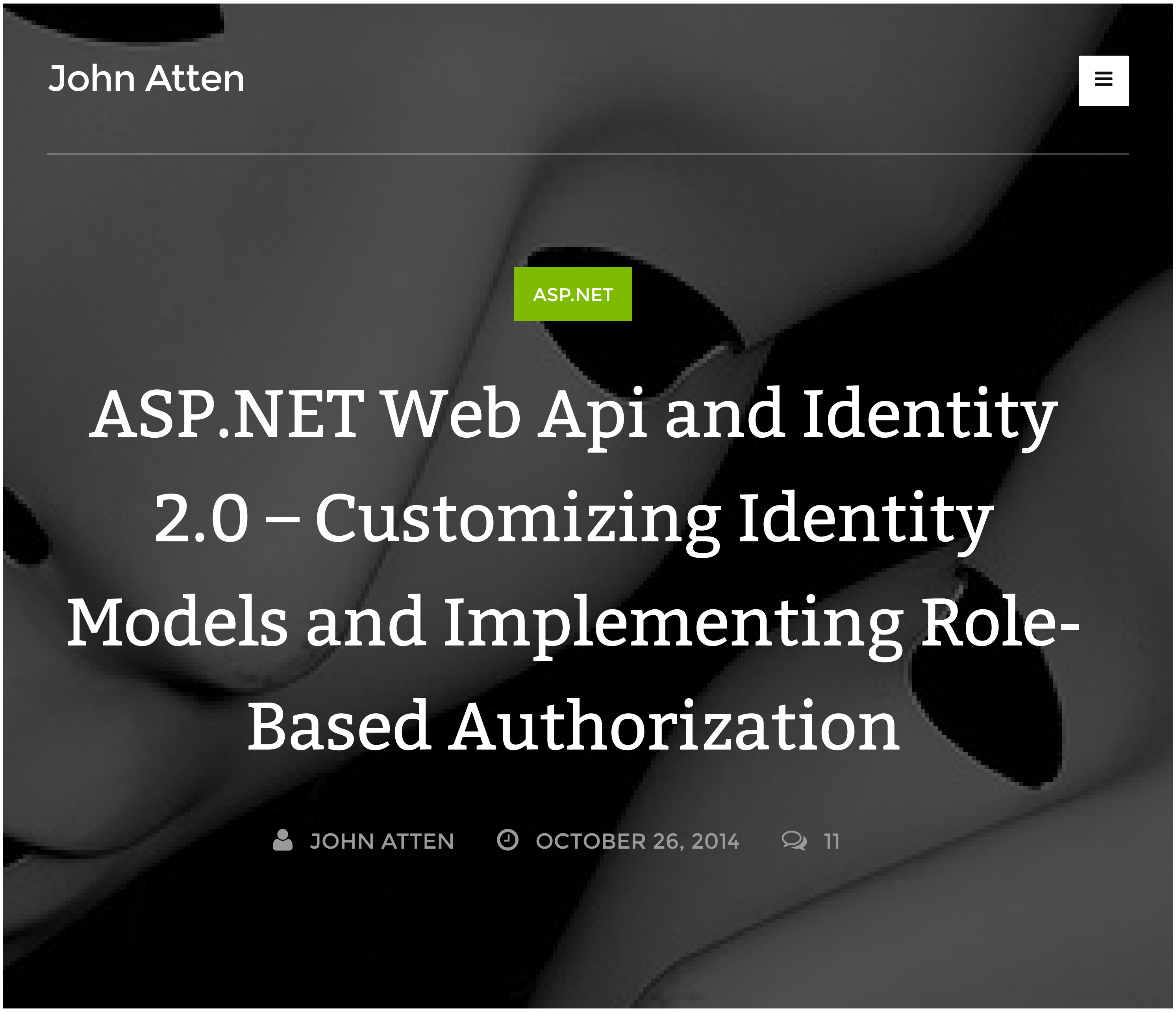


John Atten

ASP.NET

ASP.NET Web Api and Identity 2.0 – Customizing Identity Models and Implementing Role-Based Authorization

JOHN ATTEN OCTOBER 26, 2014 11

Image by madamepsychosis | Some Rights Reserved

In a previous post, we took a high-level look at using Identity 2.0 in the context of a Web Api application. We essentially poked and prodded the default Visual Studio Web Api project template, learned where things live, and got a basic sense for how it all is supposed to work.

However, the VS project template is very basic, using

Bearer Tokens as the primary authentication mechanism, and does not offer any out-of-the-box support for advanced authorization scenarios. In a nutshell, the VS Project template affords us basic token-based authentication, and that's about it.

Additionally, the User model is simplistic, and there are no Role models defined. This may be intentional, since in some cases you may be using a Web Api project as a simple authentication service. In other scenarios, though, you may want to customize the User model and/or add Role-Based Authentication to the mix.

We have previously looked at customizing User and Role models in the context of an ASP.NET MVC application, and how we need to modify the stock MVC project to accommodate these customizations. In this post, we will do the same for a Web Api project.

- Adding a Role Model, and Customizing ApplicationUser
- Adding a DBInitializer and Other Identity Config Items
- Modify AccountController to Use ApplicationUser
- Add Initialization for ApplicationRoleManager in Startup.Auth
- About the ApplicationDbInitializer
- Adding Custom Properties to ApplicationUser and ApplicationRole

- Create a Simple Web Api Client Application
- A Little More On the Nature of Bearer Tokens (but only a little)
- A Note About Bearer Tokens and Security
- Adding Role-Based Authorization to the Web Api Application
- Add a Vanilla Users Role as a Default in ApplicationDbInitializer
- Modify Client Code to Attempt Controller Access
- Accessing Custom User Properties
- Role-Based Authorization Versus Claims-Based Authorization

You can find the source code for the example Web Api project on Github:

- Source Code on Github

Consider Your Use Case Before Deciding on Your Auth Strategy

There are a number of options for Authentication and Authorization strategies in a Web Api project. Use of Bearer tokens and Role-Based Authentication is relatively simple to implement, but is not the most advanced architecture for an authorization solution. Before deciding upon traditional Role-Based

Authorization, you may want to examine the scope of your authentication and authorization needs, and determine if something simpler, or something more advanced, may be warranted.

ASP.NET Web Api can take full advantage of Claims-Based Authorization, which, for more complex systems, may be a better choice. Similarly, as mentioned previously, if the primary purpose of your Web Api is to act as an Authentication Service, you may want to go with a more robust token system (for example, shared private keys as opposed to the bearer tokens used by default), and do away with authorization at this level.

Role-Based Authorization is a good fit in a project where there exists a modest need for different levels of authorization/access, and possibly the Web Api is a part of, or associated with, a larger MVC or other ASP.NET site where Roles are used to govern authorization. Consider a standard MVC project, in which a few roles are sufficient to manage authorization, and which serves web pages as well as offers API access.

Applying What We've Learned Previously

Fortunately, much of what we are about to do, we have seen previously, and we can even borrow bits and pieces of code we've already written. If you are just getting familiar with Identity 2.0, feel free to review previous posts in which we performed some similar customization of Users and Roles in the context of an ASP.NET MVC project:

- ASP.NET MVC and Identity 2.0: Understanding the Basics
- ASP.NET Identity 2.0: Customizing Users and Roles
- ASP.NET Identity 2.0: Introduction to Working with Identity 2.0 and Web API 2.2

Now that we have some idea what we are dealing with, let's see how we can apply it in the Web Api context.

Getting Started – Create a New ASP.NET Web Api Project

First, in Visual Studio, create a new ASP.NET Web Api project. Once the project is created, update the Nuget packages in the solution, either using Manage Packages for Solution in the context menu for Solution Explorer, or by using `Update-Package` in the Package Manager Console.

This will update all the nuget packages, and specifically update Web Api to version 2.2.

Adding a Role Model, and Customizing ApplicationUser

To get started, let's take another look at the *Models => IdentityModes.cs* file. Currently, there is not a lot there:

The Default IdentityModels.cs File in Web Api:

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;

namespace AspNetIdentity2WebApiCustomize.Models
{
    public class ApplicationUser : IdentityUser
    {
        public async Task<ClaimsIdentity> GenerateUs
                UserManager<ApplicationUser> manager
        {
            // Note the authenticationType must matc
            // CookieAuthenticationOptions.Authentic
            var userIdentity = await manager.CreateI
            // Add custom user claims here
            return userIdentity;
        }
    }

    public class ApplicationDbContext : IdentityDbCo
    {
        public ApplicationDbContext()
            : base("DefaultConnection", throwIfV1Sch
        {
        }

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

As we did in our ASP.NET MVC examples, we begin by modifying and adding to the existing models defined in *Models => IdentityModels.cs.* In fact, since we did a lot of the work previously, we will start by stealing the *IdentityModels.cs* code from the ASP.NET Extensible Template Project. Careful here. We can save ourselves some pain by pasting the classes into the existing namespace defined in the current code file, and leaving the using statements as they are for the moment:

Updated IdentityModels.cs Code:

```
// You will not likely need to customize there, but
// project-specific implementations, so here they ar
public class ApplicationUserLogin : IdentityUserLogi
public class ApplicationUserClaim : IdentityUserClai
public class ApplicationUserRole : IdentityUserRole<

// Must be expressed in terms of our custom Role anc
public class ApplicationUser
    : IdentityUser<string, ApplicationUserLogin,
    ApplicationUserRole, ApplicationUserClaim>
{
    public ApplicationUser()
    {
        this.Id = Guid.NewGuid().ToString();

        // Add any custom User properties/code here
    }


    public async Task<ClaimsIdentity>
        GenerateUserIdentityAsync(ApplicationUserMar
    {
        var userIdentity = await manager
            .CreateIdentityAsync(this, DefaultAuther
        return userIdentity;
    }
}


// Must be expressed in terms of our custom UserRole
public class ApplicationRole : IdentityRole<string,
{
    public ApplicationRole()
    {
        this.Id = Guid.NewGuid().ToString();
    }

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

    // Add any custom Role properties/code here
}


// Must be expressed in terms of our custom types:
public class ApplicationDbContext
    : IdentityDbContext<ApplicationUser, Applicatior
    string, ApplicationUserLogin, ApplicationUserRol
{
    public ApplicationDbContext()
        : base("DefaultConnection")
    {
    }

    static ApplicationDbContext()
    {
        Database.SetInitializer<ApplicationDbContext
    }

    public static ApplicationDbContext Create()
    {
        return new ApplicationDbContext();
    }
    // Add additional items here as needed
}

// Most likely won't need to customize these either,
// custom versions of all the other types:
public class ApplicationUserStore
    :UserStore<ApplicationUser, ApplicationRole, str
        ApplicationUserLogin, ApplicationUserRole,
        ApplicationUserClaim>, IUserStore<Applicatic
    IDisposable
{
    public ApplicationUserStore()
        : this(new IdentityDbContext())
    {
        base.DisposeContext = true;
    }
    public ApplicationUserStore(DbContext context)
        : base(context)
    {
    }
}


public class ApplicationRoleStore
: RoleStore<ApplicationRole, string, ApplicationUser
IQueryableRoleStore<ApplicationRole, string>,
IRoleStore<ApplicationRole, string>, IDisposable
{
    public ApplicationRoleStore()
        : base(new IdentityDbContext())
    {
        base.DisposeContext = true;
    }

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

Now, we need to add a few additional using statements at the top of the code file to bring in some references we need with the new code. Add the following to the using statements at the top of the file:

Additional Using Statements Added to IdentityModels.cs:

```
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;
using System;
```

There are two immediate problems with the code we just pasted in there. The first is probably obvious, because the VS compiler is probably telling you that there is no `DBInitializer` class defined. Also, if you Build the project, a few other problems will surface in the VS error list. We'll take care of that in a moment.

The other is not so obvious. The code we pasted in here is from an MVC project. For the most part this is fine. However, our `ApplicationUser` class defines a method `GenerateUserIdentityAsync`. The code we stole from our MVC project has this method, but defines it in terms of a single parameter of type `ApplicationUserManager`. Recall the code we pasted over, which defined `ApplicationUser` with two constructor parameters. The missing parameter in our newly copied method is of type `string`, and represents the `authenticationType`.

This is important, because `GenerateUserIdentityAsync` is called when we need to retrieve a user's `ClaimsIdentity`, which represents the various claims the specific user has within our system.

Confused yet? We don't need to worry about the details of `ClaimsIdentity` just yet. What we DO need to do is update the `GenerateUserIdentityAsync` method defined on `ApplicationUser` to accept a `string` parameter representing the `authenticationType`.

Update ApplicationUser for Web Api

To make our `ApplicationUser` class ready for use in a Web Api context, we can replace the code for the

`GenerateUserIdentityAsync` method with the following:

Update GenerateUserIdentityAsync with Authentication Type Parameter:

```
// ** Add authenticationtype as method parameter:
public async Task<ClaimsIdentity>
    GenerateUserIdentityAsync(ApplicationUserManager
{
    // Note the authenticationType must match the or
    // in CookieAuthenticationOptions.Authenticatio
    var userIdentity =
        await manager.CreateIdentityAsync(this, auth
    // Add custom user claims here
    return userIdentity;
}
```

Adding a DBInitializer and Other Identity Config Items

We mentioned earlier, and the compiler is helpfully pointing out to you, that the code we stole from the Identity Extensible Template project is attempting to use a `DBInitializer` class that doesn't exist (yet) in our Web Api project. Also, you probably notice (if you have built the project since adding the additional Identity Models), that there appear to be some problems with our new `ApplicationUser` class.

We will resolve most of these issues by once again stealing select bits of code from the Identity Extensible Template project.

If we look at the *App_Start => IdentityConfig.cs* file in our

Web Api project, we see that, as with the original *IdentityModels.cs* file, there is not much there:

The Default Identity.config File from a Web Api Project:

```
using System.Threading.Tasks;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using AspNetIdentity2WebApiCustomize.Models;

namespace AspNetIdentity2WebApiCustomize
{
    // Configure the application user manager used i
    // is defined in ASP.NET Identity and is used by
    public class ApplicationUserManager : UserManage
    {
        public ApplicationUserManager(IUserStore<App
            : base(store)
        {
        }

        public static ApplicationUserManager Create(
                IdentityFactoryOptions<ApplicationUs
                IOwinContext context)
        {
            var manager =
                    new ApplicationUserManager(
                            new UserStore<Applicatio
                                    context.Get<Appl

            // Configure validation logic for userna
            manager.UserValidator = new UserValidato
            {
                AllowOnlyAlphanumericUserNames = fal
                RequireUniqueEmail = true
            };
            // Configure validation logic for passwo
            manager.PasswordValidator = new Password
            {
                RequiredLength = 6,
                RequireNonLetterOrDigit = true,
                RequireDigit = true,
                RequireLowercase = true,
                RequireUppercase = true,
            };
            var dataProtectionProvider = options.Dat
            if (dataProtectionProvider != null)
            {
                manager.UserTokenProvider =
                        new DataProtectorTokenProvic
                                dataProtectionProvic
            }
            return manager;
        }
    }
}
```

In order to work with Roles in our Web Api project, we will need an `ApplicationRoleManager`, and as mentioned previously, we will be adding the `ApplicationDbInitializer` from the Extensible Template project.

First, we need the following using statements at the top of the *IdentityConfig.cs* file:

Using Statements for the IdentityConfig.cs File:

```
using AspNetIdentity2WebApiCustomize.Models;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using System.Data.Entity;
using System.Web;
```

Now, add the `ApplicationRoleManager` and `ApplicationDbInitializer` classes from the Extensible Template project to our *IdentityConfig.cs* file:

Add ApplicationRoleManager and

ApplicationDbInitializer to IdentityConfig.cs:

```
using AspNetIdentity2WebApiCustomize.Models;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using System.Data.Entity;
using System.Web;


namespace AspNetIdentity2WebApiCustomize
{
    public class ApplicationUserManager : UserManage
    {
        public ApplicationUserManager(IUserStore<App
            : base(store)
        {
        }

        public static ApplicationUserManager Create(
            IdentityFactoryOptions<ApplicationUserMa
            IOwinContext context)
        {
            var manager = new ApplicationUserManager
                new UserStore<ApplicationUser>(
                    context.Get<ApplicationDbContext

            // Configure validation logic for userna
            manager.UserValidator = new UserValidato
            {
                AllowOnlyAlphanumericUserNames = fal
                RequireUniqueEmail = true
            };
            // Configure validation logic for passwo
            manager.PasswordValidator = new Passworc
            {
                RequiredLength = 6,
                RequireNonLetterOrDigit = true,
                RequireDigit = true,
                RequireLowercase = true,
                RequireUppercase = true,
            };
            var dataProtectionProvider = options.Dat
            if (dataProtectionProvider != null)
            {
                manager.UserTokenProvider =
                    new DataProtectorTokenProvider<A
                    dataProtectionProvider.Creat
            }
            return manager;
        }
    }


    public class ApplicationRoleManager : RoleManage
    {
        public ApplicationRoleManager(IRoleStore<App
            : base(roleStore)
        {
        }

        public static ApplicationRoleManager Create(
            IdentityFactoryOptions<ApplicationRoleMa
            IOwinContext context)
        {
            return new ApplicationRoleManager(
                new ApplicationRoleStore(context.Get
        }
    }


    public class ApplicationDbInitializer
        : DropCreateDatabaseAlways<ApplicationDbCont
    {
        protected override void Seed(ApplicationDbCo
        {
            InitializeIdentityForEF(context);
            base.Seed(context);
        }

        //Create User=Admin@Admin.com with password=
        public static void InitializeIdentityForEF(A
        {
            var userManager = HttpContext.Current
                .GetOwinContext().GetUserManager<App

            var roleManager = HttpContext.Current
                .GetOwinContext().Get<ApplicationRol

            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(roleNa
            if (role == null)
            {
                role = new ApplicationRole(roleName)
                var roleresult = roleManager.Create(
            }

            var user = userManager.FindByName(name);
            if (user == null)
            {
                user = new ApplicationUser { UserNam
                var result = userManager.Create(user
                result = userManager.SetLockoutEnabl
            }

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

Now, we need to make a few changes to our `ApplicationUserManager`. Since we have added our customizable models, including modified versions of `UserStore` and `RoleStore`, we need to adapt the `ApplicationUserManager` to play nice. We have expressed our models with different type arguments than the default implementation expected by the Web Api project. Specifically, we have employed a customized implementation of `IUserStore`. Rather than the concrete `UserStore` defined in `Microsoft.AspNet.Identity.EntityFramework`, we have implemented our own `ApplicationUserStore`, which is expressed in terms of specific type arguments.

We now need to tune up our `ApplicationUserManager` to work with our `ApplicationUserStore`.

Change the code for `ApplicationUserManager` in *IdentityConfig.cs* to the following:

Modified ApplicationUserManager:

```
public class ApplicationUserManager
    : UserManager<ApplicationUser, string>
{
    public ApplicationUserManager(IUserStore<Applica
        : base(store)
    {
    }

    public static ApplicationUserManager Create(
        IdentityFactoryOptions<ApplicationUserManage
        IOwinContext context)
    {
        var manager = new ApplicationUserManager(
            new UserStore<ApplicationUser, Applicati
                ApplicationUserLogin, ApplicationUse
                ApplicationUserClaim>(context.Get<Ap

        // Configure validation logic for usernames
        manager.UserValidator = new UserValidator<Ap
        {
            AllowOnlyAlphanumericUserNames = false,
            RequireUniqueEmail = true
        };

        // Configure validation logic for passwords
        manager.PasswordValidator = new PasswordVali
        {
            RequiredLength = 6,
            RequireNonLetterOrDigit = true,
            RequireDigit = true,
            RequireLowercase = true,
            RequireUppercase = true,
        };
        var dataProtectionProvider = options.DataPro
        if (dataProtectionProvider != null)
        {
            manager.UserTokenProvider =
                new DataProtectorTokenProvider<Appli
                    dataProtectionProvider.Create("A
        }
        return manager;
    }
}
```

With that, the very minimal basics are in place for us to use our new, extensible model classes (including Roles, which were not directly available to us in the default Web Api implementation) in our Web Api project.

We do need to clean up one more issue though. The `AccountController` still appears to rely on `Microsoft.AspNet.Identity.EntityFramework.IdentityUser`, and we need it to use our new implementation `ApplicationUser`.

Modify AccountController to Use ApplicationUser

We can easily correct this last remaining issue. Open the `AccountController` class, and locate the `GetManageInfo()` method. We can see where a local variable `user` is declared, explicitly types as `IdentityUser`.

Below that, we can see in the `foreach()` loop, we explicitly declare an iterator variable `linkedAccount` as type `IdentityUserLogin`.

Existing Code in the Web Api AccountController

GetManageInfo() Method:

```
[Route("ManageInfo")]
public async Task<ManageInfoViewModel> GetManageInfo
    string returnUrl, bool generateState = false)
{
    IdentityUser user =
        await UserManager.FindByIdAsync(User.Identit
    if (user == null)
    {
        return null;
    }

    List<UserLoginInfoViewModel> logins = new List<U
    foreach (IdentityUserLogin linkedAccount in user
    {
        logins.Add(new UserLoginInfoViewModel
        {
            LoginProvider = linkedAccount.LoginProvi
            ProviderKey = linkedAccount.ProviderKey
        });
    }

    if (user.PasswordHash != null)
    {
        logins.Add(new UserLoginInfoViewModel
        {
            LoginProvider = LocalLoginProvider,
            ProviderKey = user.UserName,
        });
    }

    return new ManageInfoViewModel
    {
        LocalLoginProvider = LocalLoginProvider,
        Email = user.UserName,
        Logins = logins,
        ExternalLoginProviders = GetExternalLogins(r
    };
}
```

In both cases we have implemented our own versions of these types. Here, we can either change the declaration in each case to use the `var` keyword, which relieves us of the type constraint on the variable (but, some would argue, makes our code a bit ambiguous), or we can change the explicit type declaration in each case to use our own implementation.

For now, let's change the explicit type declaration to use our own implementations:

Modified Code for GetManageInfo() Method:

```
[Route("ManageInfo")]
public async Task<ManageInfoViewModel> GetManageInfo
    string returnUrl, bool generateState = false)
{
    ApplicationUser user =
        await UserManager.FindByIdAsync(User.Identit
    if (user == null)
    {
        return null;
    }

    List<UserLoginInfoViewModel> logins = new List<U
    foreach (ApplicationUserLogin linkedAccount in u
    {
        logins.Add(new UserLoginInfoViewModel
        {
            LoginProvider = linkedAccount.LoginProvi
            ProviderKey = linkedAccount.ProviderKey
        });
    }

    if (user.PasswordHash != null)
    {
        logins.Add(new UserLoginInfoViewModel
        {
            LoginProvider = LocalLoginProvider,
            ProviderKey = user.UserName,
        });
    }

    return new ManageInfoViewModel
    {
        LocalLoginProvider = LocalLoginProvider,
        Email = user.UserName,
        Logins = logins,
        ExternalLoginProviders = GetExternalLogins(r
    };
}
```

Above, we have simply changed the declared type for the local user variable from `IdentityUser` to `ApplicationUser`, and the iterator variable `linkedAccount` from I`dentityUserLogin` to `ApplicationUserLogin`.

Add Initialization for ApplicationRoleManager in Startup.Auth

Recall from our high-level exploration of ASP.NET Web Api and Identity that initialization and configuration of Identity 2.0 occurs in the `Startup` class defined in *App_Start => Startup.Auth*.

As we have seen, the original VS Web Api template did not really provide for Role-Based anything, and consequently, provides no configuration or initialization for our recently added ApplicationRoleManager at startup. We need to add a line of initialization code to our Startup.Auth file:

Add Initialization for ApplicationRoleManager in Statup.Auth:

```
public void ConfigureAuth(IAppBuilder app)
{
    app.CreatePerOwinContext(ApplicationDbContext.Cr
    app.CreatePerOwinContext<ApplicationUserManager>
    app.CreatePerOwinContext<ApplicationRoleManager>

    app.UseCookieAuthentication(new CookieAuthentica
    app.UseExternalSignInCookie(DefaultAuthenticatio

    // Configure the application for OAuth based flo
    PublicClientId = "self";
    OAuthOptions = new OAuthAuthorizationServerOptio
    {
        TokenEndpointPath = new PathString("/Token")
        Provider = new ApplicationOAuthProvider(Publ
        AuthorizeEndpointPath = new PathString("/api
        AccessTokenExpireTimeSpan = TimeSpan.FromDay
        AllowInsecureHttp = true
    };

    // Enable the application to use bearer tokens t
    app.UseOAuthBearerTokens(OAuthOptions);

    // ... Code for third-part logins omitted for br
}
```

We’ve added a single line, which initializes an instance of ApplicationRoleManager for each incoming request.

About the ApplicationDbInitializer

With the changes we’ve introduced so far, we should be able to take our new and improved Web Api project for a test spin to see if the most basic functionality works.

Before we do, though, we need to recognize that we have fundamentally changed how the EF/Code-First database generation has been changed with the introduction of our custom `ApplicationDbInitializer`.

Recall from our explorations of customizing an MVC project with extensible models, the `ApplicationDbInitializer` allows us to specify some options for how and when the database behind our application is generated, and to provide some initial data to work with.

As we move towards Role-Based Authorization and a more restrictive security model for our Api, this becomes important.

The way we currently have `ApplicationDbInitializer` configured, it derives from `DbDropCreateDatabaseAlways`, which means every time we run our application, the backing store will be destroyed, and re-created from scratch. We also have it set up to create a default User, and we assign that user to the Admin role. In this manner, we start our application with a user with Admin-level access permissions.

The default VS Web Api project doesn't take advantage of this out of the box. If we look at the class declaration for `AccountController` we see that the class itself is decorated with a simple `[Authorize]` attribute. What this essentially does is restrict access to all of the Action methods on the class to authorized users (except those methods specifically decorated with an `[AllowAnonymous]` attribute).

In other words, for now, any user who is registered, and who successfully signs in and presents a valid Bearer Token can access any of the Action methods on `AccountController`.

We'll take a closer look at implementing Role-Based Authentication momentarily, First, we will extend our ApplicationUser and ApplicationRole classes with some custom properties.

Adding Custom Properties to ApplicationUser and ApplicationRole

As we saw when we examine customizing Users and Roles within an MVC project, we will add a few simple properties to our ApplicationUser and ApplicationRole models. Modify the code for each as follows:

Add Custom Properties to ApplicationUser and ApplicationRole:

```
// Must be expressed in terms of our custom Role and
public class ApplicationUser
    : IdentityUser<string, ApplicationUserLogin,
    ApplicationUserRole, ApplicationUserClaim>
{
    public ApplicationUser()
    {
        this.Id = Guid.NewGuid().ToString();
    }


    public async Task<ClaimsIdentity>GenerateUserIde
        ApplicationUserManager manager, string authe
    {
        // Note the authenticationType must match th
        // defined in CookieAuthenticationOptions.Au
        var userIdentity = await manager.CreateIdent

        // Add custom user claims here
        return userIdentity;
    }

    // Add Custom Properties:
    public string Address { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
}


// Must be expressed in terms of our custom UserRole
public class ApplicationRole : IdentityRole<string,
{
    public ApplicationRole()
    {
      this.Id = Guid.NewGuid().ToString();
    }

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

    // Add Custom Property:
    public string Description { get; set; }
}
```

Here, we have added an `Address` and related properties to `ApplicationUser`, and a simple `Description` property to `ApplicationRole`.

Now, let's update our `ApplicationDbInitializer` to set some sample values for these new properties. Update the code for the `InitializeIdentityForEF()` method as follows:

Set Initial Values for Custom Properties in

ApplicationDbInitializer:

```
public static void InitializeIdentityForEF(Applicati
{
    var userManager = HttpContext.Current
        .GetOwinContext().GetUserManager<Application

    var roleManager = HttpContext.Current
        .GetOwinContext().Get<ApplicationRoleManager

    const string name = "admin@example.com";
    const string password = "Admin@123456";

    // Some initial values for custom properties:
    const string address = "1234 Sesame Street";
    const string city = "Portland";
    const string state = "OR";
    const string postalCode = "97209";

    const string roleName = "Admin";
    const string roleDescription = "All access pass'

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

        // Set the new custom property:
        role.Description = roleDescription;
        var roleresult = roleManager.Create(role);
    }

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

        // Set the new custom properties:
        user.Address = address;
        user.City = city;
        user.State = state;
        user.PostalCode = postalCode;

        var result = userManager.Create(user, passwo
        result = userManager.SetLockoutEnabled(user.
    }
    // Add user admin to Role Admin if not already a
    var rolesForUser = userManager.GetRoles(user.Id)
    if (!rolesForUser.Contains(role.Name))
    {
        var result = userManager.AddToRole(user.Id,
    }
}
```

With that, we should be ready to see if everything at least works correctly . . .

Create a Simple Web Api Client Application

To see if everything is working properly to this point, we will create a simple console application as an Api client.

In Visual Studio, create a new Console Application, and the use the Manage Nuget Packages for Solutions to add the Microsoft Asp.NET Web Api 2.2 Client Libraries, or use the Package Manager Console and do:

Add Web Api 2.2 via the Nuget Package Manager Console:

```
PM> Install-Package Microsoft.AspNet.WebApi.Client
```

Now that we have the required Web Api Client Libraries in our project, open the *Program.cs* file.

Make sure the following using statements are present at the top of the file. Note we have added references to `System.Net.Http` and `Newtonsoft.Json`, ad well as

`System.Threading`:

Required Using Statements for Console Api Client Application:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Net.Http;
using Newtonsoft.Json;
```

Next, let's add some very basic client code used to retrieve a token from the token endpoint of our Web Api. Add the following within the Program class:

Add Client Code to Retreive Response from Web Api Token Endpoint:

```
// You will need to substitute your own host Url her
static string host = "http://localhost:63074/";

static void Main(string[] args)
{
    Console.WriteLine("Attempting to Log in with def

    // Get hold of a Dictionary representing the JSO
    var responseDictionary =
        GetResponseAsDictionary("admin@example.com",
    foreach(var kvp in responseDictionary)
    {
        Console.WriteLine("{0}: {1}", kvp.Key, kvp.V
    }
    Console.Read();
}


static Dictionary<string, string> GetResponseAsDicti
    string userName, string password)
{
    HttpClient client = new HttpClient();
    var pairs = new List<KeyValuePair<string, string
                {
                    new KeyValuePair<string, string>
                    new KeyValuePair<string, string>
                    new KeyValuePair<string, string>
                };
    var content = new FormUrlEncodedContent(pairs);

    // Attempt to get a token from the token endpoin
    HttpResponseMessage response =
        client.PostAsync(host + "Token", content).Re
    var result = response.Content.ReadAsStringAsync(
    // De-Serialize into a dictionary and return:
    Dictionary<string, string> tokenDictionary =
        JsonConvert.DeserializeObject<Dictionary<str
    return tokenDictionary;
}
```

Note, at this point, we have added code the the Main() method, which calls out to a single, rather contrived method `GetResponseAsDictionary()` . All we are basically doing here is submitting an HTTP POST to the Token endpoint of our Web Api, and then de-serializing the JSON response body into a `Dictionary<string, string>` .

Once we have the Dictionary, we are iterating over each Key/Value pair, and writing the contents to the Console.

If everything has worked the way we expect, our Console output should be something along the lines of the following:

Console Output from Token Endpoint Response:

```
Attempting to Log in with default admin user
access_token: AuzOQkgG3BYubP1rlljcPhAzW7R7gA4Vew8dHy
uf5rflB7PCfamlcT_-KJ4q3lfx7kiFNpSF9SdMLwKP_mCSOXGbrx
Bvp_RqtXgkSmgpcNWitCU8RBz7aOaHr8r-FCklg4wUkLNE26qlR6
```

```
W4YehAzrqFAuTX3peMJBQB8K8_XxaTkRnEhSEMz9DnUnqzQjjVr5
7au787CMn7EGiDO9KRcGHAsGHOJqb8P8Z7A-ssV7tfEqJayrNH-F
Fp4_xOavE6wkYcHTfXWZJWFEMokE4NB9mtAl3lReYSZQyzKkcHWF
IG2OmhyR1wZNRCHY_6NwEMOIHGLpA_L-kFFAJPgwQWi-WljeV-X2
token_type: bearer
expires_in: 1209599
userName: admin@example.com
.issued: Sun, 26 Oct 2014 13:21:03 GMT
.expires: Sun, 09 Nov 2014 13:21:03 GMT
```

We've seen this before, in our overview article. The de-serialized JSON above represents the content of the response to our POST to the Token endpoint of our Web Api. The important part of the response is and `access_token` itself.

This Doesn't Look Any Different Than Before . . .

At this point, that de-serialized JSON response doesn't look any different than it did in our previous post, before we added all our fancy new Roles and custom properties. Shouldn't it have some new information in it now? Roles, and Addresses and stuff?

No.

A Little More On the Nature of Bearer Tokens (but only a little)

We had a really, really brief look Bearer Tokens in our Introduction to Identity in Web Api. As mentioned there, we will undertake a more in-depth exploration of Tokens in another post.

For our purposes here today, we are simply going to expand a little on what we learned previously, sufficient to understand how the `access_token` we retrieved as part of our JSON response above fits into the scheme of our newly modified Web Api project.

Bearer Tokens are, by design, “opaque” to the client. In other words, they are encoded (and sometimes encrypted) on the server, and can be decoded (and potentially decrypted) by the server. They are not designed to be decoded/decrypted by the client.

Recall from exploring the basic structure of an ASP.NET Web Api project, the ApplicationOauthProvider class, defined in the *Providers => ApplicationOauthProvider.cs* file.

When you POST a request to the Token endpoint of the ASP.NET Web Api application (at least, in the way we have it configured here), the server validates the credentials you present (in this case, user name + Password) by calling the GrantResourceOwnersCredentials() method defined on the ApplicationOauthProvider class:

The GrantResourceOwnersCredentials() Method from ApplicationOAuthProvider:

```
public override async Task GrantResourceOwnerCredent
```

```
    OAuthGrantResourceOwnerCredentialsContext contex
{
    var userManager = context.OwinContext.GetUserMan
    ApplicationUser user = await userManager.FindAsy

    if (user == null)
    {
        context.SetError("invalid_grant", "The user 
        return;
    }

    ClaimsIdentity oAuthIdentity = await user.Genera
       OAuthDefaults.AuthenticationType);
    ClaimsIdentity cookiesIdentity = await user.Gene
        CookieAuthenticationDefaults.AuthenticationT

    AuthenticationProperties properties = CreateProp
    AuthenticationTicket ticket = new Authenticatior
    context.Validated(ticket);
    context.Request.Context.Authentication.SignIn(co
}
```

We can see that this code attempts to find a user with credentials matching those presented. If a valid user is found, the method calls the `GenerateUserIdentityAsync()` method defined on the `ApplicationUser` class to obtain an instance of `ClaimsIdentity` representing the user, and any claims the user may make within our system.

The `ClaimsIdentity` is then used to create an `AuthenticationTicket`.

In the code for `GrantResourceOwnersCredentials(),` above, when `Validated()` is called on the `OAuthGrantResourceOwnerCredentialsContext`, the OWIN middleware serializes the `ClaimsIdentity` into an

All of the above is a long-winded way of saying, the important user information, including user roles, and anything else we decide needs to be a part of our token, is present in the token when it is received by the client.

The Client just can't get to it.

A Note About Bearer Tokens and Security

As mentioned in our previous posts, using bearer tokens, and submitting credentials to obtain the token from the token endpoint should only be done over SSL/STL in a production system. Bearer tokens are exactly what their name implies – anyone presenting a valid bearer token will be granted access to the system, with whatever privileges the actual "owner" of the token has.

OAuth Bearer tokens represent a fairly simple authentication/authorization scheme, In building out your Web Api, consider all of the alternatives, and make the best choice for your application.

As mentioned previously, we will take a more thorough look at Claims Identity, and token authentication/authorization in later posts.

Adding Role-Based Authorization to

the Web Api Application

Now that we understand how to authenticate ourselves using a bearer token, let's look at how we can use our new Role-Based Authorization capability within our application.

To start, let's look at the two primary controllers present in the Web Api application, `AccountController` and `ValuesController` (we are ignoring the `HomeController`, since it serves no purpose for our needs here).

We've already seen that `AccountController` is decorated with an `[Authorize]` attribute. This means that only authenticated users may access the action methods defined on this controller (unless, of course, the method itself is decorated with an `[AllowAnonymous]` attribute).

Let's look at the simplistic `ValuesController`. `ValuesController` is provided as a simple example of how one might add a basic CRUD-style functionality. `ValuesController` is similarly decorated with the `[Authorize]` attribute. Again, only authenticated users are able to access the Action methods defined on `ValuesController`.

As we saw in previous posts about implementing Role-Based Authorization in an MVC project, we can modify the access permissions for our Web Api, at either the Controller level, or the Action method level, by expanding on our use of `[Authorize]` .

Consider, we might want to restrict access to `AccountController` only to users who are in the Admin role, but allow access to `ValuesController`, and the functionality it provides, to any authenticated user.

In this case, we will want to make some modifications to our Web Api configuration.

Add a Vanilla Users Role as a Default in ApplicationDbInitializer

First, let's make sure we have two distinct Roles available in our application – the "Admin" role we already create as an initial value during configuration, and a new "Users" role. Update the `InitializeDatabaseForEF()` method as follows:

Add a Users Role and a Default User to InitializeDatabaseForEF() Method:

```
public static void InitializeIdentityForEF(Applicati
{
    var userManager = HttpContext.Current
        .GetOwinContext().GetUserManager<Application

    var roleManager = HttpContext.Current
        .GetOwinContext().Get<ApplicationRoleManager

    // Initial Admin user:
    const string name = "admin@example.com";
    const string password = "Admin@123456";

    // Some initial values for custom properties:
    const string address = "1234 Sesame Street";
    const string city = "Portland";
    const string state = "OR";
    const string postalCode = "97209";

    const string roleName = "Admin";
    const string roleDescription = "All access pass'

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

        // Set the new custom property:
        role.Description = roleDescription;
        var roleresult = roleManager.Create(role);
    }

    // Create Admin User:
    var user = userManager.FindByName(name);
    if (user == null)
    {
        user = new ApplicationUser { UserName = name

        // Set the new custom properties:
        user.Address = address;
        user.City = city;
        user.State = state;
        user.PostalCode = postalCode;

        var result = userManager.Create(user, passwo
        result = userManager.SetLockoutEnabled(user.
    }

    // Add user admin to Role Admin if not already a
    var rolesForUser = userManager.GetRoles(user.Id)
    if (!rolesForUser.Contains(role.Name))
    {
        userManager.AddToRole(user.Id, role.Name);
    }

    // Initial Vanilla User:
    const string vanillaUserName = "vanillaUser@exam
    const string vanillaUserPassword = "Vanilla@1234
    // Add a plain vannilla Users Role:
    const string usersRoleName = "Users";
    const string usersRoleDescription = "Plain vanil

    //Create Role Users if it does not exist
    var usersRole = roleManager.FindByName(usersRole
    if (usersRole == null)
    {
        usersRole = new ApplicationRole(usersRoleNam

        // Set the new custom property:
        usersRole.Description = usersRoleDescription
        var userRoleresult = roleManager.Create(user
    }

    // Create Vanilla User:
    var vanillaUser = userManager.FindByName(vanilla
    if (vanillaUser == null)
    {
        vanillaUser = new ApplicationUser
        {
            UserName = vanillaUserName,
            Email = vanillaUserName
        };

        // Set the new custom properties:
        vanillaUser.Address = address;
        vanillaUser.City = city;
        vanillaUser.State = state;
        vanillaUser.PostalCode = postalCode;

        var result = userManager.Create(vanillaUser,
        result = userManager.SetLockoutEnabled(vanil
    }

    // Add vanilla user to Role Users if not already
    var rolesForVanillaUser = userManager.GetRoles(v
    if (!rolesForVanillaUser.Contains(usersRole.Name
    {
        userManager.AddToRole(vanillaUser.Id, usersR
    }
}
```

Above, we have added a new role “Users” and another initial sample user.

Next, let's modify the `[Authorize]` attribute on our `AccountController` class, and add a Role argument:

Modified [Authorize] Attribute for AccountController:

```
[Authorize(Roles= "Admin")]
[RoutePrefix("api/Account")]
public class AccountController : ApiController
{
    // ... All the Code ...
}
```

Now, we just need to change up our client code to attempt to access some methods from each of the two controllers to see how our Role-Based Authorization is working for us.

Modify Client Code to Attempt Controller Access

Here, we will simply set up some client code to attempt to retreive some basic data from both `AccountController` and `ValuesController`. We will do this as a user in the Admin Role, and then also as a user in the Users Role.

Change the code in your Console application to match the following:

Modified Client Code to Access Both Controllers with Different Roles:

```
class Program
{
    // You will need to substitute your own host Url
    static string host = "http://localhost:63074/";

    static void Main(string[] args)
    {
        // Use the User Names/Emails and Passwords w
        string adminUserName = "admin@example.com";
        string adminUserPassword = "Admin@123456";

        string vanillaUserName = "vanillaUser@exampl
        string vanillaUserPassword = "Vanilla@123456

        // Use the new GetToken method to get a toke
        string adminUserToken = GetToken(adminUserNa
        string vaniallaUserToken = GetToken(vanillaU

        // Try to get some data as an Admin:
        Console.WriteLine("Attempting to get User in
        string adminUserInfoResult = GetUserInfo(adm
        Console.WriteLine("Admin User Info Result: {
        Console.WriteLine("");

        Console.WriteLine("Attempting to get Values 
        string adminValuesInfoResult = GetValues(adm
        Console.WriteLine("Admin Values Info Result:
        Console.WriteLine("");

        // Try to get some data as a plain old user:
        Console.WriteLine("Attempting to get User in
        string vanillaUserInfoResult = GetUserInfo(v
        Console.WriteLine("Vanilla User Info Result:
        Console.WriteLine("");

        Console.WriteLine("Attempting to get Values 
        string vanillaValuesInfoResult = GetValues(v
        Console.WriteLine("Vanilla Values Info Resul
        Console.WriteLine("");

        Console.Read();
    }


    static string GetToken(string userName, string p
    {
        HttpClient client = new HttpClient();
        var pairs = new List<KeyValuePair<string, st
                    {
                        new KeyValuePair<string, str
                        new KeyValuePair<string, str
                        new KeyValuePair<string, str
                    };
        var content = new FormUrlEncodedContent(pair

        // Attempt to get a token from the token enc
        HttpResponseMessage response =
            client.PostAsync(host + "Token", content
        var result = response.Content.ReadAsStringAs

        // De-Serialize into a dictionary and return
        Dictionary<string, string> tokenDictionary =
            JsonConvert.DeserializeObject<Dictionary
        return tokenDictionary["access_token"];
    }


    static string GetUserInfo(string token)
    {
        using (var client = new HttpClient())
        {
            client.DefaultRequestHeaders.Authorizati
            new System.Net.Http.Headers.Authenticati
            var response = client.GetAsync(host + "a
            return response.Content.ReadAsStringAsyn
        }
    }


    static string GetValues(string token)
    {
        using (var client = new HttpClient())
        {
            client.DefaultRequestHeaders.Authorizati
            new System.Net.Http.Headers.Authenticati

            var response = client.GetAsync(host + "a
            return response.Content.ReadAsStringAsyn
        }
    }
}
```

In the above, we changed our code up a bit. We now have a `GetToken()` method, which accepts a User Name/Password as arguments, and returns only the access_token string from the request to the Token endpoint of our Web Api.

Next, we added two different calls to our Web Api. One method calls the `GetUserInfo()` method on the `AccountController`, and the other calls the `Get()` method on our `ValuesController`.

If we spin up our Web Api, and, after it has spun up, run our client application, we should see the following output in our Console:

Console Output from Access Permissions Comparison:

```
Attempting to get User info as Admin User
Admin User Info Result: {"Email":"admin@example.com'
Provider":null}
Attempting to get Values info as Admin User
Admin Values Info Result: ["value1","value2"]
Attempting to get User info as Vanilla User
Vanilla User Info Result: {"Message":"Authorization
uest."}
Attempting to get Values info as Vanilla User
Vanilla Values Info Result: ["value1","value2"]
```

Note the output from that third attempt. We are trying to call into `GetUserInfo()` as a plain vanilla User, in the Users Role. Appropriately, our Web Api has returned an authorization error, since `Users` are not allowed access to the method by virtue of the `[Authorize(Roles="Admin")]` attribute on the class declaration for `AccountController`.

In contrast, both users are able to access the `Get()`

method on `ValuesController`, since this controller is decorated with a simple `[Authorize]` attribute, which requires only an authenticated user for access.

Accessing Custom User Properties

We have also added some custom properties to our `ApplicationUserModel`. Let's take a look and see if we can work with those in the context of our examples here.

If we look more closely at the `GetUserInfo()` method on `AccountController`, we find that the actual return type for this method is `UserInfoViewModel`, which is found in the *Models => AccountViewModels.cs* file. Now, in our crude, simple Console application we are not going to all the effort of de-serializing the JSON from our GET request into an object, but we COULD.

For our purposes, here, it will be sufficient to modify the `UserInfoViewModel` to reflect the additional properties we want to return, and then update the `GetUserInfo()` method to suit.

Add our custom User properties to the UserInfoViewModel class:

```
public class UserInfoViewModel
{
    public string Email { get; set; }
    public bool HasRegistered { get; set; }
    public string LoginProvider { get; set; }

    // Add our custom properties from ApplicationUse
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string PostalCode { get; set; }
}
```

Next, update the `GetUserInfo()` method to provide values for the additional properties we just added to `UserInfoViewModel`:

Update GetUserInfo() Method on AccountController with Custom Properties:

```
public UserInfoViewModel GetUserInfo()
{
    ExternalLoginData externalLogin
        = ExternalLoginData.FromIdentity(User.Identi

    // We wouldn't normally be likely to do this:
    var user = UserManager.FindByName(User.Identity.
    return new UserInfoViewModel
    {
        Email = User.Identity.GetUserName(),
        HasRegistered = externalLogin == null,
        LoginProvider = externalLogin != null ? exte

        // Pass the custom properties too:
        Address = user.Address,
        City = user.City,
        State = user.State,
        PostalCode = user.PostalCode
    };
}
```

Above, we have called out to `UserManager` to retreive an instance of our user, so we can get at the new properties we have added. We then set the corresponding values on the `UserInfoViewModel` before returning.

This example is a little contrived, and we most likely would NOT do this in a production application this way. But for now, it will serve to demonstrate that our new and improved `ApplicationUser` indeed has the custom properties we added to the model, and that we are retrieving them from the database as expected.

If we run our Console Client application one last time, we should see the following output:

Output from the Console Application with Custom User Properties:

```
Attempting to get User info as Admin User
Admin User Info Result: {"Email":"admin@example.com'
Provider":null,"Address":"1234 Sesame Street","City'
"PostalCode":"97209"}
Attempting to get Values info as Admin User
Admin Values Info Result: ["value1","value2"]
Attempting to get User info as Vanilla User
Vanilla User Info Result: {"Message":"Authorization
request."}
Attempting to get Values info as Vanilla User
Vanilla Values Info Result: ["value1","value2"]
```

And we see, our custom user properties are returned with the JSON response.

Role-Based Authorization Versus Claims-Based Authorization

In this post, we have looked briefly at implementing Role-Based Authorization in the context of an ASP.NET Web Api project, and we had the briefest look at how Bearer Tokens work. It is important to note though, that for any but the simplest of authorization/access control schemes, Role-Based Authorization ("RBA") is rapidly being overshadowed by Claims-Based Authorization.

Claims-Based Identity offers greater flexibility, and more effectively separates the Authentication and Authorization mechanism from your code. Note, in this example project, we need to specify, as part of the `[Authorize]` attribute, precisely which Roles are allowed access to which controllers and/or methods.

Claims-Based Auth is more complex than RBA, but is generally going to be a more natural fit for a Web Api scenario, unless your needs are fairly simple.

We will examine Tokens, and Claims-Based Identity, in an upcoming post.

Additional Resources and Items of Interest

- ASP.NET Identity 2.0: Introduction to Working with Identity 2.0 and Web API 2.2
- ASP.NET: Understanding OWIN, Katana, and the Middleware Pipeline
- ASP.NET Web Api 2.2: Create a Self-Hosted OWIN-Based Web Api from Scratch
- ASP.NET Identity 2.0: Customizing Users and Roles

- ASP.NET Web Api: Unwrapping HTTP Error Results and Model State Dictionaries Client-Side
- More on OAth Tokens (interesting discussion from a webcast by Apigee on August 2nd 2012)
- ASP.NET 4.5.1 WebAPI, "general" integration with OAuth2 and OAuth Authentication Servers

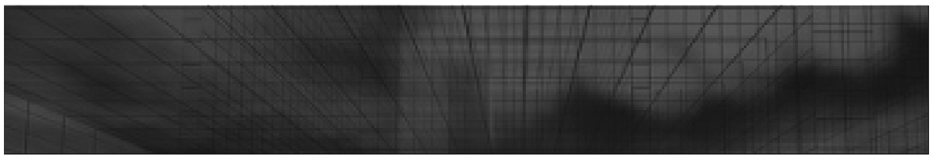
ASP.NET ASP.NET MVC C# WEB API

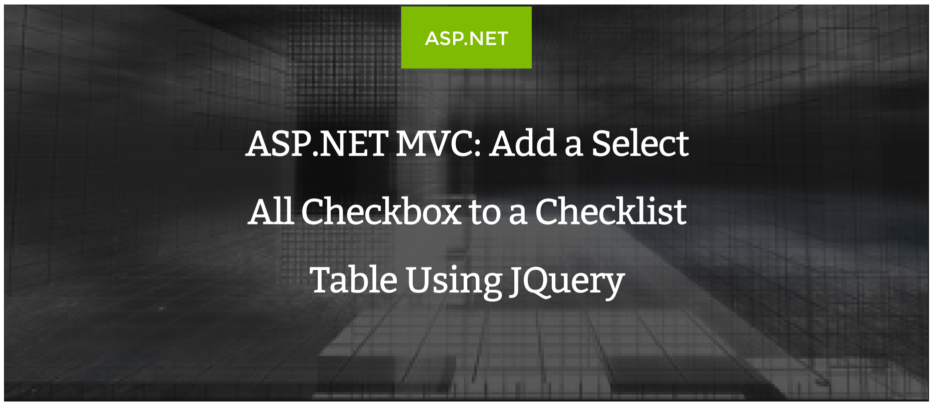
SHARE:

RELATED ARTICLES

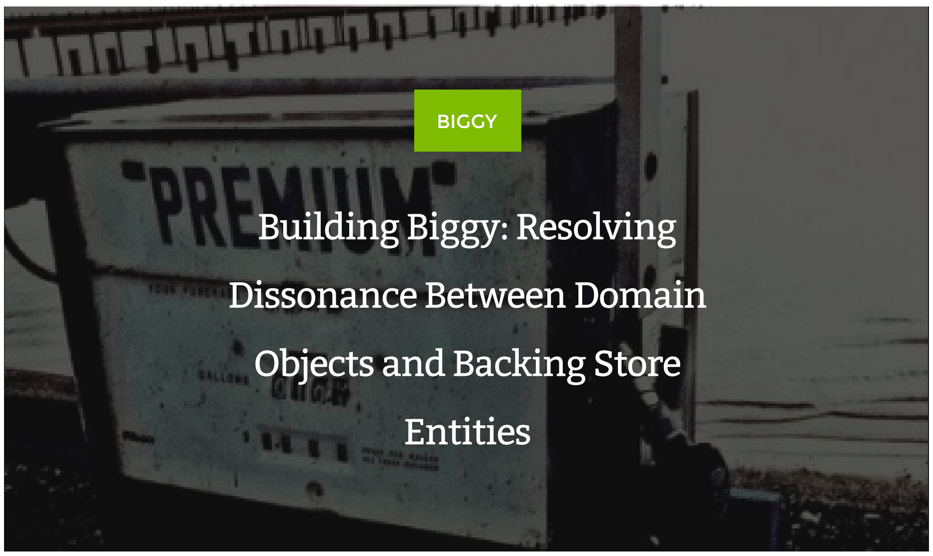
CODEPROJECT

Webmatrix 3: Integrated Git and Deployment to Azure




ASP.NET

ASP.NET MVC: Add a Select All Checkbox to a Checklist Table Using JQuery


BIGGY

Building Biggy: Resolving Dissonance Between Domain Objects and Backing Store Entities

COMMENTS

Josmonver

NOVEMBER 25, 2014

REPLY

Thanks for these great posts. I've learned ASP .Net Identity reading this blog. I'm looking forward to your new post about Tokens and Claims-Based Identity!

As said in another comment, "keep up the good work".

John Atten

NOVEMBER 25, 2014

REPLY

@Momoski – Great! Glad you found the answer, and thanks for linkiing to the SO question + solution.

Thanks for taking the time to comment as well! Cheers 🙂

Momoski

NOVEMBER 20, 2014

REPLY

I have answered my own question. I have created a customer UserStore with overrides for FindById to include the related record. See the SO link in the question to see the answer.

Momoski

NOVEMBER 20, 2014

REPLY

Sorry for post here but in your article "ASP.NET Identity 2.0 customizing Users and Roles" the comments were disabled and I have a question. However the question would also be applicable to this blog post.

How to include underlying/ related records when using FindByIdAsync?

I have asked the question on SO with more details. If you have time to answer I would really appreciate it. The question can be found here: http://stackoverflow.com/questions/27039407/include-property-on-asp-net-identity-2-0-usermanager-users-tolistasync-and-userm

---

Ben

NOVEMBER 1, 2014

REPLY

Quick question, when using Claim-based will the claims be in somehow stored in cookies and how can we trust these cookies?

---

jatten

OCTOBER 31, 2014

REPLY

Rich –

I'll look into it as soon as I can – I am travelling right now.

---

jatten

OCTOBER 31, 2014

REPLY

@Robert Gallardo –

You would do that the same way you did for the ApplicationRole or ApplicationUser. Add a property to the ApplicationUserRole class (ApplicationUserRole is also defined in the modified IdentityModels.cs file in the example project).

Rich OCTOBER 30, 2014

REPLY

First, thanks for this series of posts on Identity 2, very, very helpful. Very well written I look forward to each new installment to learn something new.

To date I have setup basic authentication as per the post on Understanding the basics and set up two factor authentication using email.

This all works as expected, however I cannot see where to customise the content of the emails that are sent. Is this possible?

I realise this is slightly off topic for this post, but comments are closed on the earlier posts.

Thanks again

Robert Gallardo OCTOBER 30, 2014

REPLY

Sorry for post here but in your article

[quote]"ASP.NET Identity 2.0 customizing Users and Roles"[/quote] the comments were disabled and I have a question: In the article you explain how to modify Users and Roles and its a very good tutorial but its not clear how to modify the user Role Identity. Say that we want to add a property to specified if a user in a role is active or the date when the user was added to the role. In other words how to add more fields to the AspNetUserRoles table in the DB. It will be great if you make an example. Thanks your work is great.

John Atten OCTOBER 26, 2014

REPLY

Thanks, and thanks for taking the time to comment! 🙂

SomeRandomGuy OCTOBER 26, 2014

REPLY

Just wanted to say thanks, I was reading and studying your previous post on Web API and then I refreshed page and saw this, thank you very much! Please continue writing on Web API and Identity, very very useful stuff. 🙂

Also, I've followed your posts on MVC and Identity, thanks for that as well!

Keep up the good work.

Best regards,

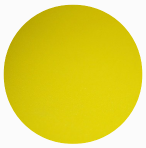
Name (Requir

Email (Require

Website URL

Add your comment here...

POST COMMENT



PREVIOUS POST

Setting Up for Mono Development in Linux Mint/Ubuntu

NEXT POST

Adding and Editing PATH Environment Variables in Windows