عنوان: **ASP.NET MVC و Identity 2.0 : مفاهیم پایه**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۳/۰۲/۰۸ ۱:۱۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, MVC, Security, ASP.NET Identity

در تاریخ 20 مارچ 2014 تیم ASP.NET نسخه نهایی Identity 2.0 را منتشر کردند . نسخه جدید برخی از ویژگی‌های درخواست شده پیشین را عرضه می‌کند و در کل قابلیت‌های احراز هویت و تعیین سطح دسترسی ارزنده ای را پشتیبانی می‌کند. این فریم ورک در تمام اپلیکیشن‌های ASP.NET می‌تواند بکار گرفته شود.

فریم ورک Identity در سال 2013 معرفی شد، که دنباله سیستم ASP.NET Membership بود. سیستم قبلی گرچه طی سالیان استفاده می‌شد اما مشکلات زیادی هم بهمراه داشت. بعلاوه با توسعه دنیای وب و نرم افزار، قابلیت‌های مدرنی مورد نیاز بودند که باید پشتیبانی می‌شدند. فریم ورک Identity در ابتدا سیستم ساده و کارآمدی برای مدیریت کاربران بوجود آورد و مشکلات پیشین را تا حد زیادی برطرف نمود. بعنوان مثال فریم ورک جدید مبتنی بر Code-first EF است، که سفارشی کردن سیستم عضویت را بسیار آسان می‌کند و به شما کنترل کامل می‌دهد. یا مثلا احراز هویت مبتنی بر پروتوکل OAuth پشتیبانی می‌شود که به شما اجازه استفاده از فراهم کنندگان خارجی مانند گوگل، فیسبوک و غیره را می‌دهد.

نسخه جدید این فریم ورک ویژگی‌های زیر را معرفی می‌کند (بعلاوه مواردی دیگر):

مدل حساب‌های کاربری توسعه داده شده. مثلا آدرس ایمیل و اطلاعات تماس را هم در بر می‌گیرد
احراز هویت دو مرحله ای (Two-Factor Authentication) توسط اطلاع رسانی ایمیلی یا پیامکی. مشابه سیستمی که گوگل، مایکروسافت و دیگران استفاده می‌کنند
تایید حساب‌های کاربری توسط ایمیل (Account Confirmation)
مدیریت کاربران و نقش‌ها (Administration of Users & Roles)
قفل کردن حساب‌های کاربری در پاسخ به Invalid log-in attempts

تامین کننده شناسه امنیتی (Security Token Provider) برای بازتولید شناسه‌ها در پاسخ به تغییرات تنظیمات امنیتی (مثلا هنگام تغییر کلمه عبور)

بهبود پشتیبانی از Social log-ins
یکپارچه سازی ساده با Claims-based Authorization

Identity 2.0 تغییرات چشم گیری نسبت به نسخه قبلی به‌وجود آورده است. به نسبت ویژگی‌های جدید، پیچیدگی‌هایی نیز معرفی شده‌اند. اگر به تازگی (مانند خودم) با نسخه 1 این فریم ورک آشنا شده و کار کرده اید، آماده شوید! گرچه لازم نیست از صفر شروع کنید، اما چیزهای بسیاری برای آموختن وجود دارد.

در این مقاله نگاهی اجمالی به نسخه‌ی جدید این فریم ورک خواهیم داشت. کامپوننت‌های جدید و اصلی را خواهیم شناخت و خواهیم دید هر کدام چگونه در این فریم ورک کار می‌کنند. بررسی عمیق و جزئی این فریم ورک از حوصله این مقاله خارج است، بنابراین به این مقاله تنها بعنوان یک نقطه شروع برای آشنایی با این فریم ورک نگاه کنید.

اگر به دنبال اطلاعات بیشتر و بررسی‌های عمیق‌تر هستید، لینک هایی در انتهای این مقاله نگاشت شده اند. همچنین طی هفته‌های آینده چند مقاله تخصصی‌تر خواهم نوشت تا از دید پیاده سازی بیشتر با این فریم ورک آشنا شوید.

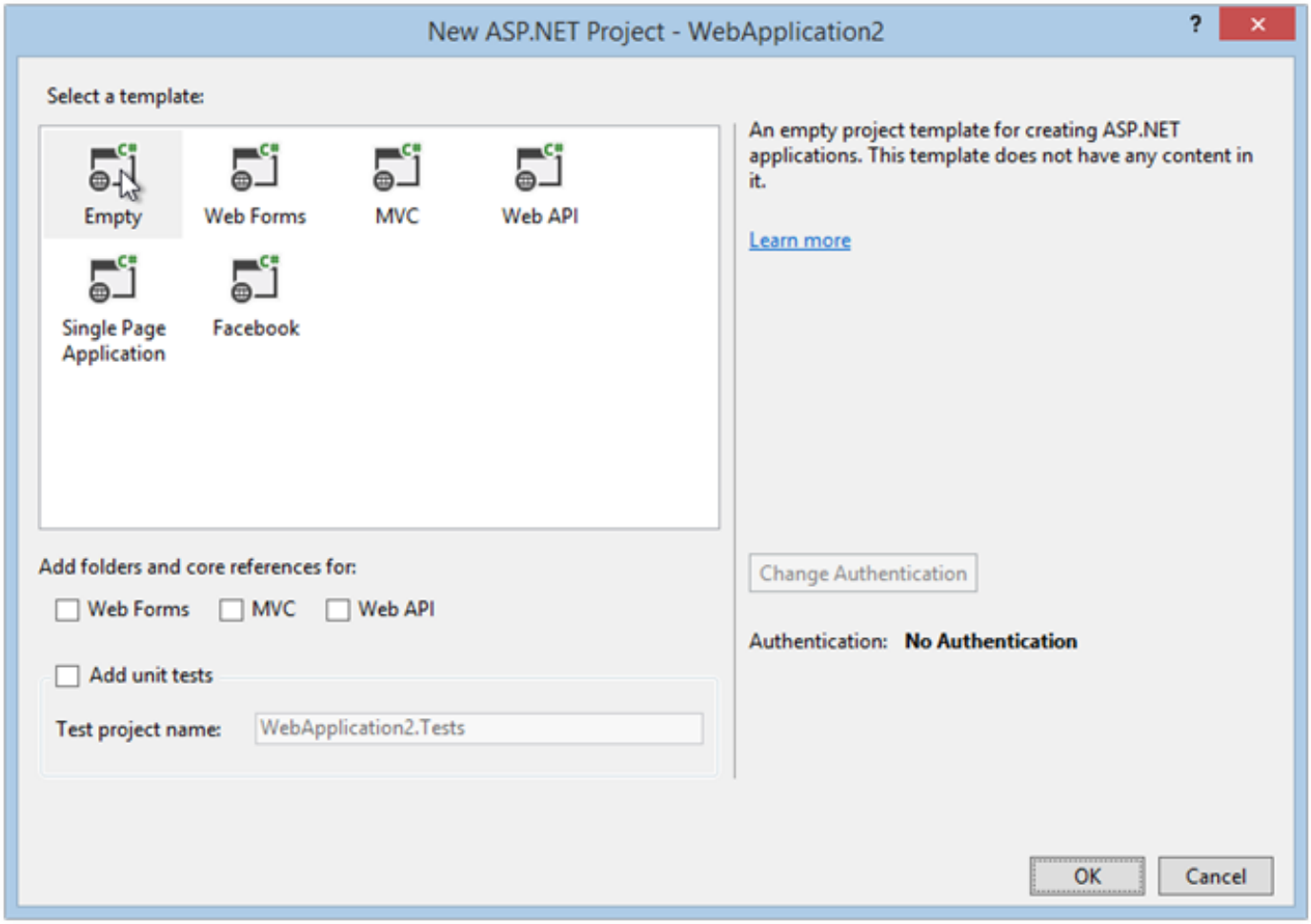
در این مقاله با مقدار قابل توجهی کد مواجه خواهید شد. لازم نیست تمام جزئیات آنها را بررسی کنید، تنها با ساختار کلی این فریم ورک آشنا شوید. کامپوننت‌ها را بشناسید و بدانید که هر کدام در کجا قرار گرفته اند، چطور کار می‌کنند و اجزای کلی سیستم چگونه پیکربندی می‌شوند. گرچه، اگر به برنامه نویسی دات نت (ASP.NET, C#) تسلط دارید و با نسخه قبلی Identity هم کار کرده اید، درک کدهای جدید کار ساده ای خواهد بود.

Identity 2.0 با نسخه قبلی سازگار نیست

اپلیکیشن هایی که با نسخه 1.0 این فریم ورک ساخته شده اند نمی‌توانند بسادگی به نسخه جدید مهاجرت کنند. قابلیت هایی جدیدی که پیاده سازی شده اند تغییرات چشمگیری در معماری این فریم ورک بوجود آورده اند، همچنین API مورد استفاده در اپلیکیشن‌ها نیز دستخوش تغییراتی شده است. مهاجرت از نسخه 1.0 به 2.0 نیاز به نوشتن کدهای جدید و اعمال تغییرات متعددی دارد که از حوصله این مقاله خارج است. فعلا همین قدر بدانید که این مهاجرت نمی‌تواند بسادگی در قالب Plug-in and play صورت پذیرد!

شروع به کار : پروژه مثال‌ها را از NuGet دریافت کنید

در حال حاظر (هنگام نوشتن این مقاله) قالب پروژه استانداردی برای اپلیکیشن‌های ASP.NET MVC که ا ز Identity 2.0 استفاده کنند وجود ندارد. برای اینکه بتوانید از نسخه جدید این فریم ورک استفاده کنید، باید پروژه مثال را توسط NuGet دریافت کنید. ابتدا پروژه جدیدی از نوع ASP.NET Web Application بسازید و قالب Empty را در دیالوگ تنظیمات انتخاب کنید.



کنسول Package Manager را باز کنید و با اجرای فرمان زیر پروژه مثال‌ها را دانلود کنید.

```
PM> Install-Package Microsoft.AspNet.Identity.Samples -Pre
```

پس از آنکه NuGet کار خود را به اتمام رساند، ساختار پروژه ای مشابه پروژه‌های استاندارد MVC مشاهده خواهید کرد. پروژه شما شامل قسمت‌های Models, Views, Controllers و کامپوننت‌های دیگری برای شروع به کار است. گرچه در نگاه اول ساختار پروژه بسیار شبیه به پروژه‌های استاندارد ASP.NET MVC به نظر می‌آید، اما با نگاهی دقیق‌تر خواهید دید که تغییرات جدیدی ایجاد شده‌اند و پیچیدگی هایی نیز معرفی شده اند.

پیکربندی Identity : دیگر به سادگی نسخه قبلی نیست

به نظر من یکی از مهم‌ترین نقاط قوت فریم ورک Identity یکی از مهم‌ترین نقاط ضعفش نیز بود. سادگی نسخه 1.0 این فریم ورک کار کردن با آن را بسیار آسان می‌کرد و به سادگی می‌توانستید ساختار کلی و روند کارکردن کامپوننت‌های آن را درک کنید. اما همین سادگی به معنای محدود بودن امکانات آن نیز بود. بعنوان مثال می‌توان به تایید حساب‌های کاربری یا پشتیبانی از احراز هویت‌های دو مرحله ای اشاره کرد.

برای شروع نگاهی اجمالی به پیکربندی این فریم ورک و اجرای اولیه اپلیکیشن خواهیم داشت. سپس تغییرات را با نسخه 1.0 مقایسه می‌کنیم.

در هر دو نسخه، فایلی بنام *Startup.cs* در مسیر ریشه پروژه خواهید یافت. در این فایل کلاس واحدی بنام Startup تعریف شده است که متد ()ConfigureAuth را فراخوانی می‌کند. چیزی که در این فایل مشاهده نمی‌کنیم، خود متد ConfigureAuth است. این بدین دلیل است که مابقی کد کلاس Startup در یک کلاس پاره ای (Partial) تعریف شده که در پوشه *App_Start* قرار دارد. نام فایل مورد نظر *Startup.Auth.cs* است که اگر آن را باز کنید تعاریف یک کلاس پاره ای بهمراه متد ()ConfigureAuth را خواهید یافت. در یک پروژه که از نسخه Identity 1.0 استفاده می‌کند، کد متد ()ConfigureAuth مطابق لیست زیر است.

```
public partial class Startup
{
    public void ConfigureAuth(IAppBuilder app)
    {
        // Enable the application to use a cookie to
        // store information for the signed in user
        app.UseCookieAuthentication(new CookieAuthenticationOptions
        {
            AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
            LoginPath = new PathString("/Account/Login")
        });

        // Use a cookie to temporarily store information about a
        // user logging in with a third party login provider
        app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);

        // Uncomment the following lines to enable logging
        // in with third party login providers
        //app.UseMicrosoftAccountAuthentication(
        //    clientId: "",
        //    clientSecret: "");

        //app.UseTwitterAuthentication(
        //   consumerKey: "",
        //   consumerSecret: "");

        //app.UseFacebookAuthentication(
        //   appId: "",
        //   appSecret: "");

        //app.UseGoogleAuthentication();
    }
}
```

در قطعه کد بالا پیکربندی لازم برای کوکی‌ها را مشاهده می‌کنید. همچنین کدهایی بصورت توضیحات وجود دارد که به منظور استفاده از تامین کنندگان خارجی مانند گوگل، فیسبوک، توییتر و غیره استفاده می‌شوند. حال اگر به کد این متد در نسخه Identity 2.0 دقت کنید خواهید دید که کد بیشتری نوشته شده است.

```
public partial class Startup {
    public void ConfigureAuth(IAppBuilder app) {
        // Configure the db context, user manager and role
        // manager to use a single instance per request
```

```
        app.CreatePerOwinContext(ApplicationDbContext.Create);
        app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);
        app.CreatePerOwinContext<ApplicationRoleManager>(ApplicationRoleManager.Create);

        // Enable the application to use a cookie to store information for the
        // signed in user and to use a cookie to temporarily store information
        // about a user logging in with a third party login provider
        // Configure the sign in cookie
        app.UseCookieAuthentication(new CookieAuthenticationOptions {
            AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
            LoginPath = new PathString("/Account/Login"),
            Provider = new CookieAuthenticationProvider {
                // Enables the application to validate the security stamp when the user
                // logs in. This is a security feature which is used when you
                // change a password or add an external login to your account.
                OnValidateIdentity = SecurityStampValidator
                    .OnValidateIdentity<ApplicationUserManager, ApplicationUser>(
                    validateInterval: TimeSpan.FromMinutes(30),
                    regenerateIdentity: (manager, user)
                    => user.GenerateUserIdentityAsync(manager))
            }
        });

        app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);

        // Enables the application to temporarily store user information when
        // they are verifying the second factor in the two-factor authentication process.
        app.UseTwoFactorSignInCookie(
            DefaultAuthenticationTypes.TwoFactorCookie,
            TimeSpan.FromMinutes(5));

        // Enables the application to remember the second login verification factor such
        // as phone or email. Once you check this option, your second step of
        // verification during the login process will be remembered on the device where
        // you logged in from. This is similar to the RememberMe option when you log in.
        app.UseTwoFactorRememberBrowserCookie(
            DefaultAuthenticationTypes.TwoFactorRememberBrowserCookie);

        // Uncomment the following lines to enable logging in
        // with third party login providers
        //app.UseMicrosoftAccountAuthentication(
        //    clientId: "",
        //    clientSecret: "");

        //app.UseTwitterAuthentication(
        //   consumerKey: "",
        //   consumerSecret: "");

        //app.UseFacebookAuthentication(
        //   appId: "",
        //   appSecret: "");

        //app.UseGoogleAuthentication();
    }
}
```

اول از همه به چند فراخوانی متد app.CreatePerOwinContext بر می‌خوریم. با این فراخوانی‌ها Callback هایی را رجیستر می‌کنیم که آبجکت‌های مورد نیاز را بر اساس نوع تعریف شده توسط type arguments وهله سازی می‌کنند. این وهله‌ها سپس توسط فراخوانی متد context.Get() قابل دسترسی خواهند بود. این به ما می‌گوید که حالا Owin بخشی از اپلیکیشن ما است و فریم ورک Identity 2.0 از آن برای ارائه قابلیت هایش استفاده می‌کند.

مورد بعدی ای که جلب توجه می‌کند فراخوانی‌های دیگری برای پیکربندی احراز هویت دو مرحله‌ای است. همچنین پیکربندی‌های جدیدی برای کوکی‌ها تعریف شده است که در نسخه قبلی وجود نداشتند.

تا اینجا پیکربندی‌های اساسی برای اپلیکیشن شما انجام شده است و می‌توانید از اپلیکیشن خود استفاده کنید. بکارگیری فراهم کنندگان خارجی در حال حاضر غیرفعال است و بررسی آنها نیز از حوصله این مقاله خارج است. این کلاس پیکربندی‌های اساسی Identity را انجام می‌دهد. کامپوننت‌های پیکربندی و کدهای کمکی دیگری نیز وجود دارند که در کلاس *IdentityConfig.cs* تعریف شده اند.

پیش از آنکه فایل *IdentityConfig.cs* را بررسی کنیم، بهتر است نگاهی به کلاس ApplicationUser بیاندازیم که در پوشه Models قرار گرفته است.

کلاس جدید ApplicationUser در Identity 2.0

اگر با نسخه 1.0 این فریم ورک اپلیکیشنی ساخته باشید، ممکن است متوجه شده باشید که کلاس پایه IdentityUser محدود و شاید ناکافی باشد. در نسخه قبلی، این فریم ورک پیاده سازی IdentityUser را تا حد امکان ساده نگاه داشته بود تا اطلاعات پروفایل کاربران را معرفی کند.

```
public class IdentityUser : IUser
{
    public IdentityUser();
    public IdentityUser(string userName);

    public virtual string Id { get; set; }
    public virtual string UserName { get; set; }

    public virtual ICollection<IdentityUserRole> Roles { get; }
    public virtual ICollection<IdentityUserClaim> Claims { get; }
    public virtual ICollection<IdentityUserLogin> Logins { get; }

    public virtual string PasswordHash { get; set; }
    public virtual string SecurityStamp { get; set; }
}
```

بین خواص تعریف شده در این کلاس، تنها Id, UserName و Roles برای ما حائز اهمیت هستند (از دید برنامه نویسی). مابقی خواص عمدتا توسط منطق امنیتی این فریم ورک استفاده می‌شوند و کمک شایانی در مدیریت اطلاعات کاربران به ما نمی‌کنند.

اگر از نسخه Identity 1.0 استفاده کرده باشید و مطالعاتی هم در این زمینه داشته باشید، می‌دانید که توسعه کلاس کاربران بسیار ساده است. مثلا برای افزودن فیلد آدرس ایمیل و اطلاعات دیگر کافی بود کلاس ApplicationUser را ویرایش کنیم و از آنجا که این فریم ورک مبتنی بر EF Code-first است بروز رسانی دیتابیس و مابقی اپلیکیشن کار چندان مشکلی نخواهد بود.

با ظهور نسخه Identity 2.0 نیاز به برخی از این سفارشی سازی‌ها از بین رفته است. گرچه هنوز هم می‌توانید بسادگی مانند گذشته کلاس ApplicationUser را توسعه و گسترش دهید، تیم ASP.NET تغییراتی بوجود آورده اند تا نیازهای رایج توسعه دهندگان را پاسخگو باشد.

اگر به کد کلاس‌های مربوطه دقت کنید خواهید دید که کلاس ApplicationUser همچنان از کلاس پایه IdentityUser ارث بری می‌کند، اما این کلاس پایه پیچیده‌تر شده است. کلاس ApplicationUser در پوشه *Models* و در فایلی بنام *IdentityModels.cs* تعریف شده است. همانطور که می‌بینید تعاریف خود این کلاس بسیار ساده است.

```
public class ApplicationUser : IdentityUser {
    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(
        UserManager<ApplicationUser> manager) {
        // Note the authenticationType must match the one
        // defined in CookieAuthenticationOptions.AuthenticationType
        var userIdentity =
            await manager.CreateIdentityAsync(this,
                DefaultAuthenticationTypes.ApplicationCookie);

        // Add custom user claims here
        return userIdentity;
    }
}
```

حال اگر تعاریف کلاس IdentityUser را بازیابی کنید (با استفاده از قابلیت Go To Definition در VS) خواهید دید که این کلاس خود از کلاس پایه دیگری ارث بری می‌کند. اگر به این پیاده سازی دقت کنید کاملا واضح است که ساختار این کلاس به کلی نسبت به نسخه قبلی تغییر کرده است.

```
public class IdentityUser<TKey, TLogin, TRole, TClaim> : IUser<TKey>
    where TLogin : Microsoft.AspNet.Identity.EntityFramework.IdentityUserLogin<TKey>
```

```
    where TRole : Microsoft.AspNet.Identity.EntityFramework.IdentityUserRole<TKey>
    where TClaim : Microsoft.AspNet.Identity.EntityFramework.IdentityUserClaim<TKey>
{
    public IdentityUser();

    // Used to record failures for the purposes of lockout
    public virtual int AccessFailedCount { get; set; }
    // Navigation property for user claims
    public virtual ICollection<TClaim> Claims { get; }
    // Email
    public virtual string Email { get; set; }
    // True if the email is confirmed, default is false
    public virtual bool EmailConfirmed { get; set; }
    // User ID (Primary Key)
    public virtual TKey Id { get; set; }
    // Is lockout enabled for this user
    public virtual bool LockoutEnabled { get; set; }
    // DateTime in UTC when lockout ends, any
    // time in the past is considered not locked out.
    public virtual DateTime? LockoutEndDateUtc { get; set; }

    // Navigation property for user logins
    public virtual ICollection<TLogin> Logins { get; }
    // The salted/hashed form of the user password
    public virtual string PasswordHash { get; set; }
    // PhoneNumber for the user
    public virtual string PhoneNumber { get; set; }
    // True if the phone number is confirmed, default is false
    public virtual bool PhoneNumberConfirmed { get; set; }
    // Navigation property for user roles
    public virtual ICollection<TRole> Roles { get; }

    // A random value that should change whenever a users
    // credentials have changed (password changed, login removed)
    public virtual string SecurityStamp { get; set; }
    // Is two factor enabled for the user
    public virtual bool TwoFactorEnabled { get; set; }
    // User name
    public virtual string UserName { get; set; }
}
```

اول از همه آنکه برخی از خواص تعریف شده هنوز توسط منطق امنیتی فریم ورک استفاده می‌شوند و از دید برنامه نویسی مربوط به مدیریت اطلاعات کاربران نیستند. اما به هر حال فیلدهای Email و PhoneNumber نیاز به ویرایش تعریف پیش فرض موجودیت کاربران را از بین می‌برد.

اما از همه چیز مهم‌تر امضا (Signature)ی خود کلاس است. این آرگومانهای جنریک چه هستند؟ به امضای این کلاس دقت کنید.

```
public class IdentityUser<TKey, TLogin, TRole, TClaim> : IUser<TKey>
    where TLogin : Microsoft.AspNet.Identity.EntityFramework.IdentityUserLogin<TKey>
    where TRole : Microsoft.AspNet.Identity.EntityFramework.IdentityUserRole<TKey>
    where TClaim : Microsoft.AspNet.Identity.EntityFramework.IdentityUserClaim<TKey>
```

نسخه جدید IdentityUser انواع آرگومانهای جنریک را پیاده سازی می‌کند که انعطاف پذیری بسیار بیشتری به ما می‌دهند. بعنوان مثال بیاد بیاورید که نوع داده فیلد Id در نسخه 1.0 رشته (string) بود. اما در نسخه جدید استفاده از آرگومانهای جنریک (در اینجا TKey) به ما اجازه می‌دهد که نوع این فیلد را مشخص کنیم. همانطور که مشاهده می‌کنید خاصیت Id در این کلاس نوع داده TKey را باز می‌گرداند.

```
public virtual TKey Id { get; set; }
```

یک مورد حائز اهمیت دیگر خاصیت Roles است که بصورت زیر تعریف شده.

```
public virtual ICollection<TRole> Roles { get; }
```

همانطور که می‌بینید نوع TRole بصورت جنریک تعریف شده و توسط تعاریف کلاس IdentityUser مشخص می‌شود. اگر به

محدودیت‌های پیاده سازی این خاصیت دقت کنید می‌بینید که نوع این فیلد به IdentityUserRole<TKey> محدود (constraint) شده است، که خیلی هم نسبت به نسخه 1.0 تغییری نکرده. چیزی که تفاوت چشمگیری کرده و باعث breaking changes می‌شود تعریف خود IdentityUserRole است.

در نسخه Identity 1.0 کلاس IdentityUserRole بصورت زیر تعریف شده بود.

```
public class IdentityUserRole
{
      public IdentityUserRole();
      public virtual IdentityRole Role { get; set; }
      public virtual string RoleId { get; set; }
      public virtual IdentityUser User { get; set; }
      public virtual string UserId { get; set; }
}
```

این کلاس را با پیاده سازی نسخه Identity 2.0 مقایسه کنید.

```
public class IdentityUserRole<TKey>
{
    public IdentityUserRole();
    public virtual TKey RoleId { get; set; }
    public virtual TKey UserId { get; set; }
}
```

پیاده سازی پیشین ارجاعاتی به آبجکت هایی از نوع IdentityRole و IdentityUser داشت. پیاده سازی نسخه 2.0 تنها شناسه‌ها را ذخیره می‌کند. اگر در اپلیکیشنی که از نسخه 1.0 استفاده می‌کند سفارشی سازی هایی انجام داده باشید (مثلا تعریف کلاس Role را توسعه داده باشید، یا سیستمی مانند Group-based Roles را پیاده سازی کرده باشید) این تغییرات سیستم قبلی شما را خواهد شکست.

بررسی پیاده سازی جدید IdentityUser از حوصله این مقاله خارج است. فعلا همین قدر بدانید که گرچه تعاریف پایه کلاس کاربران پیچیده‌تر شده است، اما انعطاف پذیری بسیار خوبی بدست آمده که شایان اهمیت فراوانی است.

از آنجا که کلاس ApplicationUser از IdentityUser ارث بری می‌کند، تمام خواص و تعاریف این کلاس پایه در ApplicationUser قابل دسترسی هستند.

کامپوننت‌های پیکربندی Identity 2.0 و کدهای کمکی

گرچه متد ConfigAuth() در کلاس Startup، محلی است که پیکربندی Identity در زمان اجرا صورت می‌پذیرد، اما در واقع کامپوننت‌های موجود در فایل *IdentityConfig.cs* هستند که اکثر قابلیت‌های Identity 2.0 را پیکربندی کرده و نحوه رفتار آنها در اپلیکیشن ما را کنترل می‌کنند.

اگر محتوای فایل *IdentityConfig.cs* را بررسی کنید خواهید دید که کلاس‌های متعددی در این فایل تعریف شده اند. می‌توان تک تک این کلاس‌ها را به فایل‌های مجزایی منتقل کرد، اما برای مثال جاری کدها را بهمین صورت رها کرده و نگاهی اجمالی به آنها خواهیم داشت. بهرحال در حال حاظر تمام این کلاس‌ها در فضای نام ApplicationName.Models قرار دارند.

Application User Manager و Application Role Manager

اولین چیزی که در این فایل به آنها بر می‌خوریم دو کلاس ApplicationUserManager و ApplicationRoleManager هستند. آماده باشید، مقدار زیادی کد با انواع داده جنریک در پیش روست!

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

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

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

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

        // Configure user lockout defaults
        manager.UserLockoutEnabledByDefault = true;
        manager.DefaultAccountLockoutTimeSpan = TimeSpan.FromMinutes(5);
        manager.MaxFailedAccessAttemptsBeforeLockout = 5;

        // Register two factor authentication providers. This application uses
        // Phone and Emails as a step of receiving a code for verifying
        // the user You can write your own provider and plug in here.
        manager.RegisterTwoFactorProvider("PhoneCode",
            new PhoneNumberTokenProvider<ApplicationUser>
        {
            MessageFormat = "Your security code is: {0}"
        });

        manager.RegisterTwoFactorProvider("EmailCode",
            new EmailTokenProvider<ApplicationUser>
        {
            Subject = "SecurityCode",
            BodyFormat = "Your security code is {0}"
        });

        manager.EmailService = new EmailService();
        manager.SmsService = new SmsService();

        var dataProtectionProvider = options.DataProtectionProvider;

        if (dataProtectionProvider != null)
        {
            manager.UserTokenProvider =
                new DataProtectorTokenProvider<ApplicationUser>(
                    dataProtectionProvider.Create("ASP.NET Identity"));
        }

        return manager;
    }

    public virtual async Task<IdentityResult> AddUserToRolesAsync(
        string userId, IList<string> roles)
    {
        var userRoleStore = (IUserRoleStore<ApplicationUser, string>)Store;
        var user = await FindByIdAsync(userId).ConfigureAwait(false);

        if (user == null)
        {
            throw new InvalidOperationException("Invalid user Id");
        }

        var userRoles = await userRoleStore
            .GetRolesAsync(user)
            .ConfigureAwait(false);

        // Add user to each role using UserRoleStore
        foreach (var role in roles.Where(role => !userRoles.Contains(role)))
        {
            await userRoleStore.AddToRoleAsync(user, role).ConfigureAwait(false);
```

```
        }

        // Call update once when all roles are added
        return await UpdateAsync(user).ConfigureAwait(false);
    }

    public virtual async Task<IdentityResult> RemoveUserFromRolesAsync(
        string userId, IList<string> roles)
    {
        var userRoleStore = (IUserRoleStore<ApplicationUser, string>) Store;
        var user = await FindByIdAsync(userId).ConfigureAwait(false);

        if (user == null)
        {
            throw new InvalidOperationException("Invalid user Id");
        }

        var userRoles = await userRoleStore
            .GetRolesAsync(user)
            .ConfigureAwait(false);

        // Remove user to each role using UserRoleStore
        foreach (var role in roles.Where(userRoles.Contains))
        {
            await userRoleStore
                .RemoveFromRoleAsync(user, role)
                .ConfigureAwait(false);
        }

        // Call update once when all roles are removed
        return await UpdateAsync(user).ConfigureAwait(false);
    }
}
```

قطعه کد بالا گرچه شاید به نظر طولانی بیاید، اما در کل، کلاس ApplicationUserManager توابع مهمی را برای مدیریت کاربران فراهم می‌کند: ایجاد کاربران جدید، انتساب کاربران به نقش‌ها و حذف کاربران از نقش ها. این کلاس بخودی خود از کلاس UserManager<ApplicationUser> ارث بری می‌کند بنابراین تمام قابلیت‌های UserManager در این کلاس هم در دسترس است. اگر به متد ()Create دقت کنید می‌بینید که وهله ای از نوع ApplicationUserManager باز می‌گرداند. بیشتر تنظیمات پیکربندی و تنظیمات پیش فرض کاربران شما در این متد اتفاق می‌افتد.

مورد حائز اهمیت بعدی در متد ()Create فراخوانی ()context.Get<ApplicationDBContext> است. بیاد بیاورید که پیشتر نگاهی به متد ()ConfigAuth داشتیم که چند فراخوانی CreatePerOwinContext داشت که توسط آنها Callback هایی را رجیستر می‌کردیم. فراخوانی متد ()context.Get<ApplicationDBContext> این Callbackها را صدا می‌زند، که در اینجا فراخوانی متد استاتیک ()ApplicationDbContext.Create خواهد بود. در ادامه بیشتر درباره این قسمت خواهید خواهند.

اگر دقت کنید می‌بینید که احراز هویت، تعیین سطوح دسترسی و تنظیمات مدیریتی و مقادیر پیش فرض آنها در متد ()Create انجام می‌شوند و سپس وهله ای از نوع خود کلاس ApplicationUserManager بازگشت داده می‌شود. همچنین سرویس‌های احراز هویت دو مرحله ای نیز در همین مرحله پیکربندی می‌شوند. اکثر پیکربندی‌ها و تنظیمات نیازی به توضیح ندارند و قابل درک هستند. اما احراز هویت دو مرحله ای نیاز به بررسی عمیق‌تری دارد. در ادامه به این قسمت خواهیم پرداخت. اما پیش از آن نگاهی به کلاس ApplicationRoleManager بیاندازیم.

```
public class ApplicationRoleManager : RoleManager<IdentityRole>
{
    public ApplicationRoleManager(IRoleStore<IdentityRole,string> roleStore)
        : base(roleStore)
    {
    }

    public static ApplicationRoleManager Create(
        IdentityFactoryOptions<ApplicationRoleManager> options,
        IOwinContext context)
    {
        var manager = new ApplicationRoleManager(
            new RoleStore<IdentityRole>(
                context.Get<ApplicationDbContext>()));

        return manager;
```

```
    }
}
```

مانند کلاس ApplicationUserManager مشاهده می‌کنید که کلاس ApplicationRoleManager از RoleManager<IdentityRole> ارث بری می‌کند. بنابراین تمام قابلیت‌های کلاس پایه نیز در این کلاس در دسترس هستند. یکبار دیگر متدی بنام ()Create را مشاهده می‌کنید که وهله ای از نوع خود کلاس بر می‌گرداند.

سرویس‌های ایمیل و پیامک برای احراز هویت دو مرحله ای و تایید حساب‌های کاربری

دو کلاس دیگری که در فایل *IdentityConfig.cs* وجود دارند کلاس‌های EmailService و SmsService هستند. بصورت پیش فرض این کلاس‌ها تنها یک wrapper هستند که می‌توانید با توسعه آنها سرویس‌های مورد نیاز برای احراز هویت دو مرحله ای و تایید حساب‌های کاربری را بسازید.

```
public class EmailService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        // Plug in your email service here to send an email.
        return Task.FromResult(0);
    }
}
```

```
public class SmsService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        // Plug in your sms service here to send a text message.
        return Task.FromResult(0);
    }
}
```

دقت کنید که هر دو این کلاس‌ها قرارداد واحدی را بنام IIdentityMessageService پیاده سازی می‌کنند. همچنین قطعه کد زیر را از متد ()ApplicationUserManager.Create بیاد آورید.

```
// Register two factor authentication providers. This application uses
// Phone and Emails as a step of receiving a code for verifying
// the user You can write your own provider and plug in here.
manager.RegisterTwoFactorProvider("PhoneCode",
    new PhoneNumberTokenProvider<ApplicationUser>
{
    MessageFormat = "Your security code is: {0}"
});

manager.RegisterTwoFactorProvider("EmailCode",
    new EmailTokenProvider<ApplicationUser>
{
    Subject = "SecurityCode",
    BodyFormat = "Your security code is {0}"
});

manager.EmailService = new EmailService();
manager.SmsService = new SmsService();
```

همانطور که می‌بینید در متد ()Create کلاس‌های EmailService و SmsService وهله سازی شده و توسط خواص مرتبط به وهله ApplicationUserManager ارجاع می‌شوند.

کلاس کمکی SignIn

هنگام توسعه پروژه مثال Identity، تیم توسعه دهندگان کلاسی کمکی برای ما ساخته‌اند که فرامین عمومی احراز هویت کاربران و ورود آنها به اپلیکیشن را توسط یک API ساده فراهم می‌سازد. برای آشنایی با نحوه استفاده از این متدها می‌توانیم به کنترلر AccountController در پوشه Controllers مراجعه کنیم. اما پیش از آن بگذارید نگاهی به خود کلاس SignInHelper داشته

باشیم.

```
public class SignInHelper
{
    public SignInHelper(
        ApplicationUserManager userManager,
        IAuthenticationManager authManager)
    {
        UserManager = userManager;
        AuthenticationManager = authManager;
    }

    public ApplicationUserManager UserManager { get; private set; }
    public IAuthenticationManager AuthenticationManager { get; private set; }

    public async Task SignInAsync(
        ApplicationUser user,
        bool isPersistent,
        bool rememberBrowser)
    {
        // Clear any partial cookies from external or two factor partial sign ins
        AuthenticationManager.SignOut(
            DefaultAuthenticationTypes.ExternalCookie,
            DefaultAuthenticationTypes.TwoFactorCookie);

        var userIdentity = await user.GenerateUserIdentityAsync(UserManager);

        if (rememberBrowser)
        {
            var rememberBrowserIdentity =
                AuthenticationManager.CreateTwoFactorRememberBrowserIdentity(user.Id);

            AuthenticationManager.SignIn(
                new AuthenticationProperties { IsPersistent = isPersistent },
                userIdentity,
                rememberBrowserIdentity);
        }
        else
        {
            AuthenticationManager.SignIn(
                new AuthenticationProperties { IsPersistent = isPersistent },
                userIdentity);
        }
    }

    public async Task<bool> SendTwoFactorCode(string provider)
    {
        var userId = await GetVerifiedUserIdAsync();

        if (userId == null)
        {
            return false;
        }

        var token = await UserManager.GenerateTwoFactorTokenAsync(userId, provider);

        // See IdentityConfig.cs to plug in Email/SMS services to actually send the code
        await UserManager.NotifyTwoFactorTokenAsync(userId, provider, token);

        return true;
    }

    public async Task<string> GetVerifiedUserIdAsync()
    {
        var result = await AuthenticationManager.AuthenticateAsync(
            DefaultAuthenticationTypes.TwoFactorCookie);

        if (result != null && result.Identity != null
            && !String.IsNullOrEmpty(result.Identity.GetUserId()))
        {
            return result.Identity.GetUserId();
        }

        return null;
    }

    public async Task<bool> HasBeenVerified()
    {
        return await GetVerifiedUserIdAsync() != null;
    }
```

```
public async Task<SignInStatus> TwoFactorSignIn(
    string provider,
    string code,
    bool isPersistent,
    bool rememberBrowser)
{
    var userId = await GetVerifiedUserIdAsync();

    if (userId == null)
    {
        return SignInStatus.Failure;
    }

    var user = await UserManager.FindByIdAsync(userId);

    if (user == null)
    {
        return SignInStatus.Failure;
    }

    if (await UserManager.IsLockedOutAsync(user.Id))
    {
        return SignInStatus.LockedOut;
    }

    if (await UserManager.VerifyTwoFactorTokenAsync(user.Id, provider, code))
    {
        // When token is verified correctly, clear the access failed
        // count used for lockout
        await UserManager.ResetAccessFailedCountAsync(user.Id);
        await SignInAsync(user, isPersistent, rememberBrowser);

        return SignInStatus.Success;
    }

    // If the token is incorrect, record the failure which
    // also may cause the user to be locked out
    await UserManager.AccessFailedAsync(user.Id);

    return SignInStatus.Failure;
}

public async Task<SignInStatus> ExternalSignIn(
    ExternalLoginInfo loginInfo,
    bool isPersistent)
{
    var user = await UserManager.FindAsync(loginInfo.Login);

    if (user == null)
    {
        return SignInStatus.Failure;
    }

    if (await UserManager.IsLockedOutAsync(user.Id))
    {
        return SignInStatus.LockedOut;
    }

    return await SignInOrTwoFactor(user, isPersistent);
}

private async Task<SignInStatus> SignInOrTwoFactor(
    ApplicationUser user,
    bool isPersistent)
{
    if (await UserManager.GetTwoFactorEnabledAsync(user.Id) &&
        !await AuthenticationManager.TwoFactorBrowserRememberedAsync(user.Id))
    {
        var identity = new ClaimsIdentity(DefaultAuthenticationTypes.TwoFactorCookie);
        identity.AddClaim(new Claim(ClaimTypes.NameIdentifier, user.Id));

        AuthenticationManager.SignIn(identity);
        return SignInStatus.RequiresTwoFactorAuthentication;
    }

    await SignInAsync(user, isPersistent, false);
    return SignInStatus.Success;
}

public async Task<SignInStatus> PasswordSignIn(
```

```
        string userName,
        string password,
        bool isPersistent,
        bool shouldLockout)
    {
        var user = await UserManager.FindByNameAsync(userName);

        if (user == null)
        {
            return SignInStatus.Failure;
        }

        if (await UserManager.IsLockedOutAsync(user.Id))
        {
            return SignInStatus.LockedOut;
        }

        if (await UserManager.CheckPasswordAsync(user, password))
        {
            return await SignInOrTwoFactor(user, isPersistent);
        }

        if (shouldLockout)
        {
            // If lockout is requested, increment access failed
            // count which might lock out the user
            await UserManager.AccessFailedAsync(user.Id);

            if (await UserManager.IsLockedOutAsync(user.Id))
            {
                return SignInStatus.LockedOut;
            }
        }

        return SignInStatus.Failure;
    }
}
```

کد این کلاس نسبتا طولانی است، و بررسی عمیق آنها از حوصله این مقاله خارج است. گرچه اگر به دقت یکبار این کلاس را مطالعه کنید می‌توانید براحتی از نحوه کارکرد آن آگاه شوید. همانطور که می‌بینید اکثر متدهای این کلاس مربوط به ورود کاربران و مسئولیت‌های تعیین سطوح دسترسی است.

این متدها ویژگی‌های جدیدی که در Identity 2.0 عرضه شده اند را در بر می‌گیرند. متد آشنایی بنام ()SignInAsync را می‌بینیم، و متدهای دیگری که مربوط به احراز هویت دو مرحله ای و external log-ins می‌شوند. اگر به متدها دقت کنید خواهید دید که برای ورود کاربران به اپلیکیشن کارهای بیشتری نسبت به نسخه پیشین انجام می‌شود.

بعنوان مثال متد Login در کنترلر AccountController را باز کنید تا نحوه مدیریت احراز هویت در Identity 2.0 را ببینید.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }

    // This doen't count login failures towards lockout only two factor authentication
    // To enable password failures to trigger lockout, change to shouldLockout: true
    var result = await SignInHelper.PasswordSignIn(
        model.Email,
        model.Password,
        model.RememberMe,
        shouldLockout: false);

    switch (result)
    {
        case SignInStatus.Success:
            return RedirectToLocal(returnUrl);
        case SignInStatus.LockedOut:
            return View("Lockout");
```

```
        case SignInStatus.RequiresTwoFactorAuthentication:
            return RedirectToAction("SendCode", new { ReturnUrl = returnUrl });
        case SignInStatus.Failure:
        default:
            ModelState.AddModelError("", "Invalid login attempt.");

            return View(model);
    }
}
```

مقایسه Sign-in با نسخه Identity 1.0

در نسخه 1.0 این فریم ورک، ورود کاربران به اپلیکیشن مانند لیست زیر انجام می‌شد. اگر متد Login در کنترلر AccountController را باز کنید چنین قطعه کدی را می‌بینید.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        var user = await UserManager.FindAsync(model.UserName, model.Password);

        if (user != null)
        {
            await SignInAsync(user, model.RememberMe);
            return RedirectToLocal(returnUrl);
        }
        else
        {
            ModelState.AddModelError("", "Invalid username or password.");
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

در قطعه کد بالا متدی در کلاس UserManager را فراخوانی می‌کنیم که مشابه قطعه کدی است که در کلاس SignInHelper دیدیم. همچنین متد SignInAsync را فراخوانی می‌کنیم که مستقیما روی کنترلر AccountController تعریف شده است.

```
private async Task SignInAsync(ApplicationUser user, bool isPersistent)
{
    AuthenticationManager.SignOut(
        DefaultAuthenticationTypes.ExternalCookie);

    var identity = await UserManager.CreateIdentityAsync(
        user, DefaultAuthenticationTypes.ApplicationCookie);

    AuthenticationManager.SignIn(
        new AuthenticationProperties() { IsPersistent = isPersistent }, identity);
}
```

مسلما با عرضه قابلیت‌های جدید در Identity 2.0 و تغییرات معماری این فریم ورک، پیچیدگی هایی معرفی می‌شود که حتی در امور ساده ای مانند ورود کاربران نیز قابل مشاهده است.

ApplicationDbContext

اگر از نسخه پیشین Identity در اپلیکیشن‌های ASP.NET MVC استفاده کرده باشید با کلاس ApplicationDbContext آشنا هستید. این کلاس پیاده سازی پیش فرض EF فریم ورک است، که اپلیکیشن شما توسط آن داده‌های مربوط به Identity را ذخیره و بازیابی می‌کند.

در پروژه مثال ها، تیم Identity این کلاس را بطور متفاوتی نسبت به نسخه 1.0 پیکربندی کرده اند. اگر فایل *IdentityModels.cs* را باز کنید تعاریف کلاس ApplicationDbContext را مانند لیست زیر خواهید یافت.

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser> {
    public ApplicationDbContext()
        : base("DefaultConnection", throwIfV1Schema: false) {
    }

    static ApplicationDbContext() {
        // Set the database intializer which is run once during application start
        // This seeds the database with admin user credentials and admin role
        Database.SetInitializer<ApplicationDbContext>(new ApplicationDbInitializer());
    }

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

قطعه کد بالا دو متد استاتیک تعریف می‌کند. یکی ()Create و دیگری ()ApplicationDbContext که سازنده دیتابیس (database initializer) را تنظیم می‌کند. این متد هنگام اجرای اپلیکیشن فراخوانی می‌شود و هر پیکربندی ای که در کلاس ApplicationDbInitializer تعریف شده باشد را اجرا می‌کند. اگر به فایل *IdentityConfig.cs* مراجعه کنیم می‌توانیم تعاریف این کلاس را مانند لیست زیر بیابیم.

```
public class ApplicationDbInitializer
    : DropCreateDatabaseIfModelChanges<ApplicationDbContext>
{
    protected override void Seed(ApplicationDbContext context)
    {
        InitializeIdentityForEF(context);
        base.Seed(context);
    }

    public static void InitializeIdentityForEF(ApplicationDbContext db)
    {
        var userManager = HttpContext
            .Current.GetOwinContext()
            .GetUserManager<ApplicationUserManager>();

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

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

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

        if (role == null)
        {
            role = new IdentityRole(roleName);
            var roleresult = roleManager.Create(role);
        }

        var user = userManager.FindByName(name);

        if (user == null)
        {
            user = new ApplicationUser { UserName = name, Email = name };

            var result = userManager.Create(user, password);
            result = userManager.SetLockoutEnabled(user.Id, false);
        }

        // Add user admin to Role Admin if not already added
        var rolesForUser = userManager.GetRoles(user.Id);

        if (!rolesForUser.Contains(role.Name))
        {
            var result = userManager.AddToRole(user.Id, role.Name);
        }
    }
}
```

پیکربندی جاری در صورتی که مدل موجودیت‌ها تغییر کنند دیتابیس را پاک کرده و مجددا ایجاد می‌کند. در غیر اینصورت از دیتابیس موجود استفاده خواهد شد. اگر بخواهیم با هر بار اجرای اپلیکیشن دیتابیس از نو ساخته شود، می‌توانیم کلاس مربوطه را به DropCreateDatabaseAlways<ApplicationDbContext> تغییر دهیم. بعنوان مثال هنگام توسعه اپلیکیشن و بمنظور تست می‌توانیم از این رویکرد استفاده کنیم تا هر بار با دیتابیسی (تقریبا) خالی شروع کنیم.

نکته حائز اهمیت دیگر متد InitializeIdentityForEF() است. این متد کاری مشابه متد Seed() انجام می‌دهد که هنگام استفاده از مهاجرت‌ها (Migrations) از آن استفاده می‌کنیم. در این متد می‌توانید رکوردهای اولیه ای را در دیتابیس ثبت کنید. همانطور که مشاهده می‌کنید در قطعه کد بالا نقشی مدیریتی بنام Admin ایجاد شده و کاربر جدیدی با اطلاعاتی پیش فرض ساخته می‌شود که در آخر به این نقش منتسب می‌گردد. با انجام این مراحل، پس از اجرای اولیه اپلیکیشن کاربری با سطح دسترسی مدیر در اختیار خواهیم داشت که برای تست اپلیکیشن بسیار مفید خواهد بود.

در این مقاله نگاهی اجمالی به Identity 2.0 در پروژه‌های ASP.NET MVC داشتیم. کامپوننت‌های مختلف فریم ورک و نحوه پیکربندی آنها را بررسی کردیم و با تغییرات و قابلیت‌های جدید به اختصار آشنا شدیم. در مقالات بعدی بررسی هایی عمیق‌تر خواهیم داشت و با نحوه استفاده و پیاده سازی قسمت‌های مختلف این فریم ورک آشنا خواهیم شد.

مطالعه بیشتر

MSDN: Announcing RTM of ASP.NET Identity 2.0.0

Identity 2.0 on Codeplex

نظرات خوانندگان

نویسنده: محمّد زارع
تاریخ: ۱۳۹۳/۰۶/۰۷ ۱۹:۷

سلام. من از ASP.NET Identity توی پروژم استفاده کردم و همه چیز درست کار میکنه (در محیط توسعه). سوالی که دارم مربوط به بررسی نقش‌های کاربر هستش. میخواستم بدونم استفاده از HttpContext.Current.User.IsInRole توی ASP.NET Identity هم درست هست یا نه؟ من الان توی محیط توسعه هیچ مشکلی در استفاده ازش نمیبینم ولی چندجا توی اینترنت دیدم که گفته بودن استفاده از این روش قدیمی هستش و بعد از Publish کردن به مشکل میخوره و بهتر هستش که از UserManager.IsInRole استفاده بشه.
ممنون میشم اگر راهنمایی بفرمایید. https://aspnetidentity.codeplex.com/workitem/2189
http://codeverge.com/asp.net.security/user.isinrole-not-working-in-production/73992

نویسنده: هومن
تاریخ: ۱۳۹۳/۰۶/۱۱ ۱۳:۲۴

سلام
نمونه پروژه ای وجود داره که با استفاده از identity 2.0 سیستم گروپ بیس رو پیاده سازی کرده باشه؟
نمونه identity 1.0 رو پیدا کردم ولی هرکاری کردم نتونستم تبدیلش کنم به نسخه دوم

نویسنده: ایلیا
تاریخ: ۱۳۹۳/۰۸/۱۹ ۱۳:۲۰

اگه در حالت EF Code First نباشیم و دیتابیسمون به صورت Database First باشه و دیتااکسسمون ADO Entity Data Model یا همون edmx باشه.چطوری می‌تونیم از Identity استفاده کنیم؟(البته از MVC 5 و EF 6 دارم استفاده می‌کنم
و دوم اینکه جداول لازم رو خودمون باید بسازیم یا مثل Membership خودش میسازه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۸/۱۹ ۱۴:۰

- Use ASP.NET Identity on existing DB-Model PART 1
- ASP.NET Identity 2.0 on existing DB-model PART 2

نویسنده: نرگس حقیقی
تاریخ: ۱۳۹۳/۰۹/۲۲ ۱۲:۳

سلام؛ آیا میشه در یه پروژه MVC که از EF6 برای ارتباط با Sql Server استفاده میکنه و بدلیل زیاد بودن تعداد جداول از class Library جداگانه‌ای به‌منظور Migrate کردن بانک استفاده میکنه، از Identity هم استفاده کرد؛ با وجود اینکه به Identity Code کاربران در جداول دیگر هم نیاز هست؟ اصلا این کار ممکنه؟

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۳/۰۹/۲۲ ۱۸:۵۳

سرویس خوبیه اما کمی کار باهاش در عین سادگی پیچیده اس.
مثلا پیاده سازی اون و تفکیک تو لایه‌های سرویس و دامین و لایه نمایش واقعا ادم گیج میکنه
هر چند زیاد هم نمی‌توان الگوی واحد کار باهاش به کار برد - البته نظر شخصی منه

نویسنده: وحید نصیری

تاریخ: ۱۳۹۳/۰۹/۲۹ ۱۳:۴۲

« اعمال تزریق وابستگی‌ها به مثال رسمی ASP.NET Identity »

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۲۹ ۱۳:۴۳

بله. مراجعه کنید به این مثال: « اعمال تزریق وابستگی‌ها به مثال رسمی ASP.NET Identity »