May 26, 2015

Customizing the ASP.NET Identity Data Model with the Entity Framework Fluent API – Part 1

Written By:

JONATHAN BROWN

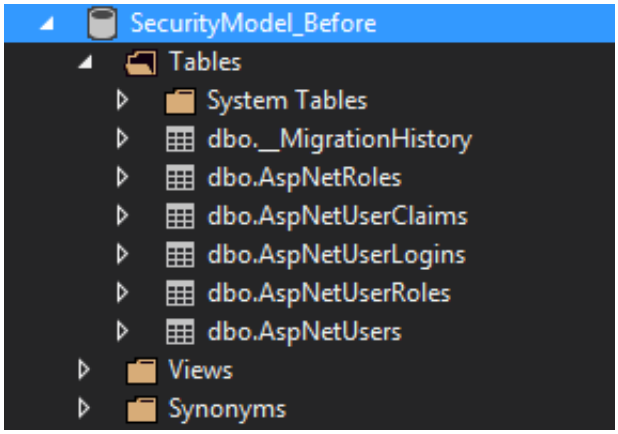
One of the first issues you will likely encounter when getting started with ASP.NET Identity centers on customizing the underlying data model. The Entity Framework provider uses Code-First to generate the data model and, initially, it may seem as if it is imposing its model upon your application. Fortunately, since the implementation of the provider uses Code-First, we can make significant customizations to the model and still take advantage of the features that ASP.NET Identity and EF provide.

In part one of this series, we will customize the ASP.NET Identity data model by simply changing the default schema and renaming the tables. In part two of this series, we will add audit fields to some of the tables and change the primary key data types from GUIDs to integers.

To get started, let's generate the default data model to see what we are working with:

1. Start by creating a new ASP.NET MVC and/or Web API project. Be sure the Authentication Mode is set to "Individual User Accounts" so the project template pulls in the required references, as well as the scaffolding for the default security model.
2. Update the default connection string ("DefaultConnection") in the web.config to point to your SQL Server database.
3. Build and run the application.

Next, navigate to the login page and attempt to sign in with any credentials. Your login attempt will fail because no accounts are registered, but the Entity Framework should have generated the default data model for users, roles, and claims. If you check the database, you will find something similar to the following:



That is all well and good and if you have worked with the Membership Provider for .NET, you should be reasonably comfortable with what you see. However, we are interested in customizing the model; so let's get started by renaming the tables and moving them into our application schema.

Step 1: Create the object model

To get started, add the following classes to your project. These classes form the object model that will be mapped to the data model. If you are following along in the attached sample project, you will find these classes under the *NAM_Sample_Pt1.Models* namespace.

ApplicationUserRole.cs

```
public class ApplicationUserRole : IdentityUserRole { }
```

ApplicationRole.cs

```
public class ApplicationRole : IdentityRole { }
```

ApplicationUserClaim.cs

```
public class ApplicationUserClaim : IdentityUserClaim { }
```

ApplicationUserLogin.cs

```
public class ApplicationUserLogin : IdentityUserLogin { }
```

IdentityModels.cs

Update the *ApplicationUser* class with the following:

```
public class ApplicationUser : IdentityUser
{
    public async Task GenerateUserIdentityAsync(ApplicationUserManager manager)
    {
        // Note the authenticationType must match the one defined in CookieAuthent
        var userIdentity = await manager.CreateIdentityAsync(this, DefaultAuthenti
        // Add custom user claims here
        return userIdentity;
     }
}
```

Step 2: Create the EF data context

Create a new security data context in IdentityModels.cs according to the following definition:

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

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

Note that the data context inherits from *IdentityDbContext*, which is the generic base data context that is included in the EF Provider for Identity. *IdentityDbContext* includes several generic type parameters, which should be set to the various types defined in the object model we created in the previous step.

We will revisit the data context once we have finished configuring the objects required to customize the model; however, be aware that this is where we will implement the fluent mapping.

Step 3: Create a custom user store

In ASP.NET Identity 2.0 user stores are the repositories for user data. The Entity Framework implementation of the user store requires a data context. Here is the implementation of our custom user store:

```
public class ApplicationUserStore :
    UserStore,
    IUserStore,
    IDisposable
{
    public ApplicationUserStore(ApplicationDbContext context) : base(context) { }
}
```

Step 4: Modify ApplicationUserManager to use the new object model

There are several lines in the *ApplicationUserManager* (included in the default project template) that must be modified. First, in the static *Create*() method, modify the creation of the *ApplicationUserManager* so that it takes an *ApplicationUserStore* and *ApplicationDbContext* as arguments in its constructor, as such:

```
        var manager = new ApplicationUserManager(new ApplicationUserStore(context.
```

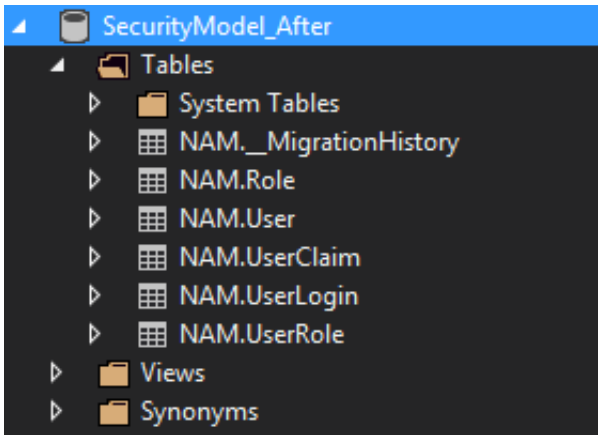
Step 5: Create the fluent mapping

We are finally ready to map our objects to our new data model. Begin by overriding *OnModelCreating*() in *ApplicationDbContext* . We will use EF Fluent API to map each of the five objects in our security object model to new tables in a new schema. The full fluent API mapping is included below:

```
        protected override void OnModelCreating(System.Data.Entity.DbModelBuilder
        {
            modelBuilder.HasDefaultSchema("NAM");

            modelBuilder.Entity().Map(c =>
            {
                c.ToTable("UserLogin");
                c.Properties(p => new
                {
                    p.UserId,
                    p.LoginProvider,
                    p.ProviderKey
                });
            }).HasKey(p => new { p.LoginProvider, p.ProviderKey, p.UserId });

            // Mapping for ApiRole
            modelBuilder.Entity().Map(c =>
            {
                c.ToTable("Role");
                c.Property(p => p.Id).HasColumnName("RoleId");
                c.Properties(p => new
                {
                    p.Name
                });
            }).HasKey(p => p.Id);
            modelBuilder.Entity().HasMany(c => c.Users).WithRequired().HasForeignK

            modelBuilder.Entity().Map(c =>
            {
                c.ToTable("User");
                c.Property(p => p.Id).HasColumnName("UserId");
                c.Properties(p => new
                {
                    p.AccessFailedCount,
                    p.Email,
                    p.EmailConfirmed,
                    p.PasswordHash,
                    p.PhoneNumber,
                    p.PhoneNumberConfirmed,
                    p.TwoFactorEnabled,
                    p.SecurityStamp,
                    p.LockoutEnabled,
                    p.LockoutEndDateUtc,
                    p.UserName
                });
            }).HasKey(c => c.Id);
            modelBuilder.Entity().HasMany(c => c.Logins).WithOptional().HasForeign
            modelBuilder.Entity().HasMany(c => c.Claims).WithOptional().HasForeign
            modelBuilder.Entity().HasMany(c => c.Roles).WithRequired().HasForeignK
```

```
            modelBuilder.Entity().Map(c =>
            {
                c.ToTable("UserRole");
                c.Properties(p => new
                {
                    p.UserId,
                    p.RoleId
                });
            })
            .HasKey(c => new { c.UserId, c.RoleId });

            modelBuilder.Entity().Map(c =>
            {
                c.ToTable("UserClaim");
                c.Property(p => p.Id).HasColumnName("UserClaimId");
                c.Properties(p => new
                {
                    p.UserId,
                    p.ClaimValue,
                    p.ClaimType
                });
            }).HasKey(c => c.Id);
        }
```

You are now ready to build and run the project. As before, navigate to the login page and attempt to sign in, which will force the creation of the new data model. You should now see the model in the custom schema with the table names we declared in the fluent mapping.



In part two of this series we will add audit fields to some of the tables and change the primary key data types from GUIDs to integers.

Attachments

NAM_Sample_Pt1 (/library/captech/blogs/customizing-the-aspnet-identity-data-model-with-the-entity-framework-fluent-api--part-1/nam_sample_pt1.zip)

Locations

Charlotte, NC (/contact/charlotte-nc)

Washington, DC Metro (/contact/washington-dc-metro)

Philadelphia, PA (/contact/philadelphia-pa)

Richmond, VA (/contact/headquarters)

Follow Us

Facebook (https://www.facebook.com/CapTechCareers?_rdr)

Instagram (https://instagram.com/lifeatcaptech)

LinkedIn (https://www.linkedin.com/company/captech-ventures)

Twitter (https://twitter.com/CapTechListens)

Youtube (https://www.youtube.com/user/CapTechConsulting)