose whatever type they want for keys, but then they own providing us with a string representation of the keys.
> 3. By default we use the string representation of GUIDs for each new user, but that is just because it provides a very easy way for us to automatically generate unique IDs.

The decision is not without its detractors in the community. The default string key described above is essentially a string representation of a Guid. As this discussion on Reddit illustrates, there is contention about the performance aspects of this against a relational database backend.

The concerns noted in the Reddit discussion focus mainly on database index performance, and are unlikely to be an issue for a large number of smaller

sites and web applications, and particularly for learning projects and students. However, as noted previously, for many of us, the auto-incrementing integer is the database primary key of choice (even in cases where it is not the BEST choice), and we want our web application to follow suit.

Identity 2.0 Core Classes use Generic Type Arguments

As we discussed in the post on customizing ASP.NET Identity 2.0 Users and Roles, the framework is built up from a structure of generic Interfaces and base classes. At the lowest level, we find interfaces, such as `IUser<TKey>` and `IRole<TKey>`. These, and related Interfaces and base classes are defined in the `Microsoft.AspNet.Identity.Core library`.

Moving up a level of abstraction, we can look at the `Microsoft.AspNet.Identity.EntityFramework` library, which uses the components defined in `…Identity.Core` to build the useful, ready-to-use classes commonly used in applications, and in particular by the Identity Samples project we have been using to explore Identity 2.0.

The *"…Identity.EntityFramework library gives us some Generic base classes, as well as a default concrete implementation for each. For example, Identity.EntityFramework gives us the following generic base implementation for a class IdentityRole…":*

Generic Base for IdentityRole:

```
public class IdentityRole<TKey, TUserRole> : IRole
where TUserRole : IdentityUserRole<TKey>
{
    public TKey Id { get; set; }
    public string Name { get; set; }
    public ICollection<TUserRole> Users { get; set
    public IdentityRole()
    {
        this.Users = new List<TUserRole>();
    }
}
```

As we can see, the above defines `IdentityRole` in terms of generic type arguments for the key and `UserRole`, and must implement the interface `IRole<TKey>`. Note that Identity defines both an `IdentityRole` class, as well as an `IdentityUserRole` class, both of which are required to make things work. More on this later.

The Identity team also provides what amounts to a default implementation of this class:

Default Implementation of IdentityRole with non-generic type arguments:

```
public class IdentityRole : IdentityRole<string, I
{
    public IdentityRole()
    {
        base.Id = Guid.NewGuid().ToString();
    }


    public IdentityRole(string roleName) : this()
    {
        base.Name = roleName;
    }
}
```

Notice how the default implementation class is defined in terms of a `string` key and a specific implementation of `IdentityUserRole`?

This means that we can only pass strings as keys, and in fact the `IdentityRole` model will be defined in our database with a string-type primary key. It also means that the specific, non-generic implementation of `IdentityUserRole` will be what is passed to the type argument into the base class.

If we steal a page from the previous post, and take a look at the default type definitions provided by Identity 2.0, we find the following (it's not exhaustive, but these are what we will be dealing with later):

Default Identity 2.0 Class Signatures with Default Type Arguments:

```
public class IdentityUserRole
    : IdentityUserRole<string> {}

public class IdentityRole
    : IdentityRole<string, IdentityUserRole> {}

public class IdentityUserClaim
    : IdentityUserClaim<string> {}

public class IdentityUserLogin
    : IdentityUserLogin<string> {}

public class IdentityUser
    : IdentityUser<string, IdentityUserLogin,
        IdentityUserRole, IdentityUserClaim>, IUse

public class IdentityDbContext
    : IdentityDbContext<IdentityUser, IdentityRole
        IdentityUserLogin, IdentityUserRole, Ident

public class UserStore<TUser>
    : UserStore<TUser, IdentityRole, string, Ident
```

```
        IdentityUserRole, IdentityUserClaim>,
        IUserStore<TUser>, IUserStore<TUser, strin
    where TUser : IdentityUser {}

public class RoleStore<TRole>
    : RoleStore<TRole, string, IdentityUserRole>,
        IQueryableRoleStore<TRole, string>, IRoleS
    where TRole : IdentityRole, new() {}
```

We can see that, starting with `IdentityUserRole`, the types are defined with string keys, and as importantly, progressively defined in terms of the others. This means that if we want to use integer keys instead of string keys for all of our models (and corresponding database tables), we need to basically implement our own version of the stack above.

Implementing Integer Keys Using Identity 2.0 and the Identity Samples Project

As in previous posts, we are going to use the Identity Samples project as our base for creating an Identity 2.0 MVC application. The Identity team has put together the Identity Samples project primarily (I assume) as a demonstration platform, but in fact it contains everything one might need (after a few tweaks, anyway) in order to build out a complete ASP.NET MVC project using the Identity 2.0 framework.

The concepts we are going to look at here apply equally well if you are building up your own Identity-based application from scratch. The ways and means

might vary according to your needs, but in general, much of what we see here will apply whether you are starting from the Identity Samples project as a base, or "rolling your own" so to speak.

The important thing to bear in mind is that the generic base types and interfaces provided by Identity framework allow great flexibility, but also introduced complexity related to the dependencies introduced by the generic type arguments. In particular, the type specified as the key for each model must propagate through the stack, or the compiler gets angry.

Getting Started – Installing the Identity Samples Project

The Identity Samples project is available on Nuget. First, create an empty ASP.NET Web Project (**It is important that you use the "Empty" template here**, not MVC, not Webforms, EMPTY). Then open the Package Manager console and type:

Install Identity Samples from the Package Manager Console:

```
PM> Install-Package Microsoft.AspNet.Identity.Samp
```

This may take a minute or two to run. When complete, your will see a basic ASP.NET MVC project in the VS Solution Explorer. Take a good look around the Identity 2.0 Samples project, and become familiar with what things are and where they are at.

Re-Engineering the Basic Identity Models

To get started, we need to re-engineer the basic model classes defined in the Identity Samples project, as well as add a few new ones. Because Identity Samples uses string-based keys for entity models, the authors, in many cases get away with depending upon the default class implementations provided by the framework itself. Where they extend, they extend from the default classes, meaning the string-based keys are still baked in to the derived classes.

Since we want to use integer keys for all of our models, we get to provide our own implementations for most of the models.

In many cases, this isn't as bad as it sounds. For example, there are a handful of model classes we need only define in terms of the generic arguments, and from there the base class implementation does the rest of the work.

NOTE: *As we proceed to modify/add new classes here, the error list in Visual Studio will begin to light up like a Christmas tree until we are done. Leave that be for the moment. If we do this correctly, there should be no errors left when we finish. IF there are, they will help us find things we missed.*

In the *Models => IdentityModels.cs* file, we find the model classes used by the Identity Samples

application. To get started, we are going to add our own definitions for `IndentityUserLogin`, `IdentityUserClaim`, and `IdentityUserRole`. The Identity Samples project simply depended upon the default framework implementations for these classes, and we need our own integer based versions. Add the following to the *IdentityModels.cs* file:

Integer-Based Definitions for UserLogin, UserClaim, and UserRole:

```
public class ApplicationUserLogin : IdentityUserLo
public class ApplicationUserClaim : IdentityUserCl
public class ApplicationUserRole : IdentityUserRol
```

Now, with that out of the way, we can define our own implementation of `IdentityRole`. The Samples project also depended upon the framework version for `IdentityRole`, and we are going to provide our own again. This time, though, there's a little more to it:

Integer-Based Definition for IdentityRole:

```
public class ApplicationRole : IdentityRole<int, A
{
    public string Description { get; set; }

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

    public ApplicationRole(string name, string des
        : this(name)
    {
        this.Description = description;
    }
}
```

Notice above, we have defined `ApplicationRole` in terms of an integer key, and also in terms of our custom class `ApplicationUserRole` ? This is important, and will continue on up the stack as we re-implement the Identity classes we need for the Identity Samples project to run as expected.

Next, we are going to modify the existing definition for `ApplicationUser` . Currently, the *IdentitySamples.cs* file includes a fairly simple definition for `ApplicationUser` which derives from the default `IdentityUser` class provided by the framework, which requires no type arguments because they have already been provided in the default implementation. We need to basically re-define `ApplicationUser` starting from the ground up.

The existing `ApplicationUser` class in the *IdentityModels.cs* file looks like this:

Existing ApplicationUser Class in IdentityModels.cs:

```
public class ApplicationUser : IdentityUser
{
    public async Task<ClaimsIdentity>
        GenerateUserIdentityAsync(UserManager<Appl
    {
        var userIdentity = await manager
            .CreateIdentityAsync(this, DefaultAuth
        return userIdentity;
    }
}
```

We need to **replace the above in its entirety** with the following:

Custom Implementation for ApplicationUser:

```
public class ApplicationUser
: IdentityUser<int, ApplicationUserLogin,
    ApplicationUserRole, ApplicationUserClaim>, IU
{
    public async Task<ClaimsIdentity>
        GenerateUserIdentityAsync(UserManager<Appl
    {
        var userIdentity = await manager
            .CreateIdentityAsync(this, DefaultAuth
        return userIdentity;
    }
}
```

Once again, instead of deriving from the default Identity framework implementation for `IdentityUser`, we have instead used the generic base, and provided our own custom type arguments. Also again, we have defined our custom `ApplicationUser` in terms of an integer key, and our own custom types.

Modified Application Db Context

Also in the *IdentityModels.cs* file is an `ApplicationDbContext` class.

Now that we have built out the basic models we are going to need, we also need to re-define the `ApplicationDbContext` in terms of these new models. As previously, the existing `ApplicationDbContext` used in the Identity Samples application is expressed only in terms of `ApplicationUser`, relying (again) upon the default concrete implementation provided by the framework.

If we look under the covers, we find the `ApplicationDbContext<ApplicationUser>` actually inherits from `IdentityDbContext<ApplicationUser>`, which in turn is derived from:

`IdentityDbContext<TUser, IdentityRole, string, IdentityUserLogin,`

IdentityUserRole, IdentityUserClaim>

where TUser : Microsoft.AspNet.Identity.EntityFramework.IdentityUser

In other words, we once again have a default concrete implementation which is defined in terms of the other default framework types, all of which further depend upon a string-based key.

In order to define a `DbContext` which will work with our new custom types, we need to express our concrete class in terms of integer keys, and our own custom derived types.

Replace the existing `ApplicationDbContext` code with the following:

Modified ApplicationDbContext:

```
public class ApplicationDbContext
    : IdentityDbContext<ApplicationUser, Applicati
    ApplicationUserLogin, ApplicationUserRole, App
{
    public ApplicationDbContext()
        : base("DefaultConnection")
    {
    }
```

```
    static ApplicationDbContext()
    {
        Database.SetInitializer<ApplicationDbConte
    }

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

Once again, we have now expressed `ApplicationDbContext` in terms of our own custom types, all of which use an integer key instead of a string.

Custom User and Role Stores

I am willing to bet that if you take a look at the Visual Studio Error Window right now, it is likely a block of what seems to be endless red error indicators. As mentioned previously, that's fine for now – ignore it.

Identity framework defines the notion of User and Role stores for accessing user and role information. As with most everything else to this point, the default framework implementations for UserStore and RoleStore are defined in terms of the other default classes we have seen to this point – in other words, they won't work with our new custom classes. We need to express a custom User store, and a custom Role store, in terms of integer keys and our own custom classes.

Add the following to the *IdentityModels.cs* file:

Adding a Custom User Store:

```
public class ApplicationUserStore :
    UserStore<ApplicationUser, ApplicationRole
    ApplicationUserLogin, ApplicationUserRole,
    ApplicationUserClaim>, IUserStore<Applicat
    IDisposable
{
    public ApplicationUserStore() : this(new I
    {
        base.DisposeContext = true;
    }

    public ApplicationUserStore(DbContext cont
        : base(context)
    {
    }
}


public class ApplicationRoleStore
    : RoleStore<ApplicationRole, int, Applicat
    IQueryableRoleStore<ApplicationRole, int>,
    IRoleStore<ApplicationRole, int>, IDisposa
{
    public ApplicationRoleStore()
        : base(new IdentityDbContext())
    {
        base.DisposeContext = true;
    }

    public ApplicationRoleStore(DbContext cont
        : base(context)
    {
    }
}
```

Re-Engineering Identity Configuration Classes

The Identity Samples project includes a file named *App_Start => IdentityConfig.cs*. In this file is a bunch of code which basically configures the Identity System for use in your application. The changes we introduced on

our *IdentityModels.cs* file will cause issues here (and basically, throughout the application) until they are addressed in the client code.

In most cases, we will either be replacing a reference to a default Identity class with one of our new custom classes, and/or calling method overrides which allow the passing of custom type arguments.

In the *IdentityConfig.cs* file, we find an `ApplicationUserManager` class, which contains code commonly called by our application to, well, manage users and behaviors. we will replace the existing code with the following, which essentially expresses `ApplicationUserManager` in terms of integer keys, and our new custom `UserStore`. If you look closely, we have added an int type argument to many of the method calls.

Customized ApplicationUserManager Class:

```
// *** PASS IN TYPE ARGUMENT TO BASE CLASS:
public class ApplicationUserManager : UserManager<
{
    // *** ADD INT TYPE ARGUMENT TO CONSTRUCTOR CA
    public ApplicationUserManager(IUserStore<Appli
        : base(store)
    {
    }

    public static ApplicationUserManager Create(
        IdentityFactoryOptions<ApplicationUserMana
        IOwinContext context)
    {
        // *** PASS CUSTOM APPLICATION USER STORE
        var manager = new ApplicationUserManager(
            new ApplicationUserStore(context.Get<A

        // Configure validation logic for username
```

```
// *** ADD INT TYPE ARGUMENT TO METHOD CAL
manager.UserValidator = new UserValidator<
{
    AllowOnlyAlphanumericUserNames = false
    RequireUniqueEmail = true
};

// Configure validation logic for password
manager.PasswordValidator = new PasswordVa
{
    RequiredLength = 6,
    RequireNonLetterOrDigit = true,
    RequireDigit = true,
    RequireLowercase = true,
    RequireUppercase = true,
};

// Configure user lockout defaults
manager.UserLockoutEnabledByDefault = true
manager.DefaultAccountLockoutTimeSpan = Ti
manager.MaxFailedAccessAttemptsBeforeLocko

// Register two factor authentication prov
// This application uses Phone and Emails
// code for verifying the user You can wri

// *** ADD INT TYPE ARGUMENT TO METHOD CAL
manager.RegisterTwoFactorProvider("PhoneCo
    new PhoneNumberTokenProvider<Applicati
{
    MessageFormat = "Your security code is
});

  // *** ADD INT TYPE ARGUMENT TO METHOD C
manager.RegisterTwoFactorProvider("EmailCo
    new EmailTokenProvider<ApplicationUser
{
    Subject = "SecurityCode",
    BodyFormat = "Your security code is {0
});

manager.EmailService = new EmailService();
manager.SmsService = new SmsService();
var dataProtectionProvider = options.DataP
if (dataProtectionProvider != null)
{
    // *** ADD INT TYPE ARGUMENT TO METHOD
    manager.UserTokenProvider =
        new DataProtectorTokenProvider<App
            dataProtectionProvider.Create(
}
```

```
        return manager;
    }
}
```

That's a lot of code there. Fortunately, modifying the `ApplicationRoleManager` class is not such a big deal. We're essentially doing the same thing – expressing `ApplicationRoleManager` in terms of integer type arguments, and our custom classes.

Replace the `ApplicationRoleManager` code with the following:

Customized ApplicationRoleManager Class:

```
// PASS CUSTOM APPLICATION ROLE AND INT AS TYPE AR
public class ApplicationRoleManager : RoleManager<
{
    // PASS CUSTOM APPLICATION ROLE AND INT AS TYP
    public ApplicationRoleManager(IRoleStore<Appli
        : base(roleStore)
    {
    }

    // PASS CUSTOM APPLICATION ROLE AS TYPE ARGUME
    public static ApplicationRoleManager Create(
        IdentityFactoryOptions<ApplicationRoleMana
    {
        return new ApplicationRoleManager(
            new ApplicationRoleStore(context.Get<A
    }
}
```

Modify The Application Database Initializer and Sign-in Manager

The `ApplicationDbInitializer` class is what manages the creation and seeding of the backing

database for our application. In this class we create a basic admin role user, and set up additional items such as the Email and SMS messaging providers.

The only thing we need to change here is where we initialize an instance of `ApplicationRole`. In the existing code, the `ApplicationDbInitializer` class instantiates an instance of `IdentityRole`, and we need to create an instance of our own `ApplicationRole` instead.

Replace the existing code with the following, or make the change highlighted below:

Modify the ApplicationDbInitializer Class:

```
public class ApplicationDbInitializer : DropCreate
{
    protected override void Seed(ApplicationDbCont
        InitializeIdentityForEF(context);
        base.Seed(context);
    }

    //Create User=Admin@Admin.com with password=Ad
    public static void InitializeIdentityForEF(App
        var userManager = HttpContext.Current.GetO
        var roleManager = HttpContext.Current.GetO
        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) {
            // *** INITIALIZE WITH CUSTOM APPLICAT
            role = new ApplicationRole(roleName);
            var roleresult = roleManager.Create(ro
        }

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

```
            result = userManager.SetLockoutEnabled
        }

        // Add user admin to Role Admin if not alr
        var rolesForUser = userManager.GetRoles(us
        if (!rolesForUser.Contains(role.Name)) {
            var result = userManager.AddToRole(use
        }
    }
}
```

Fixing up the `ApplicationSignInManager` is even more simple. Just change the `string` type argument in the class declaration to `int` :

Modify the ApplicationSignInManager Class:

```
// PASS INT AS TYPE ARGUMENT TO BASE INSTEAD OF ST
public class ApplicationSignInManager : SignInMana
{
    public ApplicationSignInManager(
        ApplicationUserManager userManager, IAuthe
        base(userManager, authenticationManager) {

    public override Task<ClaimsIdentity> CreateUse
    {
        return user.GenerateUserIdentityAsync((App
    }

    public static ApplicationSignInManager Create(
     IdentityFactoryOptions<ApplicationSignInManag
    {
        return new ApplicationSignInManager(contex
    }
}
```

Cookie Authentication Configuration

In the file *App_Start => Startup.Auth* there is a partial class definition, Startup. in the single method call

defined in the partial class, there is a call to `app.UseCookieAuthentication()`. Now that our application is using integers as keys instead of strings, we need to make a modification to the way the `CookieAuthenticationProvider` is instantiated.

The existing call to app. `UseCookieAuthentication` (found smack in the middle of the middle of the `ConfigureAuth()` method) needs to be modified. Where the code calls `OnVlidateIdentity` the existing code passes `ApplicationUserManager` and `ApplicationUser` as type arguments. What is not obvious is that this is an override which assumes a third, string type argument for the key (yep – we're back to that whole string keys thing again).

We need to change this code to call another override, which accepts a third type argument, and pass it an `int` argument.

The existing code looks like this:

Existing Call to app.UseCookieAuthentication:

```
app.UseCookieAuthentication(new CookieAuthenticati
{
    AuthenticationType = DefaultAuthenticationType
    LoginPath = new PathString("/Account/Login"),
    Provider = new CookieAuthenticationProvider
    {
        // Enables the application to validate the
        // This is a security feature which is use
        // password or add an external login to yo
        OnValidateIdentity = SecurityStampValidato
            .OnValidateIdentity<ApplicationUserMan
                validateInterval: TimeSpan.FromMin
                regenerateIdentity: (manager, user
                    => user.GenerateUserIdentityAs
    }
```

```
});
```

We need to modify this code in a couple of non-obvious ways. First, as mentioned above, we need to add a third type argument specifying that `TKey` is an int.

Less obvious is that we also need to change the name of the second argument from `regenerateIdentity` to `regenerateIdentityCallback`. Same argument, but different name in the overload we are using.

Also less than obvious is the third `Func` we need to pass into the call as `getUserIdCallback`. Here, we need to retreive a user id from a claim, which stored the Id as a string. We need to parse the result back into an `int`.

Replace the existing code above with the following:

Modified Call to app.UseCookieAuthentication:

```
app.UseCookieAuthentication(new CookieAuthenticati
{
    AuthenticationType = DefaultAuthenticationType
    LoginPath = new PathString("/Account/Login"),
    Provider = new CookieAuthenticationProvider
    {
        // Enables the application to validate the
        // This is a security feature which is use
        // password or add an external login to yo
        OnValidateIdentity = SecurityStampValidato
            // ADD AN INT AS A THIRD TYPE ARGUMENT
            .OnValidateIdentity<ApplicationUserMan
                validateInterval: TimeSpan.FromMin
                // THE NAMED ARGUMENT IS DIFFERENT
                regenerateIdentityCallback: (manag
                    => user.GenerateUserIdentityAs
                    // Need to add THIS line becau
                    getUserIdCallback:  (claim) =>
```

```
    }
});
```

With that, most of the Identity infrastructure is in place. Now we need to update a few things within our application.

Update Admin View Models

The *Models => AdminViewModels.cs* file contains class definitions for a `RolesAdminViewModel` and a `UsersAdminViewModel`. In both cases, we need to change the type of the Id property from string to int:

Modify the Admin View Models:

```
public class RoleViewModel
{
    // Change the Id type from string to int:
    public int Id { get; set; }

    [Required(AllowEmptyStrings = false)]
    [Display(Name = "RoleName")]
    public string Name { get; set; }
}

public class EditUserViewModel
{
    // Change the Id Type from string to int:
    public int Id { get; set; }

    [Required(AllowEmptyStrings = false)]
    [Display(Name = "Email")]
    [EmailAddress]
    public string Email { get; set; }

    public IEnumerable<SelectListItem> RolesList {
}
```

Update Controller Method Parameter Arguments

A good many of the controller action methods currently expect an id argument of type string. We need to go through all of the methods in our controllers and change the type of the id argument from string to int.

In each of the following controllers, we need to change the existing Id from string to int as shown for the action methods indicated (we're only showing the modified method signatures here):

Account Controller:

```
public async Task<ActionResult> ConfirmEmail(int u
```

Roles Admin Controller:

```
public async Task<ActionResult> Edit(int id)
public async Task<ActionResult> Details(int id)
public async Task<ActionResult> Delete(int id)
public async Task<ActionResult> DeleteConfirmed(in
```

Users Admin Controller:

```
public async Task<ActionResult> Details(int id)
public async Task<ActionResult> Edit(int id)
public async Task<ActionResult> Delete(int id)
public async Task<ActionResult> DeleteConfirmed(in
```

Update the Create Method on Roles Admin Controller

Anywhere we are creating a new instance of a Role, we need to make sure we are using our new `ApplicationRole` instead of the default `IdentityRole`. Specifically, in the `Create()` method of the `RolesAdminController`:

Instantiate a new ApplicationRole Instead of IdentityRole:

```
[HttpPost]
public async Task<ActionResult> Create(RoleViewMod
{
    if (ModelState.IsValid)
    {
        // Use ApplicationRole, not IdentityRole:
        var role = new ApplicationRole(roleViewMod
        var roleresult = await RoleManager.CreateA
        if (!roleresult.Succeeded)
        {
            ModelState.AddModelError("", roleresul
            return View();
        }
        return RedirectToAction("Index");
    }
    return View();
}
```

Add Integer Type Argument to GetUserId() Calls

If we take a look at our Error list now, we see the preponderance of errors are related to calls to `User.Identity.GetUserId()`. If we take a closer look at this method, we find that once again, the default version of `GetUserId()` returns a string, and that there is an overload which accepts a type argument which determines the return type.

Sadly, calls to `GetUserId()` are sprinkled liberally throughout `ManageController`, and a few places in `AccountController` as well. We need to change all of the calls to reflect the proper type argument, and the most efficient way to do this is an old fashioned Find/Replace.

Fortunately, you can use Find/Replace for the entire document on both `ManageController` and `AccountController`, and get the whole thing done in one fell swoop. Hit Ctrl + H, and in the "Find" box, enter the following:

Find all instances of:

```
Identity.GetUserId()
```

Replace with:

```
Identity.GetUserId<int>()
```

If we've done this properly, most of the glaring red errors in our error list should now be gone. There are a few stragglers, though. In these cases, we need to counter-intuitively convert the int Id back into a string.

Return a String Where Required

There are a handful of methods which call to `GetUserId()`, but regardless of the type the Id represents (in our case, now, an `int`) want a string representation of the Id passed as the argument. All of these methods are found on `ManageController`, and in each case, we just add a call to `.ToString()`.

First, in the `Index()` method of `ManageController`, we find a call to `AuthenticationManager.TwoFactorBrowserRemembered()`. Add the call to `.ToString()` after the call to `GetUserId()`:

TwoFactorBrowserRemembered:

```
public async Task<ActionResult> Index(ManageMessag
{
    ViewBag.StatusMessage =
        message == ManageMessageId.ChangePasswordS
            "Your password has been changed."
        : message == ManageMessageId.SetPasswordSu
            "Your password has been set."
        : message == ManageMessageId.SetTwoFactorS
            "Your two factor provider has been set
        : message == ManageMessageId.Error ?
            "An error has occurred."
        : message == ManageMessageId.AddPhoneSucce
            "The phone number was added."
        : message == ManageMessageId.RemovePhoneSu
            "Your phone number was removed."
        : "";

    var model = new IndexViewModel
    {
        HasPassword = HasPassword(),
        PhoneNumber = await UserManager.GetPhoneNu
        TwoFactor = await UserManager.GetTwoFactor
        Logins = await UserManager.GetLoginsAsync(

        // *** Add .ToString() to call to GetUserI
        BrowserRemembered = await AuthenticationMa
            .TwoFactorBrowserRememberedAsync(User.
    };
    return View(model);
}
```

Similarly, do the same for the `RememberBrowser` method, also on `ManageController`:

Add Call to ToString() to RememberBrowser Method:

```
[HttpPost]
public ActionResult RememberBrowser()
{
    var rememberBrowserIdentity = AuthenticationMa
        .CreateTwoFactorRememberBrowserIdentity(
            // *** Add .ToString() to call to GetU
            User.Identity.GetUserId<int>().ToStrin
    AuthenticationManager.SignIn(
        new AuthenticationProperties { IsPersisten
        rememberBrowserIdentity);
    return RedirectToAction("Index", "Manage");
}
```

Lastly,the same for the `LinkLogin()` and `LinkLoginCallback()` methods:

Add Call to ToString() to LinkLogin():

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult LinkLogin(string provider)
{
    return new AccountController
        .ChallengeResult(provider, Url.Action("Lin
            // *** Add .ToString() to call to GetU
            User.Identity.GetUserId<int>().ToStrin
}
```

Add Call to ToString() to LinkLoginCallback():

```
public async Task<ActionResult> LinkLoginCallback(
{
    var loginInfo = await AuthenticationManager
        .GetExternalLoginInfoAsync(XsrfKey, User.I
    if (loginInfo == null)
    {
        return RedirectToAction("ManageLogins", ne
    }
    var result = await UserManager
        // *** Add .ToString() to call to GetUserI
        .AddLoginAsync(User.Identity.GetUserId<int

    return result.Succeeded ? RedirectToAction("Ma
```

```
        : RedirectToAction("ManageLogins", new { M
}
```

With that, we have addressed most of the egregious issues, and we basically taken a project built against a model set using all string keys and converted it to using integers. The integer types will be propagated as auto-incrementing integer primary keys in the database backend as well.

But there are still a few things to clean up.

Fix Null Checks Against Integer Types

Scattered throughout the primary identity controllers are a bunch of null checks against the Id values received as arguments in the method calls. If you rebuild the project, the error list window in Visual Studio should now contain a bunch of the yellow "warning" items about this very thing.

You can handle this in your preferred manner, but for me, I prefer to check for a positive integer value. We'll look at the `Details()` method from the `UserAdminController` as an example, and you can take it from there.

The existing code in the `Details()` method looks like this:

Existing Details() Method from UserAdminController:

```
public async Task<ActionResult> Details(int id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatus
    }
    var user = await UserManager.FindByIdAsync(id)
    ViewBag.RoleNames = await UserManager.GetRoles
    return View(user);
}
```

In the above, we can see that previously, the code checked for a null value for the (formerly) string-typed Id argument. Now that we are receiving an `int`, the check for null is meaningless. Instead, we want to check for a positive integer value. If the check is true, then we want to process accordingly. Otherwise, we want to return the `BadRequest` result.

In other words, we need to invert the method logic. Previously, if the conditional evaluated to true, we wanted to return the error code. Now, is the result is true, we want to proceed, and only return the error result if the conditional is false. So we're going to swap our logic around.

Replace the code with the following:

Modified Details() Method with Inverted Conditional Logic:

```
public async Task<ActionResult> Details(int id)
{
    if (id > 0)
    {
        // Process normally:
        var user = await UserManager.FindByIdAsync
        ViewBag.RoleNames = await UserManager.GetR
        return View(user);
```

```
        }
        // Return Error:
        return new HttpStatusCodeResult(HttpStatusCode
    }
```

We can do something similar for the other cases in `UserAdminController`, `RolesAdminController`, and `AccountController`. Think through the logic carefully, and all should be well.

Update Roles Admin Views

Several of the View Templates currently use the default IdentityRole model instead of our new, custom `ApplicationRole`. We need to update the Views in *Views => RolesAdmin* to reflect our new custom model.

The *Create.cshtml* and *Edit.cshtml* Views both depend upon the `RoleViewModel`, which is fine. However, the *Index.cshtml*, *Details.cshtml*, and *Delete.cshtml* Views all currently refer to `IdentityRole`. Update all three as follows

The *Index.cshtml* View currently expects an `IEnumerable<IdentityRole>`. We need to change this to expect an `IEnumerable<ApplicationRole>`. Note that we need to include the project Models namespace as well:

Update the RolesAdmin Index.cshtml View:

```
@model IEnumerable<IdentitySample.Models.Applicati
// ... All the view code ...
```

All we need to change here is the first line, so I omitted the rest of the View code.

Similarly, we need to update the *Details.cshtml* and *Delete.cshtml* Views to expect `ApplicationRole` instead of `IdentityRole`. Change the first line in each to match the following:

Update the Details.cshtml and Delete.cshtml Views:

```
@model IdentitySample.Models.ApplicationRole
// ... All the view code ...
```

Obviously, if your default project namespace is something other than `IdentitySamples`, change the above to suit.

Additional Extensions are Easy Now

Now that we have essentially re-implemented most of the Identity object models with our own derived types, it is easy to add custom properties to the ApplicationUser and/.or ApplicationRole models. All of our custom types already depend upon each other in terms of the interrelated generic type arguments, so we are free to simply add what properties we wish to add, and then update our Controllers, ViewModels, and Views accordingly.

To do so, review the previous post on extending Users and Roles, but realize all of the type structure stuff is already done. Review that post just to see what goes on with updating the Controllers, Views, and

ViewModels.

A Note on Security

The basic Identity Samples application is a great starting point for building out your own Identity 2.0 application. However, realize that, as a demo, there are some things built in that should not be present in production code. For example, the database initialization currently includes hard-coded admin user credentials.

Also, the Email confirmation and two-factor authentication functionality currently circumvents the actual confirmation and two-factor process, by including links on each respective page which short-circuit the process.

The above items should be addressed before deploying an actual application based upon the Identity Samples project.

Wrapping Up

We've taken a rather exhaustive look at how to modify the Identity Samples application to use integer keys instead of strings. Along the way, we (hopefully) gained a deeper understanding of the underlying structure in an Identity 2.0 based application. There's a lot more there to learn, but this is a good start.

Additional Resources and Items of Interest

- Source Code on Github
- .ASP.NET Identity Recommended Resources by Rick Anderson
- ASP.NET Identity 2.0: Setting Up Account Validation and Two-Factor Authorization
- ASP.NET MVC and Identity 2.0: Understanding the Basics
- Routing Basics in ASP.NET MVC
- Customizing Routes in ASP.NET MVC

ASP.NET ASP.NET MVC C#

RELATED ARTICLES

C#

C# SMTP Configuration for Outlook.Com SMTP Host

ASP.NET

C# – Generate and Deliver PDF Files On-Demand from a Template Using iTextSharp

C#

Use Cross-Platform/OSS ExcelDataReader to Read Excel Files with No Dependencies on Office or ACE

VIEW COMMENTS

PREVIOUS POST
Creating A Basic Make File for Compiling C Code

NEXT POST
ASP.NET Identity 2.0: Extensible Template Projects