عنوان: **معرفی ASP.NET Identity**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۴ ۹:۴۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, Entity framework, MVC, Security, ASP.NET Identity

سیستم ASP.NET Membership بهمراه ASP.NET 2.0 در سال 2005 معرفی شد، و از آن زمان تا بحال تغییرات زیادی در چگونگی مدیریت احزار هویت و اختیارات کاربران توسط اپلیکیشن‌های وب بوجود آمده است. ASP.NET Identity نگاهی تازه است به آنچه که سیستم Membership هنگام تولید اپلیکیشن‌های مدرن برای وب، موبایل و تبلت باید باشد.

پیش زمینه: سیستم عضویت در ASP.NET

ASP.NET Membership

ASP.NET Membership طراحی شده بود تا نیازهای سیستم عضویت وب سایت‌ها را تامین کند، نیازهایی که در سال 2005 رایج بود و شامل مواردی مانند مدل احراز هویت فرم، و یک پایگاه داده SQL Server برای ذخیره اطلاعات کاربران و پروفایل هایشان می‌شد. امروزه گزینه‌های بسیار بیشتری برای ذخیره داده‌های وب اپلیکیشن‌ها وجود دارد، و اکثر توسعه دهندگان می‌خواهند از اطلاعات شبکه‌های اجتماعی نیز برای احراز هویت و تعیین سطوح دسترسی کاربرانشان استفاده کنند. محدودیت‌های طراحی سیستم ASP.NET Membership گذر از این تحول را دشوار می‌کند:

الگوی پایگاه داده آن برای SQL Server طراحی شده است، و قادر به تغییرش هم نیستید. می‌توانید اطلاعات پروفایل را اضافه کنید، اما تمام داده‌ها در یک جدول دیگر ذخیره می‌شوند، که دسترسی به آنها نیز مشکل‌تر است، تنها راه دسترسی Profile Provider API خواهد بود.

سیستم تامین کننده (Provider System) امکان تغییر منبع داده‌ها را به شما می‌دهد، مثلا می‌توانید از بانک‌های اطلاعاتی MySQL یا Oracle استفاده کنید. اما تمام سیستم بر اساس پیش فرض هایی طراحی شده است که تنها برای بانک‌های اطلاعاتی relational درست هستند. می‌توانید تامین کننده (Provider) ای بنویسید که داده‌های سیستم عضویت را در منبعی به غیر از دیتابیس‌های relational ذخیره می‌کند؛ مثلا Windows Azure Storage Tables. اما در این صورت باید مقادیر زیادی کد بنویسید. مقادیر زیادی هم System.NotImplementedException باید بنویسید، برای متد هایی که به دیتابیس‌های NoSQL مربوط نیستند.

از آنجایی که سیستم ورود/خروج سایت بر اساس مدل Forms Authentication کار می‌کند، سیستم عضویت نمی‌تواند از OWIN استفاده کند. OWIN شامل کامپوننت هایی برای احراز هویت است که شامل سرویس‌های خارجی هم می‌شود (مانند Microsoft Accounts, Facebook, Google, Twitter). همچنین امکان ورود به سیستم توسط حساب‌های کاربری سازمانی (Organizational Accounts) نیز وجود دارد مانند Active Directory و Windows Azure Active Directory. این کتابخانه از JWT، OAuth 2.0 و CORS نیز پشتیبانی می‌کند.

ASP.NET Simple Membership

ASP.NET simple membership به عنوان یک سیستم عضویت، برای فریم ورک Web Pages توسعه داده شد. این سیستم با WebMatrix و Visual Studio 2010 SP1 انتشار یافت. هدف از توسعه این سیستم، آسان کردن پروسه افزودن سیستم عضویت به یک اپلیکیشن Web Pages بود. این سیستم پروسه کلی کار را آسان‌تر کرد، اما هنوز مشکلات ASP.NET Membership را نیز داشت. محدودیت هایی نیز وجود دارند:

ذخیره داده‌های سیستم عضویت در بانک‌های اطلاعاتی non-relational مشکل است.

نمی توانید از آن در کنار OWIN استفاده کنید.

با فراهم کننده‌های موجود ASP.NET Membership بخوبی کار نمی‌کند. توسعه پذیر هم نیست.

ASP.NET Universal Providers

ASP.NET Universal Providers برای ذخیره سازی اطلاعات سیستم عضویت در Windows Azure SQL Database توسعه پیدا کردند. با SQL Server Compact هم بخوبی کار می‌کنند. این تامین کننده‌ها بر اساس Entity Framework Code First ساخته شده بودند و بدین معنا بود که داده‌های سیستم عضویت را می‌توان در هر منبع داده ای که توسط EF پشتیبانی می‌شود ذخیره کرد. با انتشار این تامین کننده‌ها الگوی دیتابیس سیستم عضویت نیز بسیار سبک‌تر و بهتر شد. اما این سیستم بر پایه زیر ساخت ASP.NET

Membership نوشته شده است، بنابراین محدودیت‌های پیشین مانند محدودیت‌های SqlMembershipProvider هنوز وجود دارند. به بیان دیگر، این سیستم‌ها همچنان برای بانک‌های اطلاعاتی relational طراحی شده اند، پس سفارشی سازی اطلاعات کاربران و پروفایل‌ها هنوز مشکل است. در آخر آنکه این تامین کننده‌ها هنوز از مدل احراز هویت فرم استفاده می‌کنند.

ASP.NET Identity همانطور که داستان سیستم عضویت ASP.NET طی سالیان تغییر و رشد کرده است، تیم ASP.NET نیز آموخته‌های زیادی از بازخوردهای مشتریان شان بدست آورده اند. این پیش فرض که کاربران شما توسط یک نام کاربری و کلمه عبور که در اپلیکیشن خودتان هم ثبت شده است به سایت وارد خواهند شد، دیگر معتبر نیست. دنیای وب اجتماعی شده است. کاربران از طریق وب سایت‌ها و شبکه‌های اجتماعی متعددی با یکدیگر در تماس هستند، خیلی از اوقت بصورت زنده! شبکه هایی مانند Facebook و Twitter. همانطور که توسعه نرم افزارهای تحت وب رشد کرده است، الگوها و مدل‌های پیاده سازی نیز تغییر و رشد کرده اند. امکان Unit Testing روی کد اپلیکیشن‌ها، یکی از مهم‌ترین دلواپسی‌های توسعه دهندگان شده است. در سال 2008 تیم ASP.NET فریم ورک جدیدی را بر اساس الگوی Model-View-Controller (MVC) اضافه کردند. هدف آن کمک به توسعه دهندگان، برای تولید برنامه‌های ASP.NET با قابلیت Unit Testing بهتر بود. توسعه دهندگانی که می‌خواستند کد اپلیکیشن‌های خود را Unit Test کنند، همین امکان را برای سیستم عضویت نیز می‌خواستند.

با در نظر گرفتن تغییراتی که در توسعه اپلیکیشن‌های وب بوجود آمده ASP.NET Identity با اهداف زیر متولد شد:

یک سیستم هویت واحد (One ASP.NET Identity system)

سیستم ASP.NET Identity می‌تواند در تمام فریم ورک‌های مشتق از ASP.NET استفاده شود. مانند ASP.NET MVC, Web Forms, Web Pages, Web API و SignalR

از این سیستم می‌توانید در تولید اپلیکیشن‌های وب، موبایل، استور (Store) و یا اپلیکیشن‌های ترکیبی استفاده کنید.

سادگی تزریق داده‌های پروفایل درباره کاربران

روی الگوی دیتابیس برای اطلاعات کاربران و پروفایل‌ها کنترل کامل دارید. مثلا می‌توانید به سادگی یک فیلد، برای تاریخ تولد در نظر بگیرید که کاربران هنگام ثبت نام در سایت باید آن را وارد کنند.

کنترل ذخیره سازی/واکشی اطلاعات

بصورت پیش فرض ASP.NET Identity تمام اطلاعات کاربران را در یک دیتابیس ذخیره می‌کند. تمام مکانیزم‌های دسترسی به داده‌ها توسط EF Code First کار می‌کنند.

از آنجا که روی الگوی دیتابیس، کنترل کامل دارید، تغییر نام جداول و یا نوع داده فیلدهای کلیدی و غیره ساده است.

استفاده از مکانیزم‌های دیگر برای مدیریت داده‌های آن ساده است، مانند SharePoint, Windows Azure Storage Table و دیتابیس‌های NoSQL.

تست پذیری

ASP.NET Identity تست پذیری اپلیکیشن وب شما را بیشتر می‌کند. می‌توانید برای تمام قسمت هایی که از ASP.NET Identity استفاده می‌کنند تست بنویسید.

تامین کننده نقش (Role Provider)

تامین کننده ای وجود دارد که به شما امکان محدود کردن سطوح دسترسی بر اساس نقوش را می‌دهد. بسادگی می‌توانید نقش‌های جدید مانند "Admin" بسازید و بخش‌های مختلف اپلیکیشن خود را محدود کنید.

Claims Based

ASP.NET Identity از امکان احراز هویت بر اساس Claims نیز پشتیبانی می‌کند. در این مدل، هویت کاربر بر اساس دسته ای از اختیارات او شناسایی می‌شود. با استفاده از این روش توسعه دهندگان برای تعریف هویت کاربران، آزادی عمل بیشتری نسبت به مدل Roles دارند. مدل نقش‌ها تنها یک مقدار منطقی (bool) است؛ یا عضو یک نقش هستید یا خیر، در حالیکه با استفاده از روش Claims می‌توانید اطلاعات بسیار ریز و دقیقی از هویت کاربر در دست داشته باشید.

تامین کنندگان اجتماعی

به راحتی می‌توانید از تامین کنندگان دیگری مانند Microsoft, Facebook, Twitter, Google و غیره استفاده کنید و اطلاعات مربوط به کاربران را در اپلیکیشن خود ذخیره کنید.

Windows Azure Active Directory

برای اطلاعات بیشتر به این لینک مراجعه کنید.

یکپارچگی با OWIN

ASP.NET Identity بر اساس OWIN توسعه پیدا کرده است، بنابراین از هر میزبانی که از OWIN پشتیبانی می‌کند می‌توانید استفاده کنید. همچنین هیچ وابستگی ای به System.Web وجود ندارد. ASP.NET Identity یک فریم ورک کامل و مستقل برای OWIN است و می‌تواند در هر اپلیکیشنی که روی OWIN میزبانی شده استفاده شود.

ASP.NET Identity از OWIN برای ورود/خروج کاربران در سایت استفاده می‌کند. این بدین معنا است که بجای استفاده از Forms Authentication برای تولید یک کوکی، از OWIN CookieAuthentication استفاده می‌شود.

پکیج NuGet

ASP.NET Identity در قالب یک بسته NuGet توزیع می‌شود. این بسته در قالب پروژه‌های ASP.NET MVC, Web Forms و Web API که با Visual Studio 2013 منتشر شدند گنجانده شده است.

توزیع این فریم ورک در قالب یک بسته NuGet این امکان را به تیم ASP.NET می‌دهد تا امکانات جدیدی توسعه دهند، باگ‌ها را برطرف کنند و نتیجه را بصورت چابک به توسعه دهندگان عرضه کنند.

شروع کار با ASP.NET Identity

ASP.NET Identity در قالب پروژه‌های ASP.NET MVC, Web Forms, Web API و SPA که بهمراه Visual Studio 2013 منتشر شده اند استفاده می‌شود. در ادامه به اختصار خواهیم دید که چگونه ASP.NET Identity کار می‌کند.

یک پروژه جدید ASP.NET MVC با تنظیمات Individual User Accounts بسازید.

پروژه ایجاد شده شامل سه بسته می‌شود که مربوط به ASP.NET Identity هستند:

Microsoft.AspNet.Identity.EntityFramework این بسته شامل پیاده سازی ASP.NET Identity با Entity Framework می‌شود، که تمام داده‌های مربوطه را در یک دیتابیس SQL Server ذخیره می‌کند.
Microsoft.AspNet.Identity.Core این بسته محتوی تمام interfaceهای ASP.NET Identity است. با استفاده از این بسته می‌توانید پیاده سازی دیگری از ASP.NET Identity بسازید که منبع داده متفاوتی را هدف قرار می‌دهد. مثلا Windows Azure Storage Table و دیتابیس‌های NoSQL.

Microsoft.AspNet.Identity.OWIN این بسته امکان استفاده از احراز هویت OWIN را در اپلیکیشن‌های ASP.NET فراهم می‌کند. هنگام تولید کوکی‌ها از OWIN Cookie Authentication استفاده خواهد شد.

اپلیکیشن را اجرا کرده و روی لینک Register کلیک کنید تا یک حساب کاربری جدید ایجاد کنید.

Register - My ASP.NET Ap
localhost:1234/Account/Register
Application name
Register.
Create a new account.
User name
Password
Confirm password
Register
© 2013 - My ASP.NET Application

هنگامیکه بر روی دکمه‌ی Register کلیک شود، کنترلر Account، اکشن متد Register را فراخوانی می‌کند تا حساب کاربری جدیدی با استفاده از ASP.NET Identity API ساخته شود.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser() { UserName = model.UserName };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            await SignInAsync(user, isPersistent: false);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            AddErrors(result);
        }
    }

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

اگر حساب کاربری با موفقیت ایجاد شود، کاربر توسط فراخوانی متد SignInAsync به سایت وارد می‌شود.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser() { UserName = model.UserName };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            await SignInAsync(user, isPersistent: false);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            AddErrors(result);
        }
    }

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

```
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);
}
```

از آنجا که ASP.NET Identity و OWIN Cookie Authentication هر دو Claims-based هستند، فریم ورک، انتظار آبجکتی از نوع ClaimsIdentity را خواهد داشت. این آبجکت تمامی اطلاعات لازم برای تشخیص هویت کاربر را در بر دارد. مثلا اینکه کاربر

مورد نظر به چه نقش هایی تعلق دارد؟ و اطلاعاتی از این قبیل. در این مرحله می‌توانید Claimهای بیشتری را به کاربر بیافزایید.

کلیک کردن روی لینک Log off در سایت، اکشن متد LogOff در کنترلر Account را اجرا می‌کند.

```
// POST: /Account/LogOff
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult LogOff()
{
    AuthenticationManager.SignOut();
    return RedirectToAction("Index", "Home");
}
```

همانطور که مشاهده می‌کنید برای ورود/خروج کاربران از AuthenticationManager استفاده می‌شود که متعلق به OWIN است. متد SignOut همتای متد FormsAuthentication.SignOut است.

کامپوننت‌های ASP.NET Identity

تصویر زیر اجزای تشکیل دهنده ASP.NET Identity را نمایش می‌دهد. بسته هایی که با رنگ سبز نشان داده شده اند سیستم کلی ASP.NET Identity را می‌سازند. مابقی بسته‌ها وابستگی هایی هستند که برای استفاده از ASP.NET Identity در اپلیکیشن‌های ASP.NET لازم اند.

دو پکیج دیگر نیز وجود دارند که به آنها اشاره نشد:

Microsoft.Security.Owin.Cookies این بسته امکان استفاده از مدل احراز هویت مبتنی بر کوکی (Cookie-based Authentication) را فراهم می‌کند. مدلی مانند سیستم ASP.NET Forms Authentication.

EntityFramework که نیازی به معرفی ندارد.

مهاجرت از Membership به ASP.NET Identity

تیم ASP.NET و مایکروسافت هنوز راهنمایی رسمی، برای این مقوله ارائه نکرده اند. گرچه پست‌های وبلاگ‌ها و منابع مختلفی وجود دارند که از جنبه‌های مختلفی به این مقوله پرداخته اند. امیدواریم تا در آینده نزدیک مایکروسافت راهنمایی‌های لازم را منتشر کند، ممکن است ابزار و افزونه هایی نیز توسعه پیدا کنند. اما در یک نگاه کلی می‌توان گفت مهاجرت بین این دو فریم ورک زیاد ساده نیست. تفاوت‌های فنی و ساختاری زیادی وجود دارند، مثلا الگوی دیتابیس‌ها برای ذخیره اطلاعات کاربران، مبتنی بودن بر فریم

ورک OWIN و غیره. اگر قصد اجرای پروژه جدیدی را دارید پیشنهاد می‌کنم از فریم ورک جدید مایکروسافت ASP.NET Identity استفاده کنید.

قدم‌های بعدی Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 and OpenID Sign-on

در این مقاله خواهید دید چگونه اطلاعات پروفایل را اضافه کنید و چطور از ASP.NET Identity برای احراز هویت کاربران توسط Facebook و Google استفاده کنید. Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to a Windows Azure Web Site

Individual User Accounts

Organizational Accounts

Customizing profile information in ASP.NET Identity in VS 2013 templates

Get more information from Social providers used in the VS 2013 project templates

https://github.com/rustd/AspnetIdentitySample

پروژه نمونه ASP.NET Identity می‌تواند مفید باشد. در این پروژه نحوه کارکردن با کاربران و نقش‌ها و همچنین نیازهای مدیریتی رایج نمایش داده شده.

نظرات خوانندگان

نویسنده: ناظم
تاریخ: ۱۳۹۲/۱۰/۱۴ ۱۱:۴۶

سلام دوست عزیز
به خاطر این مطلب بسیار خوبتون سپاسگذارم.
آیا کتاب یا راهنمای جامعی برای کسی مثل من که اصلا تجربه کار با هیچ کدام از این سیستم‌های مدیریت کاربر رو نداره سراغ دارین؟ لینکهای بالا همشون موردی هستن

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۱۴ ۱۲:۳۰

مباحث ابتدایی Forms Authentication مربوط است به ASP.NET 1.x؛ دوره‌ای در این مورد . همچنین در MVC هم قابل استفاده است (^ و ^). مباحث membership هم مربوط است به ASP.NET 2.x. کتابی در این مورد .

نویسنده: مهران
تاریخ: ۱۳۹۲/۱۰/۱۴ ۲۱:۳۹

سلام
با تشکر از مطالب و ارائه خوبتون؛
خواستم بدونم چطور میشه، کاربر را بر اساس عملیات کنترلر (Actions) تعیین دسترسی کرد؟

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۴ ۲۱:۴۴

تا بحال با کتاب یا دوره جامعی درباره ASP.NET Identity مواجه نشدم، اگر منبع مناسبی پیدا کنم به اشتراک میذارم. در چند پست آتی بیشتر درباره این فریم ورک صحبت خواهم کرد و مثال هایی عملی نیز در نظر خواهم گرفت

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۴ ۲۳:۲۷

سوالتون رو دقیقا متوجه نشدم. از خاصیت Authorize میتونید استفاده کنید، که قابل اعمال بر روی تک تک اکشن متدها و یا کل کنترلر است. خاصیت AllowAnonymous برای دسترسی عمومی استفاده می‌شود. برای اطلاعات بیشتر درباره نحوه استفاده از ASP.NET Identity و ساختار کلی OWIN لطفا به لینک‌های ضمیمه شده مراجعه کنید.

```
[Authorize]
public controller account
{
    public ActionResult Index()  { }
    public ActionResult Manage() { }

    [AllowAnonymous]
    public ActionResult Info() { }
}

[Authorize(Roles="Admin")]
public controller admin
{
    public ActionResult Index() { }
}
```

نویسنده: مهران
تاریخ: ۱۳۹۲/۱۰/۱۵ ۰:۱

فکر میکنم با مطالعه بیشتر Claims Based که در مطلب اشاره کردید، به راه حل مورد نظر برسم.

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۵ ۲:۱۶

درسته، درباره Claims-based authorization هم یک یا دو پست می‌نویسم.

نویسنده: وحید
تاریخ: ۱۳۹۲/۱۰/۲۰ ۲:۱۸

با تشکر از مطلب مفیدتون سوالی داشتم
یک جا گفتید " مثلا الگوی دیتابیس‌ها برای ذخیره اطلاعات کاربران " منظورتون از دیتابیس Table هست؟
چرا از کد زیر task برای خروجی استفاده کردید

```
async Task<ActionResult> Register
```

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۲۱ ۶:۴۹

منظور دیتابیس سیستم عضویت است، همانطور که گفته شد این دیتابیس توسط EF ساخته می‌شود، بنابراین جداول، فیلدها و دیگر موارد را میتوانید سفارشی کنید.

همانطور که از امضای این متد مشخص است، عملیات بصورت Async پردازش می‌شوند. برای اطلاعات بیشتر به این لینک نمونه مراجعه کنید.

نویسنده: Programmer
تاریخ: ۱۳۹۲/۱۰/۲۳ ۱۷:۱۸

با عرض سلام و تشکر بابت پست‌های مفید که یقیناً خیلی براش زحمت می‌کشید.

اگر ممکنه کمی در مورد ساختار Identity و کلاس‌ها و اینترفیس‌ها و نحوه کار با اونها بصورتی که بتونیم خودمون هم Custom Implement اش رو انجام بدیم توضیح بدید.

اینکه اینترفیس هایی چون IUser و IUserStore و IUserPasswordStore و IUserSecurityStampStore و در طرف دیگه UserManager و UserStore و IdentityUser چی هستند و با چه هدفی تعریف شدند و قراره چیکار کنند و در آخر برای اینکه مطابق آموزش‌هایی که در سایت هست در مورد MVC و تعریف لایه‌های مختلف و سرویس‌های مورد نیاز، چطور باید از Identity استفاده کرد که هماهنگ با اون مطالب باشه؟

ممنون

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۲۳ ۲۰:۴

در پست‌های آتی پیاده سازی یک تامین کننده (Provider) برای MySQL رو بررسی می‌کنم. برای اطلاعات بیشتر به Implementing a Custom MySQL ASP.NET Identity Storage Provider مراجعه کنید.

هنوز مستندات کامل و رسمی برای این فریم ورک عرضه نشده اما مطالب مفید زیادی در اینترنت وجود دارند. چند لینک در زیر لیست شده:

ASP.NET Core Identity

The good, the bad and the ugly of ASP.NET Identity

درباره تطابق با آموزش‌های سایت: دقیقا متوجه نشدم منظورتون کدوم الگوها است، اما چند نکته تکمیلی:
بصورت پیش فرض وقتی ASP.NET Identity به پروژه اضافه میشه کلاسی بنام ApplicationDbContext ایجاد میشه که بعنوان DbContext پیش فرض برای دیتابیس عضویت استفاده خواهد شد. اگر موجودیت جدیدی برای پروفایل کاربران بسازید باید بعنوان یک DbSet<T> به این کلاس افزوده بشه. اگر نیازی به تفکیک دیتابیس سیستم عضویت و دیتابیس اپلیکیشن نیست، بهتره از یک DbContext استفاده کنید. لایه بندی مدل ها، سرویس‌ها و کد دسترسی به داده‌ها هم ساده است چرا که کل سیستم توسط EF Code First مدیریت میشه. بنابراین استفاده از الگوهای رایج مانند تزریق وابستگی‌ها و غیره مشابه دیگر سناریوهای EF Code First است.

نویسنده: کامران سادین
تاریخ: ۱۳۹۲/۱۰/۲۹ ۱۷:۱

با سلام.
بنده می‌خواستم بدونم آیا در .NET ورژن 4 هم میتوانیم استفاده کنیم؟
با ویژوال استادیو 2012

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۲۹ ۱۸:۴۰

با نصب پکیج‌های مربوط به ASP.NET Identity و غیرفعال کردن Forms Auth می‌تونید همچین کاری بکنید اما توصیه نمیشه.
سیستم Identity اکثر عملیاتش رو بصورت Async انجام میده که نیاز به NET 4.5. داره. دلایل دیگه ای هم وجود داره که اگر یک جستجوی ساده در اینترنت بکنید مطالب خوبی در این باره پیدا می‌کنید، مثلا لینک زیر:

http://stackoverflow.com/questions/19237285/using-asp-net-identity-in-mvc-4

نویسنده: سوران
تاریخ: ۱۳۹۲/۱۲/۰۱ ۱۲:۳۷

با تشکر از مطالب آموزنده شما ،
من در ارتباط با ارث بری از کلاس IdentityUser یک سوال داشتم. با توجه به نمونه کدهای تمپلیت vs2013 یک کلاس ApplicationUser از کلاس IdentityUser ارث بری می‌کنه و DbContext هم مربوط به این کلاس میشه، یعنی به صورت زیر

```
ApplicationDbContext : IdentityDbContext<ApplicationUser>
```

حال سوال من اینه که اگه چند کلاس داشته باشیم که بخوایم از IdentityUser ارث بری داشته باشند، چطور میشه اونها را در یک DbContext استفاده کرد؟
اگر مقاله یا مثالی در این مورد معرفی کنید ممنون میشم .
با تشکر

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۲/۰۳ ۵:۲۴

کلاس کاربر:

```
public class AppUser : IdentityUser
{
    public string Email { get; set; }
    public string ConfirmationToken { get; set; }
    public bool IsConfirmed { get; set; }

    public virtual UserProfile Profile { get; set; }
}
```

کلاس پروفایل کاربر:

```
public class UserProfile
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public DateTime? Birthday { get; set; }

    public byte[] Avatar { get; set; }
}
```

کلاس کانتکست دیتابیس:

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

    static SampleDbContext()
    {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<SampleDbContext>());
    }

    public DbSet<UserProfile> UserProfiles { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Product> Products { get; set; }
    ...
}
```

این کلاس‌ها می‌تونن تو لایه دیگری مثل Domain Models تعریف بشن.

نویسنده: رضا گرمارودی
تاریخ: ۱۳۹۲/۱۲/۰۳ ۹:۲۱

سلام؛ Identity در کار با SQLCE مشکل داره! هنگام چک کردن نام کاربری و کلمه عبور پیغامی مبنی بر استفاده از تابع Lowercase میدهد که گویا در SqlCe به شیوه دیگری باید صدا زده شود !

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۲/۰۳ ۹:۳۶

این پروژه سورس باز هست. مشکلات آن‌را برای رفع در اینجا گزارش کنید. نحوه‌ی گزارش مشکل هم باید کمی فنی باشد. حداقل جزئیات exception و stack trace آن باید گزارش شوند.

نویسنده: احمد
تاریخ: ۱۳۹۳/۰۲/۲۳ ۱۰:۲۸

سلام
آیا در پروژه‌های windows application که از wcf استفاده میکنند هم میتوانیم از این سیستم استفاده کنیم.؟

نویسنده: داریوش حمیدی
تاریخ: ۱۳۹۳/۰۷/۱۸ ۲۰:۴

سلام ; در کلاس IdentityUser این خصوصیت دو به چی اشاره دارند ؟

1-SecurityStamp

این مقدار Random تغیر می‌کنید وقتی که اطلاعات کاربری تغیر پیدا می‌کند مثل تغیر رمز عبور ..

2-TwoFactorEnabled

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۲۹ ۱۳:۴۶

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

عنوان: **متدهای احراز هویت در VS 2013**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۵ ۲:۱۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, MVC, Authentication, ASP.NET Identity, Security

ویژوال استودیو 2013 چندین گزینه برای احراز هویت در قالب‌های پیش فرض پروژه‌های ASP.NET Web Forms, MVC, Web API ارائه می‌کند:

No Authentication

Individual User Accounts

Organizational Accounts

Windows Authentication

No Authentication

اگر گزینه **No Authentication** را انتخاب کنید، پروژه ایجاد شده صفحاتی را برای ورود به سایت نخواهد ساخت. همچنین رابط کاربری ای برای نمایش کاربر فعلی، کلاس‌های موجودیت‌ها برای یک دیتابیس عضویت و رشته‌های اتصالی نیز وجود نخواهند داشت.

Individual User Accounts

اگر گزینه‌ی **Individual User Accounts** را انتخاب کنید، اپلیکیشن شما برای استفاده از ASP.NET Identity (که پیش از این با نام ASP.NET Membership شناخته می‌شد) پیکربندی می‌شود. ASP.NET Identity کاربران را قادر می‌سازد تا با ساختن حساب کاربری جدیدی در سایت و یا با استفاده از تامین کننده‌های ثالثی مانند Facebook, Google و غیره به سایت وارد شوند. این فریم ورک برای ذخیره‌ی داده‌های پروفایل کاربران، بصورت پیش فرض از یک دیتابیس SQL Server LocalDB استفاده می‌کند که می‌توانید بعدا آنرا بر روی SQL Server یا Windows Azure SQL Database نیز منتشر کنید.

این قابلیت‌ها در Visual Studio 2013 در نسخه قبلی نیز وجود داشتند، اما کد سیستم عضویت آن مجددا بازنویسی شده‌است. این بازنویسی دارای مزایای زیر است:

سیستم عضویت جدید بجای استفاده از ماژول ASP.NET Forms Authentication بر پایه OWIN نوشته شده است. این بدین معنا است که از یک مکانیزم احراز هویت واحد می‌توانید در اپلیکیشن‌های ASP.NET Web Forms, MVC, Web API و SignalR استفاده کنید. سیستم عضویت جدید توسط Entity Framework Code First مدیریت می‌شود و شامل تمامی کلاس‌هایی است که نماینده جداول و موجودیت‌ها هستند. این بدین معنا است که روی الگوی دیتابیس کنترل کامل دارید. سفارشی سازی و تغییر اطلاعات کاربران و پروفایل هایشان بسیار ساده‌تر است، تنها لازم است برای اعمال تغییرات از Code First Migrations استفاده کنید.

سیستم عضویت جدید بصورت خودکار در تمام قالب‌های پروژه پیش فرض، نصب و پیاده سازی می‌شود. این امکان برای تمام پروژه‌هایی که دات نت فریم ورک 4.5 را هدف قرار می‌دهند وجود دارد. ASP.NET Identity هنگام تولید وب سایت‌های اینترنتی که اکثر کاربرانشان خارجی (External) هستند گزینه خوبی است. اگر سازمان شما از Active Directory و یا Office 365 استفاده می‌کند و می‌خواهید پروژه‌تان قادر باشد تا احراز هویت کارمندان و شرکای تجاری تان را مدیریت کند، **Organizational Accounts** گزینه بهتری است.

برای اطلاعات بیشتر درباره‌ی Individual User Accounts به لینک‌های زیر مراجعه کنید:

asp.net/identity

Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 and OpenID Sign-on

Web API - External Authentication Services

Adding External Logins to your ASP.NET application in Visual Studio 2013

Organizational Accounts

اگر گزینه **Organizational Accounts** را انتخاب کنید پروژه ایجاد شده برای استفاده از Windows Identity Foundation (WIF) پیکربندی خواهد شد. این فریم ورک برای احراز هویت کاربران از Windows Azure Active Directory (WAAD) استفاده می‌کند که شامل Office 365 نیز می‌شود.

Windows Authentication

اگر گزینه‌ی **Windows Authentication** را انتخاب کنید، پروژه ایجاد شده برای استفاده از Windows Authentication IIS Module پیکربندی خواهد شد. چنین اپلیکیشنی نام دامنه و نام کاربری را نمایش خواهد که یا از Active Directory می‌آید، یا از یک ماشین محلی (local machine). اما رابط کاربری ای برای ورود به سیستم وجود ندارد؛ چرا که اینگونه اپلیکیشن‌ها برای سایت‌های اینترانتی (Intranet) استفاده خواهند شد.

یک راه دیگر انتخاب گزینه On-Premises زیر شاخه Organizational Accounts است. این گزینه بجای استفاده از ماژول Windows Authentication از فریم ورک Windows Identity Foundation برای احراز هویت استفاده می‌کند. انجام چند مرحله دستی برای پیکربندی این گزینه لازم است، اما WIF امکاناتی را عرضه می‌کند که در ماژول احراز هویت ویندوز وجود ندارند. برای مثال، هنگام استفاده از WIF می‌توانید تنظیمات لازم را در Active Directory انجام دهید تا قادر به واکشی اطلاعات پوشه‌ها باشید (directory data querying).

گزینه‌های احراز هویت Organizational Accounts

دیالوگ **Configure Authentication** گزینه‌های متعددی برای احراز هویت توسط (Windows Azure Active Directory including Office 365) و Windows Server Active Directory در اختیارتان می‌گذارد:

Cloud - Single Organization

Cloud - Multi Organization

On-Premises

اگر می‌خواهید یکی از گزینه‌های WAAD را امتحان کنید اما حساب کاربری ای ندارید، روی این لینک کلیک کنید تا ثبت نام کنید.

نکته: اگر یکی از گزینه‌های WAAD را انتخاب کنید، باید اطلاعات هویتی (Credentials) یک مدیر کل را وارد کنید. برای نسخه نهایی Visual Studio 2013 برنامه هایی وجود دارد تا دیگر نیازی نباشد چنین مراحلی را تکمیل کنید. در این صورت ویژوال استودیو تنظیماتی را نمایش خواهد داد که یک مدیر می‌تواند بعدا از آنها استفاده کند تا اپلیکیشن را بصورت دستی در WAAD پیکربندی کند.

Cloud - Single Organization Authentication

از این گزینه برای احراز هویت کاربرانی استفاده کنید که در قالب یک OWIN Tenant تعریف می‌شوند. برای مثال سایتی با نام Company.com داریم که برای کارمندان این سازمان از طریق company.onmicrosoft.com قابل دسترسی خواهد بود. نمی‌توانید WAAD را طوری پیکربندی کنید که کاربران tenantهای دیگر نیز به اپلیکیشن شما دسترسی داشته باشند.

Domain

نام دامنه‌ای در WAAD که می‌خواهید اپلیکیشن را برای آن پیکربندی کنید، مثلا company.onmicrosoft.com. اگر از custom domain

ها استفاده می‌کنید مانند company.com بجای company.onmicrosoft.com می‌توانید این اطلاعات را اینجا وارد کنید.

سطح دسترسی

اگر اپلیکیشن نیاز به کوئری گرفتن یا بروز رسانی اطلاعات پوشه‌ها (directory information) را توسط Graph API دارد، از گزینه‌های **Single Sign-On, Read Directory Data** و یا **Single Sign-On, Read and Write Directory Data** استفاده کنید. در غیر اینصورت گزینه **Single Sign-On** را رها کنید. برای اطلاعات بیشتر به Application Access Levels و Using the Graph API to Query Windows Azure AD مراجعه کنید.

Application ID URI

بصورت پیش فرض، قالب پروژه یک شناسه application ID URI برای شما تولید می‌کند، که این کار با الحاق نام پروژه شما به نام دامنه WAAD صورت می‌گیرد. برای مثال، اگر نام پروژه Example باشد و نام دامنه contoso.onmicrosoft.com، شناسه خروجی https://contoso.onmicrosoft.com/Example می‌شود. اگر می‌خواهید بصورت دستی این فیلد را مقدار دهی کنید، گزینه **More Options** را انتخاب کنید. این شناسه باید با https:// شروع شود.

بصورت پیش فرض، اگر اپلیکیشنی که در WAAD تهیه و تدارک دیده شده است، شناسه‌ای یکسان با شناسه موجود در پروژه Visual Studio داشته باشد، پروژه شما به اپلیکیشن موجود در WAAD متصل خواهد شد. اگر می‌خواهید تدارکات جدیدی ببینید تیک گزینه **Overwrite the application entry if one with the same ID already exists** را حذف کنید.

اگر تیک این گزینه حذف شده باشد، و ویژوال استودیو اپلیکیشنی با شناسه‌ای یکسان را پیدا کند، عددی به آخر URI اضافه خواهد شد. مثلا فرض کنید نام پروژه Example است و اپلیکیشنی نیز با شناسه https://contoso.onmicrosoft.com/Example در WAAD وجود دارد. در این صورت اپلیکیشن جدیدی با شناسه ای مانند https://contoso.onmicrosoft.com/Example_ 20130619330903 ایجاد می‌شود.

تهیه و تدارک اپلیکیشن در WAAD

برای آنکه یک اپلیکیشن WAAD ایجاد کنید و یا پروژه را به یک اپلیکیشن موجود متصل کنید، ویژوال استودیو به اطلاعات ورود یک مدیر کل برای دامنه مورد نظر، نیاز دارد. هنگامی که در دیالوگ **Configure Authentication** روی **OK** کلیک می‌کنید، اطلاعات ورود یک مدیر کل از شما درخواست می‌شود و نهایتا هنگامیکه روی **Create Project** کلیک می‌کنید، ویژوال استودیو اپلیکیشن شما را در WAAD پیکربندی می‌کند.

برای اطلاعات بیشتر درباره نحوه استفاده از مدل احراز هویت **Cloud - Single Organization** به لینک‌های زیر مراجعه فرمایید:
Windows Azure Authentication

Adding Sign-On to Your Web Application Using Windows Azure AD

Developing ASP.NET Apps with Windows Azure Active Directory

مقالات مذکور برای ویژوال استودیو 2013 بروز رسانی نشده اند. برخی از مراحلی که در این مقالات بصورت دستی باید انجام شوند در Visual Studio 2013 مکانیزه شده است.

Cloud - Multi Organization Authentication

از این گزینه برای احراز هویت کاربرانی استفاده کنید که در WAADtenantهای متعددی تعریف شده‌اند. برای مثال، نام سایت contoso.com است و برای کارمندان دو سازمان از طریق آدرس‌های contoso.onmicrosoft.com و fabrikam.onmicrosoft.com قابل دسترسی خواهد بود. نحوه پیکربندی این مدل نیز مانند قسمت قبلی است.

برای اطلاعات بیشتر درباره احراز هویت **Cloud - Multi Organization** به لینک‌های زیر مراجعه کنید:

Easy Web App Integration with Windows Azure Active Directory, ASP.NET & Visual Studio

Developing Multi-Tenant Web Applications with Windows Azure AD

On-Premises Organizational Accounts

این گزینه را هنگامی انتخاب کنید که کاربران در Windows Server Active Directory (AD) تعریف شده اند و نمی‌خواهید از WAAD استفاده کنید. از این مدل برای ایجاد وب سایت‌های اینترنت و اینترانت می‌توانید استفاده کنید. برای یک وب سایت اینترنتی از Active Directory Federation Services (ADFS) استفاده کنید.

برای یک وب سایت اینترانتی، می‌توانید کلا این گزینه را رها کنید و از Windows Authentication استفاده کنید. در صورت استفاده از گزینه Windows Authentication لازم نیست تا آدرس سند متادیتا (metadata document URL) را فراهم کنید، همچنین توجه داشته باشید که Windows Authentication امکان کوئری گرفتن از پوشه‌ها و کنترل سطوح دسترسی در Active Directory را ندارد.

On-Premises Authority

آدرس سند متادیتا. این سند اطلاعاتی درباره مختصات Authority دارد که اپلیکیشن از آنها برای به پیش بردن روند احراز هویت و ورود به سایت استفاده می‌کند.

Application ID URI

یک شناسه منحصر به فرد که AD از آن برای شناسایی اپلیکیشن استفاده می‌کند. می‌توانید این فیلد را خالی رها کنید تا ویژوال استودیو بصورت خودکار اپلیکیشنی بدین منظور بسازد.

قدم‌های بعدی

در این مقاله با مدل‌های مختلف احراز هویت در اپلیکیشن‌های Visual Studio 2013 آشنا شدید و برخی تغییرات و امکانات جدید نیز بررسی شدند. برای اطلاعات تکمیلی به ASP.NET and Web Tools for Visual Studio 2013 Release Notes مراجعه کنید.

نظرات خوانندگان

نویسنده: هومن
تاریخ: ۱۳۹۳/۰۳/۱۹ ۱۹:۵۴

سلام
- بابت مطلب خوبتون ممنون.من یه سوال داشتم ، در پورتال‌های سازمانی و در حالتی که نیاز نداریم پالیسی رو از اکتیو بخونیم ، بهتره از حالت آخر یعنی Windows Authentication استفاده کنیم درسته؟
من از این حالت استفاده کردم، و یک مشکل و یک سوال برام پیش اومده...
- مشکلم اینه که در سیستمی که با یوزر اکتیو بالا اومده خوب کار میکنه و یوزر رو تشخیص میده، اما روی لپتاپم چون لوکال هست و طبیعیه اکتیو دایرکتوری موجود نیست نمیتونه یوزر رو تشخیص بده(پنجره لاگین مرورگر را باز میکند) و من نمیتونم برناممو گسترش بدم و برای تست نیاز به اجرای برنامه دارم ، این مشکلو چجوری برطرف کنم؟
- سوالم اینه که ما هیچ کجا اسم دامین یا آی پی سرور رو به برنامه نمیدیم ، پس سیستم احراز هویت چجوری دامین رو تشخیص میده و باهاش کار میکنه ؟

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۳/۲۰ ۹:۱۸

اگر از IE استفاده کنید، مشکلی نباید باشه. چون IE با سیستم اعتبارسنجی مبتنی بر ویندوز یکپارچه هست. اگر با IE صفحه لاگین مرورگر باز میشه، به تنظیمات امنیتی اون مراجعه کنید و سایت رو در قسمت trusted sites اضافه کنید:
http://support.microsoft.com/kb/258063

سمت سرور هم باید در تنظیمات IIS، گزینه‌ی اعتبارسنجی مبتنی بر ویندوز فعال باشه:

http://www.asp.net/mvc/tutorials/older-versions/security/authenticating-users-with-windows-authentication-cs

عنوان: **توسعه اپلیکیشن های ASP.NET با Windows Azure Active Directory**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۵ ۵:۱۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, MVC, Visual Studio 2013, ASP.NET Identity, Windows Azure Active Directory, Security

ابزار ASP.NET برای Windows Azure Active Directory فعال کردن احراز هویت در وب اپلیکیشن هایی که روی Windows Azure Web Sites میزبانی شده اند را ساده‌تر می‌کند. می‌توانید از Windows Azure Authentication برای احراز هویت کاربران Office 365 استفاده کنید، حساب‌های کاربری را از On-Premise Active Directory خود همگام سازی (Sync) کنید و یا از یک دامنه سفارشی Windows Azure Active Directory بهره ببرید. فعال سازی Windows Azure Authentication، اپلیکیشن شما را طوری پیکربندی می‌کند تا تمامی کاربران را با استفاده از یک Windows Azure Active Directory tenant احراز هویت کند.
این مقاله ساختن یک اپلیکیشن ASP.NET را که بر اساس organizational accounts پیکربندی شده و بر روی Windows Azure Active Directory میزبانی می‌شود، مرور می‌کند.
پیش نیاز ها

Visual Studio Express 2013 RC for Web یا Visual Studio 2013 RC

یک حساب کاربری در Windows Azure. می‌توانید یک حساب رایگان بسازید .

یک مدیر کلی به حساب کاربری Active Directory خود اضافه کنید

وارد Windows Azure Portal شوید.
یک حساب کاربری Windows Azure Active Directory (AD) انتخاب یا ایجاد کنید. اگر قبلا حساب کاربری ساخته اید از همان استفاده کنید در غیر اینصورت یک حساب جدید ایجاد کنید. مشترکین Windows Azure یک AD پیش فرض با نام **Default Directory** خواهند داشت.
در حساب کاربری AD خود یک کاربر جدید در نقش Global Administrator بسازید. اکانت AD خود را انتخاب کنید و **Add User** را کلیک کنید. برای اطلاعات کامل‌تر به Managing Windows Azure AD from the Windows Azure Portal 1- Sign Up with an Organizational Account مراجعه کنید.

یک نام کاربری انتخاب کرده و به مرحله بعد بروید.

نام کاربری را وارد کنید و نقش **Global Administrator** را به آن اختصاص دهید. مدیران کلی به یک آدرس ایمیل متناوب هم نیاز دارند. به مرحله بعد بروید.

بر روی **Create** کلیک کنید و کلمه‌ی عبور موقتی را کپی کنید. پس از اولین ورود باید کلمه عبور را تغییر دهید.

یک اپلیکیشن ASP.NET بسازید

در ویژوال استودیو یک پروژه جدید ASP.NET Web Forms یا MVC بسازید و روی **Change Authentication** کلیک کنید.

گزینه **Organizational Accounts** را انتخاب کنید. نام دامنه خود را وارد کنید و سپس گزینه **Single Sign On, Read directory data** را انتخاب کنید. به مرحله بعد بروید.

نکته: در قسمت **More Options** می توانید قلمرو اپلیکیشن (Application ID URI) را تنظیم کنید. تنظیمات پیش فرض برای اکثر

کاربران مناسب است اما در صورت لزوم می‌توانید آنها را ویرایش کنید، مثلا از طریق Windows Azure Portal دامنه‌های سفارشی خودتان را تنظیم کنید.

اگر گزینه **Overwrite** را انتخاب کنید اپلیکیشن جدیدی در Windows Azure برای شما ساخته خواهد شد. در غیر اینصورت فریم ورک سعی می‌کند اپلیکیشنی با شناسه یکسان پیدا کند (در پست متدهای احراز هویت در VS 2013 به تنظیمات این قسمت پرداخته شده).

اطلاعات مدیر کلی دامنه در Active Directory خود را وارد کنید (Credentials) و پروژه را با کلیک کردن بر روی **Create Project** بسازید.

با کلیدهای ترکیبی Ctrl + F5، اپلیکیشن را اجرا کنید. مرورگر شما باید یک اخطار SSL Certificate به شما بدهد. این بدین دلیل است که مدرک استفاده شده توسط IIS Express مورد اعتماد (trusted) نیست. این اخطار را بپذیرید و اجازه اجرا را به آن بدهید. پس از آنکه اپلیکیشن خود را روی Windows Azure منتشر کردید، این پیغام دیگر تولید نمی‌شود؛ چرا که Certificateهای استفاده شده trusted هستند.

با حساب کاربری سازمانی (organizational account) که ایجاد کرده‌اید، وارد شوید.

همانطور که مشاهده می‌کنید هم اکنون به سایت وارد شده اید.

توزیع اپلیکیشن روی Windows Azure

در Windows Azure Portal یک وب سایت را به‌همراه یک دیتابیس، ایجاد کنید. در پانل سمت چپ صفحه روی **Websites** کلیک کنید و بعد **New** را انتخاب کنید. سپس گزینه **Custom Create** را برگزینید.

اپلیکیشن را روی Windows Azure منتشر کنید. روی پروژه کلیک راست کرده و **Publish** را انتخاب کنید. در مرحله تنظیمات (Settings) مشاهده می‌کنید که احراز هویت حساب‌های سازمانی (organizational accounts) فعال است. همچنین سطح دسترسی به خواندن تنظیم شده است. در قسمت Database رشته اتصال دیتابیس را تنظیم کنید.

حال به وب سایت Windows Azure خود بروید و توسط حساب کاربری ایجاد شده وارد سایت شوید. در این مرحله دیگر نباید خطای امنیتی SSL را دریافت کنید.

خواندن اطلاعات پروفایل کاربران توسط Graph API

قالب پروژه ویژوال استودیو برای organizational accounts یک متد و نما بنام UserProfile به پروژه اضافه کرده است.

```
[Authorize]
```

```
public async Task<ActionResult> UserProfile()
{
    string tenantId = ClaimsPrincipal.Current.FindFirst(TenantSchema).Value;

    // Get a token for calling the Windows Azure Active Directory Graph
    AuthenticationContext authContext = new
AuthenticationContext(String.Format(CultureInfo.InvariantCulture, LoginUrl, tenantId));
    ClientCredential credential = new ClientCredential(AppPrincipalId, AppKey);
    AuthenticationResult assertionCredential = authContext.AcquireToken(GraphUrl, credential);
    string authHeader = assertionCredential.CreateAuthorizationHeader();
    string requestUrl = String.Format(
        CultureInfo.InvariantCulture,
        GraphUserUrl,
        HttpUtility.UrlEncode(tenantId),
        HttpUtility.UrlEncode(User.Identity.Name));

    HttpClient client = new HttpClient();
    HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, requestUrl);
    request.Headers.TryAddWithoutValidation("Authorization", authHeader);
    HttpResponseMessage response = await client.SendAsync(request);
    string responseString = await response.Content.ReadAsStringAsync();
    UserProfile profile = JsonConvert.DeserializeObject<UserProfile>(responseString);

    return View(profile);
}
```

کلیک کردن لینک **UserProfile** اطلاعات پروفایل کاربر جاری را نمایش می‌دهد.

اطلاعات بیشتر

Managing Windows Azure AD from the Windows Azure Portal 1– Sign Up with an Organizational Account

Adding Sign-On to Your Web Application Using Windows Azure AD

Using the Graph API to Query Windows Azure AD

نظرات خوانندگان

نویسنده: حسین
تاریخ: ۱۳۹۲/۱۰/۱۵ ۱۲:۲۰

خیلی ممنون از زحمات شما
چند وقت پیش قصد داشتم ثبت نام کنم، زمانی که میخواهید حساب کاربری ایجاد کنید نیاز به شماره موبایل داره تا اجازه بده مراحل ثبت نام کامل بشه ولی اصلا امکان انتخاب کشور ایران وجود ندارد و موفق نشدم

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۶ ۱۳:۱۴

از حساب‌های کاربری مایکروسافت می تونید استفاده کنید.

نویسنده: بهروز
تاریخ: ۱۳۹۲/۱۰/۱۹ ۱۲:۱

ممنون بابت مطلب خوبتون.

لطفا یکم شفاف‌تر بگین که چطور برا تو حساب کاربری ایجاد کنیم؟ من هرکاری میکنم این صفحه میاد!
https://manage.windowsazure.com/Error/NoSubscriptions

نویسنده: کامران سادین
تاریخ: ۱۳۹۲/۱۱/۱۲ ۴:۲۲

سلام. من شماره موبایل هم دادم اما اطلاعات مستر کارت و ویزا کارت میخواد!

عنوان: **افزودن ASP.NET Identity به یک پروژه Web Forms**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۸ ۱۳:۴۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, Entity framework, Security, ASP.Net MVC, ASP.NET Identity, Forms Authentication

با نصب و اجرای Visual Studio 2013 Express for Web یا Visual Studio 2013 شروع کنید.

یک پروژه جدید بسازید (از صفحه شروع یا منوی فایل)

گزینه Visual C# و سپس **ASP.NET Web Application** را انتخاب کنید. نام پروژه را به "WebFormsIdentity" تغییر داده و OK کنید.

در دیالوگ جدید ASP.NET گزینه **Empty** را انتخاب کنید.

دقت کنید که دکمه **Change Authentication** غیرفعال است و هیچ پشتیبانی ای برای احراز هویت در این قالب پروژه وجود ندارد.

افزودن پکیج‌های ASP.NET Identity به پروژه

روی نام پروژه کلیک راست کنید و گزینه **Manage NuGet Packages** را انتخاب کنید. در قسمت جستجوی دیالوگ باز شده عبارت "Identity.E" را وارد کرده و این پکیج را نصب کنید.

دقت کنید که نصب کردن این پکیج وابستگی‌ها را نیز بصورت خودکار نصب می‌کند: Entity Framework و ASP.NET Idenity Core.

افزودن فرم‌های وب لازم برای ثبت نام کاربران

یک فرم وب جدید بسازید.

در دیالوگ باز شده نام فرم را به **Register** تغییر داده و تایید کنید.

فایل ایجاد شده جدید را باز کرده و کد Markup آن را با قطعه کد زیر جایگزین کنید.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Register.aspx.cs"
Inherits="WebFormsIdentity.Register" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
```

```
</head>
<body style="
    <form id="form1" runat="server">
    <div>
        <h4 style="Register a new user</h4>
        <hr />
        <p>
            <asp:Literal runat="server" ID="StatusMessage" />
        </p>
        <div style="margin-bottom:10px">
            <asp:Label runat="server" AssociatedControlID="UserName">User name</asp:Label>
            <div>
                <asp:TextBox runat="server" ID="UserName" />
            </div>
        </div>
        <div style="margin-bottom:10px">
            <asp:Label runat="server" AssociatedControlID="Password">Password</asp:Label>
            <div>
                <asp:TextBox runat="server" ID="Password" TextMode="Password" />
            </div>
        </div>
        <div style="margin-bottom:10px">
            <asp:Label runat="server" AssociatedControlID="ConfirmPassword">Confirm
password</asp:Label>
            <div>
                <asp:TextBox runat="server" ID="ConfirmPassword" TextMode="Password" />
            </div>
        </div>
        <div>
            <div>
                <asp:Button runat="server" OnClick="CreateUser_Click" Text="Register" />
            </div>
        </div>
    </div>
    </form>
</body>
</html>
```

این تنها یک نسخه ساده شده *Register.aspx* است که از چند فیلد فرم و دکمه ای برای ارسال آنها به سرور استفاده می‌کند.

فایل کد این فرم را باز کرده و محتویات آن را با قطعه کد زیر جایگزین کنید.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using System;
using System.Linq;

namespace WebFormsIdentity
{
   public partial class Register : System.Web.UI.Page
   {
      protected void CreateUser_Click(object sender, EventArgs e)
      {
         // Default UserStore constructor uses the default connection string named: DefaultConnection
         var userStore = new UserStore<IdentityUser>();
         var manager = new UserManager<IdentityUser>(userStore);

         var user = new IdentityUser() { UserName = UserName.Text };
         IdentityResult result = manager.Create(user, Password.Text);

         if (result.Succeeded)
         {
            StatusMessage.Text = string.Format("User {0} was created successfully!", user.UserName);
         }
         else
         {
            StatusMessage.Text = result.Errors.FirstOrDefault();
         }
      }
   }
}
```

کد این فرم نیز نسخه ای ساده شده است. فایلی که بصورت خودکار توسط VS برای شما ایجاد می‌شود متفاوت است.

کلاس *IdentityUser* پیاده سازی پیش فرض EntityFramework از قرارداد IUser است. قرارداد IUser تعریفات حداقلی یک کاربر در ASP.NET Identity Core را در بر می‌گیرد.

کلاس *UserStore* پیاده سازی پیش فرض EF از یک فروشگاه کاربر (user store) است. این کلاس چند قرارداد اساسی ASP.NET Identity Core را پیاده سازی می‌کند: *IUserClaimStore*, *IUserLoginStore*, *IUserStore* و *IUserRoleStore*.

کلاس *UserManager* دسترسی به APIهای مربوط به کاربران را فراهم می‌کند. این کلاس تمامی تغییرات را بصورت خودکار در *UserStore* ذخیره می‌کند.

کلاس *IdentityResult* نتیجه یک عملیات هویتی را معرفی می‌کند (identity operations).

پوشه **App_Data** را به پروژه خود اضافه کنید.

فایل *Web.config* پروژه را باز کنید و رشته اتصال جدیدی برای دیتابیس اطلاعات کاربران اضافه کنید. این دیتابیس در زمان اجرا (runtime) بصورت خودکار توسط EF ساخته می‌شود. این رشته اتصال شبیه به رشته اتصالی است که هنگام ایجاد پروژه بصورت

خودکار برای شما تنظیم می‌شود.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=169433
  -->
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit
http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
   <connectionStrings>
      <add name="DefaultConnection" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\WebFormsIdentity.mdf;Initial
Catalog=WebFormsIdentity;Integrated Security=True"
            providerName="System.Data.SqlClient" />
   </connectionStrings>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
EntityFramework">
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
    <providers>
      <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
</configuration>
```

همانطور که مشاهده می‌کنید نام این رشته اتصال *DefaultConnection* است.

روی فایل *Register.aspx* کلیک راست کنید و گزینه **Set As Start Page** را انتخاب کنید. اپلیکیشن خود را با کلیدهای ترکیبی Ctrl + F5 اجرا کنید که تمام پروژه را کامپایل نیز خواهد کرد. یک نام کاربری و کلمه عبور وارد کنید و روی **Register** کلیک کنید.

ASP.NET Identity از اعتبارسنجی نیز پشتیبانی می‌کند، مثلا در این مرحله می‌توانید از اعتبارسنج هایی که توسط ASP.NET Identity Core عرضه می‌شوند برای کنترل رفتار فیلدهای نام کاربری و کلمه عبور استفاده کنید. اعتبارسنج پیش فرض کاربران (User) که UserValidator نام دارد خاصیتی با نام AllowOnlyAlphanumericUserNames دارد که مقدار پیش فرضش هم true است. اعتبارسنج پیش فرض کلمه عبور (MinimumLengthValidator) اطمینان حاصل می‌کند که کلمه عبور حداقل 6 کاراکتر باشد. این اعتبارسنج‌ها بصورت propertyها در کلاس UserManager تعریف شده اند و می‌توانید آنها را overwrite کنید و اعتبارسنجی سفارشی خود را پیاده کنید. از آنجا که الگوی دیتابیس سیستم عضویت توسط Entity Framework مدیریت می‌شود، روی الگوی دیتابیس کنترل کامل دارید، پس از Data Annotations نیز می‌توانید استفاده کنید.

تایید دیتابیس LocalDbIdentity که توسط EF ساخته می‌شود

از منوی **View** گزینه **Server Explorer** را انتخاب کنید.

MvcAuth - Microsoft Visual Studio
FILE
EDIT
VIEW
PROJECT
BUILD
DEBUG
TEAM
TOOLS
Solution Explorer Ctrl+Alt+L
Team Explorer Ctrl+\, Ctrl+M
Server Explorer Ctrl+Alt+S
Architecture Explorer Ctrl+\, Ctrl+R
SQL Server Object Explorer Ctrl+\, Ctrl+S
Bookmark Window Ctrl+K, Ctrl+W
Call Hierarchy Ctrl+Alt+K
Class View Ctrl+Shift+C
Code Definition Window Ctrl+\, D
Document Outline Ctrl+Alt+T
Object Browser Ctrl+Alt+J
Error List Ctrl+\, E
Output Ctrl+Alt+O
Start Page
Task List Ctrl+\, T
Toolbox Ctrl+Alt+X
Notifications Ctrl+W, N
Find Results
Other Windows
Toolbars
Full Screen Shift+Alt+Enter
All Windows Shift+Alt+M
Properties Window F4
Property Pages Shift+F4

گره **DefaultConnection (WebFormsIdentity)** و سپس **Tables** را باز کنید. روی جدول **AspNetUsers** کلیک راست کرده و **Show Table Data** را انتخاب کنید.

پیکربندی اپلیکیشن برای استفاده از احراز هویت OWIN

تا این مرحله ما تنها امکان ایجاد حساب‌های کاربری را فراهم کرده ایم. حال نیاز داریم امکان احراز هویت کاربران برای ورود آنها به سایت را فراهم کنیم. ASP.NET Identity برای احراز هویت مبتنی بر فرم (forms authentication) از OWIN Authentication استفاده می‌کند. OWIN Cookie Authentication مکانیزمی برای احراز هویت کاربران بر اساس cookieها و claimها است (claims-based). این مکانیزم می‌تواند توسط Entity Framework روی OWIN یا IIS استفاده شود.

با چنین مدلی، می‌توانیم از پکیج‌های احراز هویت خود در فریم ورک‌های مختلفی استفاده کنیم، مانند ASP.NET MVC و ASP.NET Web Forms. برای اطلاعات بیشتر درباره پروژه Katana و نحوه اجرای آن بصورت Host Agnostic به لینک Getting Started with the Katana Project مراجعه کنید.

نصب پکیج‌های احراز هویت روی پروژه

روی نام پروژه خود کلیک راست کرده و **Manage NuGet Packages** را انتخاب کنید. در قسمت جستجوی دیالوگ باز شده عبارت "*Identity.Owin*" را وارد کنید و این پکیج را نصب کنید.

به دنبال پکیجی با نام *Microsoft.Owin.Host.SystemWeb* بگردید و آن را نیز نصب کنید.

پکیج **Microsoft.Aspnet.Identity.Owin** حاوی یک سری کلاس Owin Extension است و امکان مدیریت و پیکربندی OWIN Authentication در پکیج‌های ASP.NET Identity Core را فراهم می‌کند.

پکیج **Microsoft.Owin.Host.SystemWeb** حاوی یک سرور OWIN است که اجرای اپلیکیشن‌های مبتنی بر OWIN را روی IIS و با استفاده از ASP.NET Request Pipeline ممکن می‌سازد. برای اطلاعات بیشتر به OWIN Middleware in the IIS integrated pipeline مراجعه کنید.

افزودن کلاس‌های پیکربندی Startup و Authentication

روی پروژه خود کلیک راست کرده و گزینه **Add** و سپس **Add New Item** را انتخاب کنید. در قسمت جستجوی دیالوگ باز شده عبارت "owin" را وارد کنید. نام کلاس را " *Startup* " تعیین کرده و تایید کنید.

فایل Startup.cs را باز کنید و قطعه کد زیر را با محتویات آن جایگزین کنید تا احراز هویت OWIN Cookie Authentication پیکربندی شود.

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;

[assembly: OwinStartup(typeof(WebFormsIdentity.Startup))]

namespace WebFormsIdentity
{
   public class Startup
   {
      public void Configuration(IAppBuilder app)
      {
         // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=316888
         app.UseCookieAuthentication(new CookieAuthenticationOptions
         {
            AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
            LoginPath = new PathString("/Login")
         });
      }
   }
}
```

این کلاس حاوی خاصیت OwinAttribute است که کلاس راه انداز OWIN را نشانه گذاری می‌کند. هر اپلیکیشن OWIN یک کلاس راه انداز (startup) دارد که توسط آن می‌توانید کامپوننت‌های application pipeline را مشخص کنید. برای اطلاعات بیشتر درباره این مدل، به OWIN Startup Class Detection مراجعه فرمایید.

افزودن فرم‌های وب برای ثبت نام و ورود کاربران

فایل *Register.cs* را باز کنید و قطعه کد زیر را وارد کنید. این قسمت پس از ثبت نام موفقیت آمیز کاربر را به سایت وارد می‌کند.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Owin.Security;
using System;
using System.Linq;
using System.Web;

namespace WebFormsIdentity
{
   public partial class Register : System.Web.UI.Page
   {
      protected void CreateUser_Click(object sender, EventArgs e)
      {
         // Default UserStore constructor uses the default connection string named: DefaultConnection
         var userStore = new UserStore<IdentityUser>();
         var manager = new UserManager<IdentityUser>(userStore);
         var user = new IdentityUser() { UserName = UserName.Text };

         IdentityResult result = manager.Create(user, Password.Text);

         if (result.Succeeded)
         {
            var authenticationManager = HttpContext.Current.GetOwinContext().Authentication;
            var userIdentity = manager.CreateIdentity(user,
DefaultAuthenticationTypes.ApplicationCookie);
            authenticationManager.SignIn(new AuthenticationProperties() { }, userIdentity);
            Response.Redirect("~/Login.aspx");
         }
         else
         {
            StatusMessage.Text = result.Errors.FirstOrDefault();
         }
      }
   }
}
```

از آنجا که ASP.NET Identity و OWIN Cookie Authentication هر دو مبتنی بر Claims هستند، فریم ورک از برنامه نویس اپلیکیشن انتظار دارد تا برای کاربر یک آبجکت از نوع ClaimsIdentity تولید کند. این آبجکت تمام اطلاعات اختیارات کاربر را در بر می‌گیرد، مثلا اینکه کاربر به چه نقش هایی تعلق دارد. همچنین در این مرحله می‌توانید اختیارات (Claims) جدیدی به کاربر اضافه کنید.

شما با استفاده از AuthenticationManager که متعلق به OWIN است می‌توانید کاربر را به سایت وارد کنید. برای این کار شما متد SignIn را فراخوانی می‌کنید و آبجکتی از نوع ClaimsIdentity را به آن پاس می‌دهید. این کد کاربر را به سایت وارد می‌کند و یک کوکی برای او می‌سازد. این فراخوانی معادل همان FormAuthentication.SetAuthCookie است که توسط ماژول FormsAuthentication استفاده می‌شود.

روی پروژه خود کلیک راست کرده، فرم وب جدیدی با نام **Login** بسازید.

فایل *Login.aspx* را باز کنید و کد Markup آن را مانند قطعه کد زیر تغییر دهید.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Login.aspx.cs"
Inherits="WebFormsIdentity.Login" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title></title>
</head>
<body style="font-family: Arial, Helvetica, sans-serif; font-size: small">
   <form id="form1" runat="server">
      <div>
```

```
         <h4 style="font-size: medium">Log In</h4>
         <hr />
         <asp:PlaceHolder runat="server" ID="LoginStatus" Visible="false">
            <p>
               <asp:Literal runat="server" ID="StatusText" />
            </p>
         </asp:PlaceHolder>
         <asp:PlaceHolder runat="server" ID="LoginForm" Visible="false">
            <div style="margin-bottom: 10px">
               <asp:Label runat="server" AssociatedControlID="UserName">User name</asp:Label>
               <div>
                  <asp:TextBox runat="server" ID="UserName" />
               </div>
            </div>
            <div style="margin-bottom: 10px">
               <asp:Label runat="server" AssociatedControlID="Password">Password</asp:Label>
               <div>
                  <asp:TextBox runat="server" ID="Password" TextMode="Password" />
               </div>
            </div>
            <div style="margin-bottom: 10px">
               <div>
                  <asp:Button runat="server" OnClick="SignIn" Text="Log in" />
               </div>
            </div>
         </asp:PlaceHolder>
         <asp:PlaceHolder runat="server" ID="LogoutButton" Visible="false">
            <div>
               <div>
                  <asp:Button runat="server" OnClick="SignOut" Text="Log out" />
               </div>
            </div>
         </asp:PlaceHolder>
      </div>
   </form>
</body>
</html>
```

محتوای فایل *Login.aspx.cs* را نیز مانند لیست زیر تغییر دهید.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Owin.Security;
using System;
using System.Web;
using System.Web.UI.WebControls;

namespace WebFormsIdentity
{
   public partial class Login : System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e)
      {
         if (!IsPostBack)
         {
            if (User.Identity.IsAuthenticated)
            {
               StatusText.Text = string.Format("Hello {0}!", User.Identity.GetUserName());
               LoginStatus.Visible = true;
               LogoutButton.Visible = true;
            }
            else
            {
               LoginForm.Visible = true;
            }
         }
      }

      protected void SignIn(object sender, EventArgs e)
      {
         var userStore = new UserStore<IdentityUser>();
         var userManager = new UserManager<IdentityUser>(userStore);
         var user = userManager.Find(UserName.Text, Password.Text);

         if (user != null)
         {
            var authenticationManager = HttpContext.Current.GetOwinContext().Authentication;
            var userIdentity = userManager.CreateIdentity(user,
```

```
DefaultAuthenticationTypes.ApplicationCookie);

            authenticationManager.SignIn(new AuthenticationProperties() { IsPersistent = false },
userIdentity);
            Response.Redirect("~/Login.aspx");
         }
         else
         {
            StatusText.Text = "Invalid username or password.";
            LoginStatus.Visible = true;
         }
      }

      protected void SignOut(object sender, EventArgs e)
      {
         var authenticationManager = HttpContext.Current.GetOwinContext().Authentication;
         authenticationManager.SignOut();
         Response.Redirect("~/Login.aspx");
      }
   }
}
```

متد Page_Load حالا وضعیت کاربر جاری را بررسی می‌کند و بر اساس وضعیت Context.User.Identity.IsAuthenticated تصمیم گیری می‌کند.

نمایش نام کاربر جاری: فریم ورک ASP.NET Identity روی System.Security.Principal.Identity متدهایی نوشته است که به شما امکان دریافت نام و شناسه کاربر جاری را می‌دهد. این متدها در اسمبلی Microsoft.AspNet.Identity.Core وجود دارند. این متدها جایگزین HttpContext.User.Identity.Name هستند.

متد SignIn

این متد، متد CreateUser_Click را که پیشتر بصورت خودکار ایجاد شده جایگزین می‌کند و پس از ایجاد موفقیت آمیز حساب کاربری، کاربر جاری را به سایت وارد می‌کند. فریم ورک OWIN متدهایی روی System.Web.HttpContext افزوده است که به شما این امکان را می‌دهند که یک ارجاع از نوع IOwinContext بگیرید. این متدها در اسمبلی Microsoft.Owin.Host.SystemWeb وجود دارند. کلاس OwinContext خاصیتی از نوع IAuthenticationManager دارد که امکانات احراز هویت موجود برای درخواست جاری را معرفی می‌کند.

پروژه را با Ctrl + F5 اجرا کنید و کاربر جدیدی بسازید. پس از وارد کردن نام کاربری و کلمه عبور و کلیک کردن دکمه Register باید بصورت خودکار به سایت وارد شوید و نام خود را مشاهده کنید.

همانطور که مشاهده می‌کنید در این مرحله حساب کاربری جدید ایجاد شده و به سایت وارد شده اید. روی Log out کلیک کنید تا از سایت خارج شوید. پس از آن باید به صفحه ورود هدایت شوید.
حالا یک نام کاربری یا کلمه عبور نامعتبر وارد کنید و روی **Log in** کلیک کنید.

متد UserManager.Find مقدار null بر می‌گرداند، بنابراین پیام خطای "*Invalid username or password*" نمایش داده خواهد شد.

http://localhost:41969/Login.a
Log In
Invalid username or password.
User name
RaquelS
Password
Log in

نظرات خوانندگان

نویسنده: Programmer
تاریخ: ۱۳۹۲/۱۰/۱۹ ۱۴:۴۱

با عرض سلام و تشکر بابت ترجمه روانتون. خیلی وقت بود که منتظر همچین پستی بودم. اینکه بشه با EF عملیات احراز هویت رو با مکانیسمی قویتر از Memberdship سابق انجام داد. اگر ممکنه همین مثال رو در قالب پروژه MVC انجام بدید. ممنون

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۹ ۱۸:۹

سلام، متشکرم.
ASP.NET Identity بصورت پیش فرض در قالب پروژه‌های VS 2013 استفاده میشه. در پست‌های قبلی بیشتر درباره این فریم بحث شده که می‌تونید مراجعه کنید. برای اطلاعات بیشتر به ASP.NET Identity سر بزنید.

نویسنده: کامران
تاریخ: ۱۳۹۲/۱۲/۲۶ ۱۶:۰

سلام.
من اگر بخوام لیست کاربرایی که یک نقش خاص مثلا "Admin" رو دارن لیست کنم چطور میتونم دسترسی داشته باشم؟

نویسنده: غلامرضا
تاریخ: ۱۳۹۳/۰۵/۳۰ ۳:۴۹

سلام و میبخشید آیا به غیر از روش ایداعی این سایت روش دیگه ای برای استفاده از asp.net identity با EF db first وجود داره؟ اگه هست لطفا راهنمایی کنید.

نویسنده: غلامرضا
تاریخ: ۱۳۹۳/۰۵/۳۱ ۱۶:۳۲

جواب رو پیدا کردم اینم لینکش. فیلم

عنوان: **توزیع یک اپلیکیشن ASP.NET MVC 5 روی Windows Azure**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۹ ۱۰:۵۰
آدرس: www.dotnettips.info
گروه‌ها: Tips, OpenID, ASP.NET MVC 5, ASP.NET Identity, Windows Azure, OAuth, MVC, Security

این مقاله به شما نشان می‌دهد چگونه یک اپلیکیشن وب ASP.NET MVC 5 بسازید که کاربران را قادر می‌سازد با اطلاعات Facebook یا Google احراز هویت شده و به سایت وارد شوند. همچنین این اپلیکیشن را روی Windows Azure توزیع (Deploy) خواهید کرد.
می توانید بصورت رایگان یک حساب کاربری Windows Azure بسازید. اگر هم Visual Studio 2013 را ندارید، بسته SDK بصورت خودکار Visual Studio 2013 for Web را نصب می‌کند. پس از آن می‌توانید به توسعه رایگان اپلیکیشن‌های Azure بپردازید، اگر می‌خواهید از Visual Studio 2012 استفاده کنید به این مقاله مراجعه کنید. این مقاله نسبت به لینک مذکور بسیار ساده‌تر است.
این مقاله فرض را بر این می‌گذارد که شما هیچ تجربه ای در کار با Windows Azure ندارید. در انتهای این مقاله شما یک اپلیکیشن مبتنی بر داده (data-driven) و امن خواهید داشت که در فضای رایانش ابری اجرا می‌شود.
چیزی که شما یاد می‌گیرید:
چطور یک اپلیکیشن وب ASP.NET MVC 5 بسازید و آن را روی یک وب سایت Windows Azure منتشر کنید.
چگونه از OpenID ، OAuth و سیستم عضویت ASP.NET برای ایمن سازی اپلیکیشن خود استفاده کنید.
چگونه از API جدید سیستم عضویت برای مدیریت اعضا و نقش‌ها استفاده کنید.
چگونه از یک دیتابیس SQL برای ذخیره داده‌ها در Windows Azure استفاده کنید.

شما یک اپلیکیشن مدیریت تماس (Contact Manager) ساده خواهید نوشت که بر پایه ASP.NET MVC 5 بوده و از Entity Framework برای دسترسی داده استفاده می‌کند. تصویر زیر صفحه ورود نهایی اپلیکیشن را نشان می‌دهد.

http://localhost:12564/Accou
Log in - Contact Manager
CM Demo
Register Log in
Home About Contact
Log in.
Use a local account to log in.
User name
Password
Remember me?
Log in
Register if you don't have an account.
Use another service to log in.
Facebook
Google
Yahoo
© 2013 - Contact Manager

توجه: برای تمام کردن این مقاله به یک حساب کاربری Windows Azure نیاز دارید، که بصورت رایگان می‌توانید آن را بسازید. برای اطلاعات بیشتر به Windows Azure Free Trial مراجعه کنید.

در این مقاله:

برپایی محیط توسعه (development environment)
برپایی محیط Windows Azure
ایجاد یک اپلیکیشن ASP.NET MVC 5
توزیع اپلیکیشن روی Windows Azure
افزودن یک دیتابیس به اپلیکیشن
افزودن یک OAuth Provider
استفاده از Membership API
توزیع اپلیکیشن روی Windows Azure
قدم‌های بعدی

برپایی محیط توسعه

برای شروع Windows Azure SDK for .NET را نصب کنید. برای اطلاعات بیشتر به Windows Azure SDK for Visual Studio 2013 مراجعه کنید. بسته به اینکه کدام یک از وابستگی‌ها را روی سیستم خود دارید، پروسه نصب می‌تواند از چند دقیقه تا نزدیک دو ساعت طول بکشد. توسط Web Platform می‌توانید تمام نیازمندی‌های خود را نصب کنید.

هنگامی که این مرحله با موفقیت به اتمام رسید، تمام ابزار لازم برای شروع به کار را در اختیار دارید.

برپایی محیط Windows Azure

در قدم بعدی باید یک وب سایت Windows Azure و یک دیتابیس بسازیم.

ایجاد یک وب سایت و دیتابیس در Windows Azure

وب سایت Windows Azure شما در یک محیط اشتراکی (shared) میزبانی می‌شود، و این بدین معنا است که وب سایت‌های شما روی ماشین‌های مجازی (virtual machines) اجرا می‌شوند که با مشتریان دیگر Windows Azure به اشتراک گذاشته شده اند. یک محیط میزبانی اشتراکی گزینه ای کم هزینه برای شروع کار با رایانش‌های ابری است. اگر در آینده ترافیک وب سایت شما رشد چشم گیری داشته باشد، می‌توانید اپلیکیشن خود را طوری توسعه دهید که به نیازهای جدید پاسخگو باشد و آن را روی یک ماشین مجازی اختصاصی (dedicated VMs) میزبانی کنید. اگر معماری پیچیده‌تری نیاز دارید، می‌توانید به یک سرویس Windows Azure Cloud مهاجرت کنید. سرویس‌های ابری روی ماشین‌های مجازی اختصاصی اجرا می‌شوند که شما می‌توانید تنظیمات آنها را بر اساس نیازهای خود پیکربندی کنید.

Windows Azure SQL Database یک سرویس دیتابیس رابطه ای (relational) و مبتنی بر Cloud است که بر اساس تکنولوژی‌های SQL Server ساخته شده. ابزار و اپلیکیشن هایی که با SQL Server کار می‌کنند با SQL Database نیز می‌توانند کار کنند.

در پرتال مدیریتی Windows Azure روی **Web Sites** در قسمت چپ صفحه کلیک کنید، و گزینه **New** را برگزینید.

روی **Web Site** و سپس **Custom Create** کلیک کنید.

در مرحله **Create Web Site** در قسمت **URL** یک رشته وارد کنید که آدرسی منحصر بفرد برای اپلیکیشن شما خواهد بود. آدرس کامل وب سایت شما، ترکیبی از مقدار این فیلد و مقدار روبروی آن است.

در لیست **Database** گزینه **Create a free 20 MB SQL Database** را انتخاب کنید.

در لیست **Region** همان مقداری را انتخاب کنید که برای وب سایت تان انتخاب کرده اید. تنظیمات این قسمت مشخص می‌کند که ماشین مجازی (VM) شما در کدام مرکز داده (data center) خواهد بود.

در قسمت **DB Connection String Name** مقدار پیش فرض *DefaultConnection* را بپذیرید.

دکمه فلش پایین صفحه را کلیک کنید تا به مرحله بعد، یعنی مرحله **Specify Database Settings** بروید.

در قسمت **Name** مقدار *ContactDB* را وارد کنید (تصویر زیر).

در قسمت **Server** گزینه **New SQL Database Server** را انتخاب کنید. اگر قبلا دیتابیس ساخته اید می‌توانید آن را از کنترل dropdown انتخاب کنید.

مقدار قسمت **Region** را به همان مقداری که برای ایجاد وب سایت تان تنظیم کرده اید تغییر دهید.

یک **Login Name** و **Password** مدیر (administrator) وارد کنید. اگر گزینه **New SQL Database server** را انتخاب کرده اید، چنین کاربری وجود ندارد و در واقع اطلاعات یک حساب کاربری جدید را وارد می‌کنید تا بعدا هنگام دسترسی به دیتابیس از آن استفاده کنید. اگر دیتابیس دیگری را از لیست انتخاب کرده باشید، اطلاعات یک حساب کاربری موجود از شما دریافت خواهد شد. در مثال این مقاله ما گزینه **Advanced** را رها می‌کنیم. همچنین در نظر داشته باشید که برای دیتابیس‌های رایگان تنها از یک Collation می‌توانید استفاده کنید.

دکمه تایید پایین صفحه را کلیک کنید تا مراحل تمام شود.

تصویر زیر استفاده از یک SQL Server و حساب کاربری موجود (existing) را نشان می‌دهد.

پرتال مدیریتی پس از اتمام مراحل، به صفحه وب سایت‌ها باز می‌گردد. ستون **Status** نشان می‌دهد که سایت شما در حال ساخته شدن است. پس از مدتی (معمولا کمتر از یک دقیقه) این ستون نشان می‌دهد که سایت شما با موفقیت ایجاد شده. در منوی پیمایش سمت چپ، تعداد سایت هایی که در اکانت خود دارید در کنار آیکون **Web Sites** نمایش داده شده است، تعداد دیتابیس‌ها نیز در کنار آیکون **SQL Databases** نمایش داده می‌شود.

یک اپلیکیشن ASP.NET MVC 5 بسازید

شما یک وب سایت Windows Azure ساختید، اما هنوز هیچ محتوایی در آن وجود ندارد. قدم بعدی ایجاد یک اپلیکیشن وب در ویژوال استودیو و انتشار آن است. ابتدا یک پروژه جدید بسازید.

نوع پروژه را **ASP.NET Web Application** انتخاب کنید.

نکته: در تصویر بالا نام پروژه "MyExample" است اما حتما نام پروژه خود را به "ContactManager" تغییر دهید. قطعه کدهایی که در ادامه مقاله خواهید دید نام پروژه را ContactManager فرض می‌کنند.

در دیالوگ جدید ASP.NET نوع اپلیکیشن را **MVC** انتخاب کنید و دکمه **Change Authentication** را کلیک کنید.

گزینه پیش فرض **Individual User Accounts** را بپذیرید. برای اطلاعات بیشتر درباره متدهای دیگر احراز هویت به این لینک مراجعه کنید. دکمه‌های OK را کلیک کنید تا تمام مراحل تمام شوند.

تنظیم تیتر و پاورقی سایت

فایل *_Layout.cshtml* را باز کنید. دو نمونه از متن "My ASP.NET MVC Application" را با عبارت "Contact Manager" جایگزین کنید.

عبارت "Application name" را هم با "CM Demo" جایگزین کنید.

اولین Action Link را ویرایش کنید و مقدار *Home* را با *Cm* جایگزین کنید تا از *CmController* استفاده کند.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - Contact Manager</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")

</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="colla
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("CM Demo", "Index", "Cm", null, new { @class
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @Html.Partial("_LoginPartial")
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - Contact Manager</p>
        </footer>
    </div>

    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>
```

اپلیکیشن را بصورت محلی اجرا کنید

اپلیکیشن را با Ctrl + F5 اجرا کنید. صفحه اصلی باید در مرورگر پیش فرض باز شود.

اپلیکیشن شما فعلا آماده است و می‌توانید آن را روی Windows Azure توزیع کنید. بعدا دیتابیس و دسترسی داده نیز اضافه خواهد شد.

اپلیکیشن را روی Windows Azure منتشر کنید

در ویژوال استودیو روی نام پروژه کلیک راست کنید و گزینه **Publish** را انتخاب کنید. ویزارد **Publish Web** باز می‌شود.

در قسمت **Profile** روی **Import** کلیک کنید.

حال دیالوگ **Import Publish Profile** نمایش داده می‌شود.

یکی از متدهای زیر را استفاده کنید تا ویژوال استودیو بتواند به اکانت Windows Azure شما متصل شود.
روی **Sign In** کلیک کنید تا با وارد کردن اطلاعات حساب کاربری وارد Windows Azure شوید.

این روش ساده‌تر و سریع‌تر است، اما اگر از آن استفاده کنید دیگر قادر به مشاهده Windows Azure SQL Database یا Mobile Services در پنجره **Server Explorer** نخواهید بود.
روی **Manage subscriptions** کلیک کنید تا یک management certificate نصب کنید، که دسترسی به حساب کاربری شما را ممکن می‌سازد.

در دیالوگ باکس **Manage Windows Azure Subscriptions** به قسمت **Certificates** بروید. سپس **Import** را کلیک کنید. مراحل را دنبال کنید تا یک فایل subscription را بصورت دانلود دریافت کنید (فایل‌های *publishsettings*.) که اطلاعات اکانت Windows Azure شما را دارد.

نکته امنیتی: این فایل تنظیمات را بیرون از پوشه‌های سورس کد خود دانلود کنید، مثلا پوشه Downloads. پس از اتمام عملیات Import هم این فایل را حذف کنید. کاربر مخربی که به این فایل دسترسی پیدا کند قادر خواهد بود تا سرویس‌های Windows Azure شما را کاملا کنترل کند.

برای اطلاعات بیشتر به How to Connect to Windows Azure from Visual Studio مراجعه کنید.

در دیالوگ باکس **Import Publish Profile** وب سایت خود را از لیست انتخاب کنید و OK را کلیک کنید.

در دیالوگ باکس **Publish Web** روی **Publish** کلیک کنید.

اپلیکیشن شما حالا در فضای ابری اجرا می‌شود. دفعه بعد که اپلیکیشن را منتشر کنید تنها فایل‌های تغییر کرده (یا جدید) آپلود خواهند شد.

یک دیتابیس به اپلیکیشن اضافه کنید

در مرحله بعد یک دیتابیس خواهیم ساخت تا اپلیکیشن ما بتواند اطلاعات را نمایش دهد و ویرایش کند. برای ایجاد دیتابیس و دسترسی به داده‌ها از Entity Framework استفاده خواهیم کرد.

کلاس‌های مدل Contacts را اضافه کنید

در پوشه Models پروژه یک کلاس جدید ایجاد کنید.

نام کلاس را به *Contact.cs* تغییر دهید و دکمه Add را کلیک کنید.

کد فایل Contact.cs را با قطعه کد زیر مطابقت دهید.

```
using System.ComponentModel.DataAnnotations;
using System.Globalization;
namespace ContactManager.Models
{
    public class Contact
    {
        public int ContactId { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string Zip { get; set; }
        [DataType(DataType.EmailAddress)]
        public string Email { get; set; }
    }
}
```

این کلاس موجودیت Contact را در دیتابیس معرفی می‌کند. داده هایی که می‌خواهیم برای هر رکورد ذخیره کنیم تعریف شده اند، بعلاوه یک فیلد Primary Key که دیتابیس به آن نیاز دارد.

یک کنترلر و نما برای داده‌ها اضافه کنید

ابتدا پروژه را Build کنید (Ctrl + Shift+ B). این کار را باید پیش از استفاده از مکانیزم Scaffolding انجام دهید.

یک کنترلر جدید به پوشه Controllers اضافه کنید.

در دیالوگ باکس **Add Scaffold** گزینه **MVC 5 Controller with views, using EF** را انتخاب کنید.

در دیالوگ **Add Controller** نام "CmController" را برای کنترلر وارد کنید. (تصویر زیر.)

در لیست **Model** گزینه **Contact (ContactManager.Models)** را انتخاب کنید.

در قسمت **Data context class** گزینه **ApplicationDbContext (ContactManager.Models)** را انتخاب کنید. این **ApplicationDbContext** هم برای اطلاعات سیستم عضویت و هم برای داده‌های Contacts استفاده خواهد شد.

روی Add کلیک کنید. ویژوال استودیو بصورت خودکار با استفاده از Scaffolding متدها و Viewهای لازم برای عملیات CRUD را فراهم می‌کند، که همگی از مدل Contact استفاده می‌کنند.

فعالسازی مهاجرت ها، ایجاد دیتابیس، افزودن داده نمونه و یک راه انداز

مرحله بعدی فعال کردن قابلیت Code First Migrations است تا دیتابیس را بر اساس الگویی که تعریف کرده اید بسازد.

از منوی **Tools** گزینه **Library Package Manager** و سپس **Package Manager Console** را انتخاب کنید.

در پنجره باز شده فرمان زیر را وارد کنید.

```
enable-migrations
```

فرمان **enable-migrations** یک پوشه با نام *Migrations* می سازد و فایلی با نام *Configuration.cs* را به آن اضافه می‌کند. با استفاده از این کلاس می‌توانید داده‌های اولیه دیتابیس را وارد کنید و مهاجرت‌ها را نیز پیکربندی کنید.

در پنجره **Package Manager Console** فرمان زیر را وارد کنید.

```
add-migration Initial
```

فرمان **add-migration initial** فایلی با نام **<data_stamp> initial** ساخته و آن را در پوشه *Migrations* ذخیره می‌کند. در این مرحله دیتابیس شما ایجاد می‌شود. در این فرمان، مقدار **initial** اختیاری است و صرفا برای نامگذاری فایل مهاجرت استفاده شده. فایل‌های جدید را می‌توانید در **Solution Explorer** مشاهده کنید.

در کلاس **Initial** متد **Up** جدول Contacts را می‌سازد. و متد **Down** (هنگامی که می‌خواهید به وضعیت قبلی بازگردید) آن را drop می‌کند.

حال فایل *Migrations/Configuration.cs* را باز کنید. فضای نام زیر را اضافه کنید.

```
using ContactManager.Models;
```

حال متد *Seed* را با قطعه کد زیر جایگزین کنید.

```
protected override void Seed(ContactManager.Models.ApplicationDbContext context)
{
    context.Contacts.AddOrUpdate(p => p.Name,
       new Contact
       {
           Name = "Debra Garcia",
           Address = "1234 Main St",
           City = "Redmond",
           State = "WA",
           Zip = "10999",
           Email = "debra@example.com",
       },
        new Contact
        {
            Name = "Thorsten Weinrich",
            Address = "5678 1st Ave W",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "thorsten@example.com",
        },
        new Contact
        {
            Name = "Yuhong Li",
            Address = "9012 State st",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "yuhong@example.com",
        },
        new Contact
        {
            Name = "Jon Orton",
            Address = "3456 Maple St",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "jon@example.com",
        },
        new Contact
        {
            Name = "Diliana Alexieva-Bosseva",
            Address = "7890 2nd Ave E",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "diliana@example.com",
        }
        );
}
```

این متد دیتابیس را Seed می‌کند، یعنی داده‌های پیش فرض و اولیه دیتابیس را تعریف می‌کند. برای اطلاعات بیشتر به Seeding and Debugging Entity Framework (EF) DBs مراجعه کنید.

در پنجره **Package Manager Console** فرمان زیر را وارد کنید.

```
update-database
```

```
Package Manager Console
Package source: NuGet official package source
PM> enable-migrations
Checking if the context targets an existing database...
Code First Migrations enabled for project ContactManager.
PM> add-migration Initial
Scaffolding migration 'Initial'.
The Designer Code for this migration file includes a snapshot of your current Code Fi
 model. This snapshot is used to calculate the changes to your model when you scaffol
the next migration. If you make additional changes to your model that you want to
include in this migration, then you can re-scaffold it by running 'Add-Migration
201209182148203_Initial' again.
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target
database.
Applying code-based migrations: [201209182148203_Initial].
Applying code-based migration: 201209182148203_Initial.
Running Seed method.
PM>
100 %
```

فرمان **update-database** مهاجرت نخست را اجرا می‌کند، که دیتابیس را می‌سازد. بصورت پیش فرض این یک دیتابیس SQL Server Express LocalDB است.

حال پروژه را با CTRL + F5 اجرا کنید.

همانطور که مشاهده می‌کنید، اپلیکیشن داده‌های اولیه (Seed) را نمایش می‌دهد، و لینک هایی هم برای ویرایش، حذف و مشاهده جزئیات رکوردها فراهم می‌کند. می‌توانید داده‌ها را مشاهده کنید، رکورد جدید ثبت کنید و یا داده‌های قبلی را ویرایش و حذف کنید.

یک تامین کننده OAuth2 و OpenID اضافه کنید OAuth یک پروتکل باز است که امکان authorization امن توسط یک متد استاندارد را فراهم می‌کند. این پروتکل می‌تواند در اپلیکیشن‌های وب، موبایل و دسکتاپ استفاده شود. قالب پروژه ASP.NET MVC internet از OAuth و OpenID استفاده می‌کند تا فیسبوک، توییتر، گوگل و حساب‌های کاربری مایکروسافت را بعنوان تامین کنندگان خارجی تعریف کند. به سادگی می‌توانید قطعه کدی را ویرایش کنید و از تامین کننده احراز هویت مورد نظرتان استفاده کنید. مراحلی که برای اضافه کردن این تامین کنندگان باید دنبال کنید، بسیار مشابه همین مراحلی است که در این مقاله دنبال خواهید کرد. برای اطلاعات بیشتر درباره نحوه استفاده از فیسبوک بعنوان یک تامین کننده احراز هویت به Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 and OpenID Sign-on مراجعه کنید.

علاوه بر احراز هویت، اپلیکیشن ما از نقش‌ها (roles) نیز استفاده خواهد کرد تا از authorization پشتیبانی کند. تنها کاربرانی که به نقش *canEdit* تعلق داشته باشند قادر به ویرایش اطلاعات خواهند بود (یعنی ایجاد، ویرایش و حذف رکورد ها).

فایل *App_Start/Startup.Auth.cs* را باز کنید. توضیحات متد *app.UseGoogleAuthentication* را حذف کنید.

حال اپلیکیشن را اجرا کنید و روی لینک **Log In** کلیک کنید.

زیر قسمت **User another service to log in** روی دکمه **Google** کلیک کنید. اطلاعات کاربری خود را وارد کنید. سپس **Accept** را کلیک کنید تا به اپلیکیشن خود دسترسی کافی بدهید (برای آدرس ایمیل و اطلاعات پایه).

حال باید به صفحه ثبت نام (Register) هدایت شوید. در این مرحله می‌توانید در صورت لزوم نام کاربری خود را تغییر دهید. نهایتا روی **Register** کلیک کنید.

Register - Contact Manag
localhost:62951/Account/ExternalLoginCall
CM Demo
Register.
Associate your Google account.
Association Form
You've successfully authenticated with Google. Please enter a user name for this site below and click the Register button to finish logging in.
User name
RickAnderson
Register
© 2013 - Contact Manager

استفاده از Membership API

در این قسمت شما یک کاربر محلی و نقش *canEdit* را به دیتابیس عضویت اضافه می‌کنید. تنها کاربرانی که به این نقش تعلق دارند قادر به ویرایش داده‌ها خواهند بود. یکی از بهترین تمرین‌ها (best practice) نام گذاری نقش‌ها بر اساس عملیاتی است که می‌توانند اجرا کنند. بنابراین مثلا *canEdit* نسبت به نقشی با نام *admin* ترجیح داده می‌شود. هنگامی که اپلیکیشن شما رشد می‌کند و بزرگتر می‌شود، شما می‌توانید نقش‌های جدیدی مانند *canDeleteMembers* اضافه کنید، بجای آنکه از نام‌های گنگی مانند *superAdmin* استفاده کنید.

فایل *Migrations/Configuration.cs* را باز کنید و عبارات زیر را به آن اضافه کنید.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
```

متد **AddUserAndRole** را به این کلاس اضافه کنید.

```
bool AddUserAndRole(ContactManager.Models.ApplicationDbContext context)
 {
    IdentityResult ir;
    var rm = new RoleManager<IdentityRole>
        (new RoleStore<IdentityRole>(context));
    ir = rm.Create(new IdentityRole("canEdit"));
    var um = new UserManager<ApplicationUser>(
        new UserStore<ApplicationUser>(context));
    var user = new ApplicationUser()
    {
       UserName = "user1",
    };
    ir = um.Create(user, "Passw0rd1");
    if (ir.Succeeded == false)
       return ir.Succeeded;
    ir = um.AddToRole(user.Id, "canEdit");
    return ir.Succeeded;
 }
```

حالا از متد **Seed** این متد جدید را فراخوانی کنید.

```
protected override void Seed(ContactManager.Models.ApplicationDbContext context)
{
    AddUserAndRole(context);
    context.Contacts.AddOrUpdate(p => p.Name,
        // Code removed for brevity
}
```

این کدها نقش جدیدی با نام *canEdit* و کاربری با نام *user1* می سازد. سپس این کاربر به نقش مذکور اضافه می‌شود.

کدی موقتی برای تخصیص نقش *canEdit* به کاربران جدید Social Provider ها

در این قسمت شما متد **ExternalLoginConfirmation** در کنترلر Account را ویرایش خواهید کرد. یا این تغییرات، کاربران جدیدی که توسط OAuth یا OpenID ثبت نام می‌کنند به نقش *canEdit* اضافه می‌شوند. تا زمانی که ابزاری برای افزودن و مدیریت نقش‌ها بسازیم، از این کد موقتی استفاده خواهیم کرد. تیم مایکروسافت امیدوار است ابزاری مانند WSAT برای مدیریت کاربران و نقش‌ها در آینده عرضه کند. بعدا در این مقاله با اضافه کردن کاربران به نقش‌ها بصورت دستی از طریق **Server Explorer** نیز آشنا خواهید شد.

فایل **Controllers/AccountController.cs** را باز کنید و متد **ExternalLoginConfirmation** را پیدا کنید.

درست قبل از فراخوانی **SignInAsync** متد **AddToRoleAsync** را فراخوانی کنید.

```
await UserManager.AddToRoleAsync(user.Id, "CanEdit");
```

کد بالا کاربر ایجاد شده جدید را به نقش canEdit اضافه می‌کند، که به آنها دسترسی به متدهای ویرایش داده را می‌دهد. تصویری

از تغییرات کد در زیر آمده است.

```
//
// POST: /Account/ExternalLoginConfirmation
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ExternalLoginConfirmation(ExternalLoginConfirmati
{
    if (User.Identity.IsAuthenticated)
    {
        return RedirectToAction("Manage");
    }

    if (ModelState.IsValid)
    {
        // Get the information about the user from the external login provider
        var info = await AuthenticationManager.GetExternalLoginInfoAsync();
        if (info == null)
        {
            return View("ExternalLoginFailure");
        }
        var user = new ApplicationUser() { UserName = model.UserName };
        var result = await UserManager.CreateAsync(user);
        if (result.Succeeded)
        {
            result = await UserManager.AddLoginAsync(user.Id, info.Login);
            if (result.Succeeded)
            {
                await UserManager.AddToRoleAsync(user.Id, "CanEdit");
                await SignInAsync(user, isPersistent: false);
                return RedirectToLocal(returnUrl);
            }
        }
        AddErrors(result);
    }

    ViewBag.ReturnUrl = returnUrl;
    return View(model);
}
```

در ادامه مقاله اپلیکیشن خود را روی Windows Azure منتشر خواهید کرد و با استفاده از Google و تامین کنندگان دیگر وارد سایت می‌شوید. هر فردی که به آدرس سایت شما دسترسی داشته باشد، و یک حساب کاربری Google هم در اختیار داشته باشد

می‌تواند در سایت شما ثبت نام کند و سپس دیتابیس را ویرایش کند. برای جلوگیری از دسترسی دیگران، می‌توانید وب سایت خود را متوقف (stop) کنید.

در پنجره **Package Manager Console** فرمان زیر را وارد کنید.

```
Update-Database
```

فرمان را اجرا کنید تا متد **Seed** را فراخوانی کند. حال **AddUserAndRole** شما نیز اجرا می‌شود. تا این مرحله نقش *canEdit* ساخته شده و کاربر جدیدی با نام *user1* ایجاد و به آن افزوده شده است.

محافظت از اپلیکیشن توسط SSL و خاصیت Authorize

در این قسمت شما با استفاده از خاصیت Authorize دسترسی به اکشن متدها را محدود می‌کنید. کاربران ناشناس (Anonymous) تنها قادر به مشاهده متد **Index** در کنترلر **home** خواهند بود. کاربرانی که ثبت نام کرده اند به متدهای **Index** و **Details** در کنترلر **Cm** و صفحات About و Contact نیز دسترسی خواهند داشت. همچنین دسترسی به متدهایی که داده‌ها را تغییر می‌دهند تنها برای کاربرانی وجود دارد که در نقش *canEdit* هستند.

خاصیت Authorize و RequireHttps را به اپلیکیشن اضافه کنید. یک راه دیگر افزودن این خاصیت‌ها به تمام کنترلرها است، اما تجارب امنیتی توصیه می‌کند که این خاصیت‌ها روی کل اپلیکیشن اعمال شوند. با افزودن این خاصیت‌ها بصورت global تمام کنترلرها و اکشن متدهایی که می‌سازید بصورت خودکار محافظت خواهند شد، و دیگر لازم نیست بیاد داشته باشید کدام کنترلرها و متدها را باید ایمن کنید.

برای اطلاعات بیشتر به Securing your ASP.NET MVC App and the new AllowAnonymous Attribute مراجعه کنید.

فایل *App_Start/FilterConfig.cs* را باز کنید و متد *RegisterGlobalFilters* را با کد زیر مطابقت دهید.

```
public static void
RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new HandleErrorAttribute());
    filters.Add(new System.Web.Mvc.AuthorizeAttribute());
    filters.Add(new RequireHttpsAttribute());
}
```

خاصیت Authorize در کد بالا از دسترسی کاربران ناشناس به تمام متدهای اپلیکیشن جلوگیری می‌کند. شما برای اعطای دسترسی به متدهایی خاص از خاصیت AllowAnonymous استفاده خواهید کرد. در آخر خاصیت RequireHTTPS باعث می‌شود تا تمام دسترسی‌ها به اپلیکیشن وب شما از طریق HTTPS صورت گیرد.

حالا خاصیت AllowAnonymous را به متد **Index** در کنترلر Home اضافه کنید. از این خاصیت برای اعطای دسترسی به تمامی کاربران سایت استفاده کنید. قسمتی از کد کنترلر Home را در زیر می‌بینید.

```
namespace ContactManager.Controllers
 {
    public class HomeController : Controller
    {
       [AllowAnonymous]
       public ActionResult Index()
       {
          return View();
       }
```

یک جستجوی عمومی برای عبارت *AllowAnonymous* انجام دهید. همانطور که مشاهده می‌کنید این خاصیت توسط متدهای ورود و ثبت نام در کنترلر Account نیز استفاده شده است.

در کنترلر *CmController* خاصیت *[Authorize(Roles="canEdit")]* را به تمام متدهایی که با داده سر و کار دارند اضافه کنید، به غیر از متدهای Index و Details. قسمتی از کد کامل شده در زیر آمده است.

```
public class CmController : Controller
{
    private ContactManagerContext db = new ContactManagerContext();

    // GET: /Cm/Create

    [Authorize(Roles = "canEdit")]
    public ActionResult Create()
    {
        return View();
    }

    // POST: /Cm/Create
    [HttpPost]
    [ValidateAntiForgeryToken]
    [Authorize(Roles = "canEdit")]
    public ActionResult Create([Bind(Include = "ContactId,Name,Address,City,
    {
        if (ModelState.IsValid)
        {
            db.Contacts.Add(contact);
            db.SaveChanges();
            return RedirectToAction("Index");
        }

        return View(contact);
    }

    // GET: /Cm/Edit/5
    [Authorize(Roles = "canEdit")]
    public ActionResult Edit(int? id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        Contact contact = db.Contacts.Find(id);
        if (contact == null)
        {
            return HttpNotFound();
        }
        return View(contact);
    }
```

فعال سازی SSL برای پروژه

در Solution Explorer پروژه خود را انتخاب کنید. سپس کلید F4 را فشار دهید تا دیالوگ خواص (Properties) باز شود. حال مقدار خاصیت **SSL Enabled** را به true تنظیم کنید. آدرس **SSL URL** را کپی کنید. این آدرس چیزی شبیه به https://localhost:44300/ خواهد بود.

روی نام پروژه کلیک راست کنید و **Properties** را انتخاب کنید. در قسمت چپ گزینه **Web** را انتخاب کنید. حالا مقدار **Project Url** را به آدرسی که کپی کرده اید تغییر دهید. نهایتا تغییرات را ذخیره کنید و پنجره را ببندید.

حال پروژه را اجرا کنید. مرورگر شما باید یک پیام خطای اعتبارسنجی به شما بدهد. دلیلش این است که اپلیکیشن شما از یک

Valid Certificate استفاده نمی‌کند. هنگامی که پروژه را روی Windows Azure منتشر کنید دیگر این پیغام را نخواهید دید. چرا که سرورهای مایکروسافت همگی لایسنس‌های معتبری دارند. برای اپلیکیشن ما می‌توانید روی **Continue to this website** را انتخاب کنید.

حال مرورگر پیش فرض شما باید صفحه **Index** از کنترلر home را به شما نمایش دهد.

اگر از یک نشست قبلی هنوز در سایت هستید (logged-in) روی لینک **Log out** کلیک کنید و از سایت خارج شوید.

روی لینک‌های **About** و **Contact** کلیک کنید. باید به صفحه ورود به سایت هدایت شوید چرا که کاربران ناشناس اجازه دسترسی به این صفحات را ندارند.

روی لینک **Register** کلیک کنید و یک کاربر محلی با نام *Joe* بسازید. حال مطمئن شوید که این کاربر به صفحات Home, About و Contact دسترسی دارد.

روی لینک *CM Demo* کلیک کنید و مطمئن شوید که داده‌ها را مشاهده می‌کنید.

حال روی یکی از لینک‌های ویرایش (Edit) کلیک کنید. این درخواست باید شما را به صفحه ورود به سایت هدایت کند، چرا که کاربران محلی جدید به نقش *canEdit* تعلق ندارند.

با کاربر *user1* که قبلا ساختید وارد سایت شوید. حال به صفحه ویرایشی که قبلا درخواست کرده بودید هدایت می‌شوید.

اگر نتوانستید با این کاربر به سایت وارد شوید، کلمه عبور را از سورس کد کپی کنید و مجددا امتحان کنید. اگر همچنان نتوانستید به سایت وارد شوید، جدول **AspNetUsers** را بررسی کنید تا مطمئن شوید کاربر *user1* ساخته شده است. این مراحل را در ادامه مقاله خواهید دید.

در آخر اطمینان حاصل کنید که می‌توانید داده‌ها را تغییر دهید.

اپلیکیشن را روی Windows Azure منتشر کنید

ابتدا پروژه را Build کنید. سپس روی نام پروژه کلیک راست کرده و گزینه **Publish** را انتخاب کنید.

در دیالوگ باز شده روی قسمت **Settings** کلیک کنید. روی **File Publish Options** کلیک کنید تا بتوانید **Remote connection string** را برای **ApplicationDbContext** و دیتابیس ContactDB انتخاب کنید.

اگر ویژوال استودیو را پس از ساخت Publish profile بسته و دوباره باز کرده اید، ممکن است رشته اتصال را در لیست موجود نبینید. در چنین صورتی، بجای ویرایش پروفایل انتشار، یک پروفایل جدید بسازید. درست مانند مراحلی که پیشتر دنبال کردید.

Publish Web
cm22 *
Configuration:
Release
File Publish Options
Databases
ApplicationDbContext (DefaultConnection)
Remote connection string
...
ContactDB
Use this connection string at runtime (update destination web.config)
Execute Code First Migrations (runs on application start)
< Prev
Next >
Publish
Close

زیر قسمت **ContactManagerContext** گزینه **Execute Code First Migrations** را انتخاب کنید.

حال **Publish** را کلیک کنید تا اپلیکیشن شما منتشر شود. با کاربر *user1* وارد سایت شوید و بررسی کنید که می‌توانید داده‌ها را ویرایش کنید یا خیر.

حال از سایت خارج شوید و توسط یک اکانت Google یا Facebook وارد سایت شوید، که در این صورت نقش canEdit نیز به شما تعلق می‌گیرد.

برای جلوگیری از دسترسی دیگران، وب سایت را متوقف کنید

در **Server Explorer** به قسمت **Web Sites** بروید. حال روی هر نمونه از وب سایت‌ها کلیک راست کنید و گزینه **Stop Web Site** را انتخاب کنید.

Server Explorer
Data Connections
ContactManagerContext (ContactManager)
DefaultConnection (ContactManager)
Servers
SharePoint Connections
Windows Azure
Cloud Services
Mobile Services
Service Bus
SQL Databases
ContactDB
Storage
Virtual Machines
Web Sites
cm1234.azurewebsites.net
cm1234.azurewebsites.net
Refresh
View Settings
Stop Web Site
Open in Browser
Attach Debugger
Open in Management Portal
View Streaming Logs in Output Window
Download Publish Profile
Properties
Alt+Enter

یک راه دیگر متوقف کردن وب سایت از طریق پرتال مدیریت Windows Azure است.

فراخوانی AddToRoleAsync را حذف و اپلیکیشن را منتشر و تست کنید

کنترلر Account را باز کنید و کد زیر را از متد **ExternalLoginConfirmation** حذف کنید.

```
await UserManager.AddToRoleAsync(user.Id, "CanEdit");
```

پروژه را ذخیره و Build کنید. حال روی نام پروژه کلیک راست کرده و **Publish** را انتخاب کنید.

دکمه **Start Preview** را فشار دهید. در این مرحله تنها فایل هایی که نیاز به بروز رسانی دارند آپلود خواهند شد.

وب سایت را راه اندازی کنید. ساده‌ترین راه از طریق پرتال مدیریت Windows Azure است. توجه داشته باشید که تا هنگامی که وب سایت شما متوقف شده، نمی‌توانید اپلیکیشن خود را منتشر کنید.

حال به ویژوال استودیو بازگردید و اپلیکیشن را منتشر کنید. اپلیکیشن Windows Azure شما باید در مرورگر پیش فرض تان باز شود. حال شما در حال مشاهده صفحه اصلی سایت بعنوان یک کاربر ناشناس هستید.

روی لینک **About** کلیک کنید، که شما را به صفحه ورود هدایت می‌کند.

روی لینک **Register** در صفحه ورود کلیک کنید و یک حساب کاربری محلی بسازید. از این حساب کاربری برای این استفاده می‌کنیم که ببینیم شما به صفحات فقط خواندنی (read-only) و نه صفحاتی که داده‌ها را تغییر می‌دهند دسترسی دارید یا خیر. بعدا در ادامه مقاله، دسترسی حساب‌های کاربری محلی (local) را حذف می‌کنیم.

مطمئن شوید که به صفحات About و Contact دسترسی دارید.

لینک **CM Demo** را کلیک کنید تا به کنترلر *CmController* هدایت شوید.

روی یکی از لینک‌های Edit کلیک کنید. این کار شما را به صفحه ورود به سایت هدایت می‌کند. در زیر قسمت **User another service to log in** یکی از گزینه‌های Google یا Facebook را انتخاب کنید و توسط حساب کاربری ای که قبلا ساختید وارد شوید.

حال بررسی کنید که امکان ویرایش اطلاعات را دارید یا خیر.

نکته: شما نمی‌توانید در این اپلیکیشن از اکانت گوگل خود خارج شده، و با همان مرورگر با اکانت گوگل دیگری وارد اپلیکیشن شوید. اگر دارید از یک مرورگر استفاده می‌کنید، باید به سایت گوگل رفته و از آنجا خارج شوید. برای وارد شدن به اپلیکیشن توسط یک اکانت دیگر می‌توانید از یک مرورگر دیگر استفاده کنید.

دیتابیس SQL Azure را بررسی کنید

در **Server Explorer** دیتابیس **ContactDB** را پیدا کنید. روی آن کلیک راست کرده و **Open in SQL Server Object Explorer** را

انتخاب کنید.

توجه: اگر نمی‌توانید گره **SQL Databases** را باز کنید و یا **ContactDB** را در ویژوال استودیو نمی‌بینید، باید مراحلی را طی کنید تا یک پورت یا یکسری پورت را به فایروال خود اضافه کنید. دقت داشته باشید که در صورت اضافه کردن Port Rangeها ممکن است چند دقیقه زمان نیاز باشد تا بتوانید به دیتابیس دسترسی پیدا کنید.

روی جدول **AspNetUsers** کلیک راست کرده و **View Data** را انتخاب کنید.

SQL Server Object Explorer
SQL Server
(localdb)\Projects (SQL Server 11.0.3000.0 -
ox6jx6raw7.database.windows.net (SQL Serv
Databases
ContactDB
Tables
System Tables
dbo.__MigrationHistory
dbo.AspNetRoles
dbo.AspNetUserClaims
dbo.AspNetUserLogins
dbo.AspNetUserRoles
dbo.AspNetUsers
Data Comparison...
Script As
View Code
View Designer
View Permissions
View Data
Delete
Del
Rename
Refresh
Properties

dbo.AspNetUsers [Data]

Max Rows: 1000

Id	UserName	PasswordHash	Secu...	Discriminator
3b7fd83f-fc68-4f4d-ae27-b9922d17602d	JoeLocalUser	AGgn9Gfmxw...	c3337...	ApplicationUser
8a5687c2-86ed-40c9-853b-26ef40b7a2bb	RickGM000	NULL	5489a...	ApplicationUser
94715c84-9919-4146-ad2b-0cf692c7c70d	JoeLocalUser2	AOIMz9t+/+z...	d9f47...	ApplicationUser
9af370af-8055-4cef-965f-990214c24ccf	user1	AJjTVn2u86D...	7f3ab...	ApplicationUser
a18b44da-a9ac-4153-89a0-d9c808f22df8	joeLocal	ACDwfL01Klf...	2977f...	ApplicationUser
NULL	NULL	NULL	NULL	NULL

حالا روی **AspNetUserRoles** کلیک راست کنید و **View Data** را انتخاب کنید.

dbo.AspNetUserRoles [Data] | dbo.AspNetUsers [Data]

Max Rows: 1000

UserId	RoleId
8a5687c2-86ed-40c9-853b-26ef40b7a2bb	e43a4145-7089-4292-9057-af56e5d8e940
9af370af-8055-4cef-965f-990214c24ccf	e43a4145-7089-4292-9057-af56e5d8e940
NULL	NULL

اگر شناسه کاربران (User ID) را بررسی کنید، مشاهده می‌کنید که تنها دو کاربر *user1* و اکانت گوگل شما به نقش *canEdit* تعلق دارند.

Cannot open server login error

اگر خطایی مبنی بر "Cannot open server" دریافت می‌کنید، مراحل زیر را دنبال کنید.

شما باید آدرس IP خود را به لیست آدرس‌های مجاز (Allowed IPs) اضافه کنید. در پرتال مدیریتی Windows Azure در قسمت چپ صفحه، گزینه **SQL Databases** را انتخاب کنید.

دیتابیس مورد نظر را انتخاب کنید. حالا روی لینک **Set up Windows Azure firewall rules for this IP address** کلیک کنید.

هنگامی که با پیغام "The current IP address xxx.xxx.xxx.xxx is not included in existing firewall rules. Do you want? to update the firewall rules" مواجه شدید **Yes** را کلیک کنید. افزودن یک آدرس IP بدین روش معمولا کافی نیست و در فایروال‌های سازمانی و بزرگ باید Range بیشتری را تعریف کنید.

مرحله بعد اضافه کردن محدوده آدرس‌های مجاز است.

مجددا در پرتال مدیریتی Windows Azure روی **SQL Databases** کلیک کنید. سروری که دیتابیس شما را میزبانی می‌کند انتخاب کنید.

در بالای صفحه لینک **Configure** را کلیک کنید. حالا نام rule جدید، آدرس شروع و پایان را وارد کنید.

در پایین صفحه **Save** را کلیک کنید.

در آخر می‌توانید توسط SSOX به دیتابیس خود متصل شوید. از منوی **View** گزینه **SQL Server Object Explorer** را انتخاب کنید. روی **SQL Server** کلیک راست کرده و **Add SQL Server** را انتخاب کنید.

در دیالوگ **Connect to Server** متد احراز هویت را به **SQL Server Authentication** تغییر دهید. این کار نام سرور و اطلاعات ورود پرتال Windows Azure را به شما می‌دهد.

در مرورگر خود به پرتال مدیریتی بروید و **SQL Databases** را انتخاب کنید. دیتابیس **ContactDB** را انتخاب کرده و روی **View SQL Database connection strings** کلیک کنید. در صفحه **Connection Strings** مقادیر **Server** و **User ID** را کپی کنید. حالا مقادیر را در دیالوگ مذکور در ویژوال استودیو بچسبانید. مقدار فیلد **User ID** در قسمت **Login** وارد می‌شود. در آخر هم کلمه عبوری که هنگام ساختن دیتابیس تنظیم کردید را وارد کنید.

حالا می‌توانید با مراحلی که پیشتر توضیح داده شد به دیتابیس **Contact DB** مراجعه کنید.

افزودن کاربران به نقش canEdit با ویرایش جداول دیتابیس

پیشتر در این مقاله، برای اضافه کردن کاربران به نقش canEdit از یک قطعه کد استفاده کردیم. یک راه دیگر تغییر جداول دیتابیس بصورت مستقیم است. مراحلی که در زیر آمده اند اضافه کردن کاربران به یک نقش را نشان می‌دهند.

در **SQL Server Object Explorer** روی جدول **AspNetUserRoles** کلیک راست کنید و **View Data** را انتخاب کنید.

dbo.AspNetUserRoles [Data] | dbo.AspNetUsers [Data]

Max Rows: 1000

UserId	RoleId
8a5687c2-86ed-40c9-853b-26ef40b7a2bb	e43a4145-7089-4292-9057-af56e5d8e940
9af370af-8055-4cef-965f-990214c24ccf	e43a4145-7089-4292-9057-af56e5d8e940
NULL	NULL

حالا *RoleId* را کپی کنید و در ردیف جدید بچسبانید.

dbo.AspNetUserRoles [Data] | dbo.AspNetUsers [Data]

Max Rows: 1000

UserId	RoleId
8a5687c2-86ed-40c9-853b-26ef40b7a2bb	e43a4145-7089-4292-9057-af56e5d8e940
9af370af-8055-4cef-965f-990214c24ccf	e43a4145-7089-4292-9057-af56e5d8e940
	e43a4145-7089-4292-9057-af56e5d8e940
NULL	NULL

شناسه کاربر مورد نظر را از جدول **AspNetUsers** پیدا کنید و مقدار آن را در ردیف جدید کپی کنید. همین! کاربر جدید شما به نقش canEdit اضافه شد.

نکاتی درباره ثبت نام محلی (Local Registration)

ثبت نام فعلی ما از بازنشانی کلمه‌های عبور (password reset) پشتیبانی نمی‌کند. همچنین اطمینان حاصل نمی‌شود که کاربران سایت انسان هستند (مثلا با استفاده از یک CAPTCHA). پس از آنکه کاربران توسط تامین کنندگان خارجی (مانند گوگل) احراز هویت شدند، می‌توانند در سایت ثبت نام کنند. اگر می‌خواهید ثبت نام محلی را برای اپلیکیشن خود غیرفعال کنید این مراحل را دنبال کنید:

در کنترلر Account متدهای *Register* را ویرایش کنید و خاصیت **AllowAnonymous** را از آنها حذف کنید (هر دو متد GET و POST).

این کار ثبت نام کاربران ناشناس و بدافزارها (bots) را غیر ممکن می‌کند.

در پوشه *Views/Shared* فایل *_LoginPartial.cshtml* را باز کنید و لینک Register را از آن حذف کنید.

در فایل *Views/Account/Login.cshtml* نیز لینک Register را حذف کنید.

اپلیکیشن را دوباره منتشر کنید.

قدم‌های بعدی

برای اطلاعات بیشتر درباره نحوه استفاده از Facebook بعنوان یک تامین کننده احراز هویت، و اضافه کردن اطلاعات پروفایل به قسمت ثبت نام کاربران به لینک زیر مراجعه کنید. Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 and

OpenID Sign-on

برای یادگیری بیشتر درباره ASP.NET MVC 5 هم به سری مقالات Getting Started with ASP.NET MVC 5 می توانید مراجعه کنید. همچنین سری مقالات Getting Started with EF and MVC مطالب خوبی درباره مفاهیم پیشرفته EF ارائه می‌کند.

نظرات خوانندگان

نویسنده: مهمان
تاریخ: ۱۳۹۲/۱۰/۱۹ ۱۴:۴

دوست عزیز

با صبر و حوصله و دقت فراوان یک مقاله خوب را منتشر کردید. ممنون(رای من 5)

عنوان: **ساخت یک اپلیکیشن ساده ToDo با ASP.NET Identity**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۹ ۱۳:۳۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, ASP.NET MVC 5, ASP.NET Identity, MVC

یک سناریوی فرضی را در نظر بگیرید. اگر بخواهیم IdentityDbContext و دیگر DbContextهای اپلیکیشن را ادغام کنیم چه باید کرد؟ مثلا یک سیستم وبلاگ که برخی کاربران می‌توانند پست جدید ثبت کنند، برخی تنها می‌توانند کامنت بگذارند و تمامی کاربران هم اختیارات مشخص دیگری دارند. در چنین سیستمی شناسه کاربران (User ID) در بسیاری از مدل‌ها (موجودیت‌ها و مدل‌های اپلیکیشن) وجود خواهد داشت تا مشخص شود هر رکورد به کدام کاربر متعلق است. در این مقاله چنین سناریو هایی را بررسی می‌کنیم و best practiceهای مربوطه را مرور می‌کنیم.
در این پست یک اپلیکیشن ساده ToDo خواهیم ساخت که امکان تخصیص to-doها به کاربران را نیز فراهم می‌کند. در این مثال خواهیم دید که چگونه می‌توان مدل‌های مختص به سیستم عضویت (IdentityDbContext) را با مدل‌های دیگر اپلیکیشن مخلوط و استفاده کنیم.

تعریف نیازمندی‌های اپلیکیشن

تنها کاربران احراز هویت شده قادر خواهند بود تا لیست ToDoهای خود را ببینند، آیتم‌های جدید ثبت کنند یا داده‌های قبلی را ویرایش و حذف کنند.
کاربران نباید آیتم‌های ایجاد شده توسط دیگر کاربران را ببینند.
تنها کاربرانی که به نقش Admin تعلق دارند باید بتوانند تمام ToDoهای ایجاد شده را ببینند.

پس بگذارید ببینیم چگونه می‌شود اپلیکیشنی با ASP.NET Identity ساخت که پاسخگوی این نیازمندی‌ها باشد.
ابتدا یک پروژه ASP.NET MVC جدید با مدل احراز هویت Individual User Accounts بسازید. در این اپلیکیشن کاربران قادر خواهند بود تا بصورت محلی در وب سایت ثبت نام کنند و یا با تامین کنندگان دیگری مانند گوگل و فیسبوک وارد سایت شوند.
برای ساده نگاه داشتن این پست ما از حساب‌های کاربری محلی استفاده می‌کنیم.
در مرحله بعد ASP.NET Identity را راه اندازی کنید تا بتوانیم نقش مدیر و یک کاربر جدید بسازیم. می‌توانید با اجرای اپلیکیشن راه اندازی اولیه را انجام دهید. از آنجا که سیستم ASP.NET Identity توسط Entity Framework مدیریت می‌شود می‌توانید از تنظیمات پیکربندی Code First برای راه اندازی دیتابیس خود استفاده کنید.
در قدم بعدی راه انداز دیتابیس را در Global.asax تعریف کنید.

```
Database.SetInitializer<MyDbContext>(new MyDbInitializer());
```

ایجاد نقش مدیر و کاربر جدیدی که به این نقش تعلق دارد

اگر به قطعه کد زیر دقت کنید، می‌بینید که در خط شماره 5 متغیری از نوع UserManager ساخته ایم که امکان اجرای عملیات گوناگونی روی کاربران را فراهم می‌کند. مانند ایجاد، ویرایش، حذف و اعتبارسنجی کاربران. این کلاس که متعلق به سیستم ASP.NET Identity است همتای SQLMembershipProvider در ASP.NET 2.0 است.
در خط 6 یک RoleManager می‌سازیم که امکان کار با نقش‌ها را فراهم می‌کند. این کلاس همتای SQLRoleMembershipProvider در ASP.NET 2.0 است.
در این مثال نام کلاس کاربران (موجودیت کاربر در IdentityDbContext) برابر با "MyUser" است، اما نام پیش فرض در قالب‌های پروژه VS 2013 برابر با "ApplicationUser" می‌باشد.

```
public class MyDbInitializer : DropCreateDatabaseAlways<MyDbContext>
     {
          protected override void Seed(MyDbContext context)
          {
              var UserManager = new UserManager<MyUser>(new

                                                UserStore<MyUser>(context));

              var RoleManager = new RoleManager<IdentityRole>(new
                                            RoleStore<IdentityRole>(context));
```

```
            string name = "Admin";
            string password = "123456";


            //Create Role Admin if it does not exist
            if (!RoleManager.RoleExists(name))
            {
                var roleresult = RoleManager.Create(new IdentityRole(name));
            }

            //Create User=Admin with password=123456
            var user = new MyUser();
            user.UserName = name;
            var adminresult = UserManager.Create(user, password);

            //Add User Admin to Role Admin
            if (adminresult.Succeeded)
            {
                var result = UserManager.AddToRole(user.Id, name);
            }
            base.Seed(context);
        }
    }
```

حال فایلی با نام Models/AppModels.cs بسازید و مدل EF Code First اپلیکیشن را تعریف کنید. از آنجا که از EF استفاده می‌کنیم، روابط کلیدها بین کاربران و ToDoها بصورت خودکار برقرار می‌شود.

```
public class MyUser : IdentityUser
      {
          public string HomeTown { get; set; }
          public virtual ICollection<ToDo>
                              ToDoes { get; set; }
      }

      public class ToDo
      {
          public int Id { get; set; }
          public string Description { get; set; }
          public bool IsDone { get; set; }
          public virtual MyUser User { get; set; }
      }
```

در قدم بعدی با استفاده از مکانیزم Scaffolding کنترلر جدیدی بهمراه تمام Viewها و متدهای لازم برای عملیات CRUD بسازید.
برای اطلاعات بیشتر درباره نحوه استفاده از مکانیزم Scaffolding به این لینک مراجعه کنید.
لطفا دقت کنید که از DbContext فعلی استفاده کنید. این کار مدیریت داده‌های Identity و اپلیکیشن شما را یکپارچه‌تر می‌کند.
DbContext شما باید چیزی شبیه به کد زیر باشد.

```
public class MyDbContext : IdentityDbContext<MyUser>
      {
          public MyDbContext()
              : base("DefaultConnection")
          {
           }

           protected override void OnModelCreating(DbModelBuilder modelBuilder)
           {
          public System.Data.Entity.DbSet<AspnetIdentitySample.Models.ToDo>
                     ToDoes { get; set; }
      }
```

تنها کاربران احراز هویت شده باید قادر به اجرای عملیات CRUD باشند

برای این مورد از خاصیت Authorize استفاده خواهیم کرد که در MVC 4 هم وجود داشت. برای اطلاعات بیشتر لطفا به این لینک مراجعه کنید.

```
[Authorize]
public class ToDoController : Controller
```

کنترلر ایجاد شده را ویرایش کنید تا کاربران را به ToDoها اختصاص دهد. در این مثال تنها اکشن متدهای Create و List را بررسی خواهیم کرد. با دنبال کردن همین روش می‌توانید متدهای Edit و Delete را هم بسادگی تکمیل کنید.
یک متد constructor جدید بنویسید که آبجکتی از نوع UserManager می‌پذیرد. با استفاده از این کلاس می‌توانید کاربران را در ASP.NET Identity مدیریت کنید.

```
private MyDbContext db;
          private UserManager<MyUser> manager;
          public ToDoController()
          {
              db = new MyDbContext();
              manager = new UserManager<MyUser>(new UserStore<MyUser>(db));
          }
```

اکشن متد Create را بروز رسانی کنید

هنگامی که یک ToDo جدید ایجاد می‌کنید، کاربر جاری را در ASP.NET Identity پیدا می‌کنیم و او را به ToDoها اختصاص می‌دهیم.

```
public async Task<ActionResult> Create
          ([Bind(Include="Id,Description,IsDone")] ToDo todo)
          {
              var currentUser = await manager.FindByIdAsync
                                              (User.Identity.GetUserId());
              if (ModelState.IsValid)
              {
                  todo.User = currentUser;
                  db.ToDoes.Add(todo);
                  await db.SaveChangesAsync();
                  return RedirectToAction("Index");
              }

              return View(todo);
          }
```

اکشن متد List را بروز رسانی کنید

در این متد تنها ToDoهای کاربر جاری را باید بگیریم.

```
public ActionResult Index()
          {
              var currentUser = manager.FindById(User.Identity.GetUserId());

               return View(db.ToDoes.ToList().Where(
                                  todo => todo.User.Id == currentUser.Id));
          }
```

تنها مدیران سایت باید بتوانند تمام ToDoها را ببینند

بدین منظور ما یک اکشن متد جدید به کنترل مربوطه اضافه می‌کنیم که تمام ToDoها را لیست می‌کند. اما دسترسی به این متد را تنها برای کاربرانی که در نقش مدیر وجود دارند میسر می‌کنیم.

```
[Authorize(Roles="Admin")]
          public async Task<ActionResult> All()
          {
```

```
            return View(await db.ToDoes.ToListAsync());
        }
```

نمایش جزئیات کاربران از جدول ToDo ها

از آنجا که ما کاربران را به ToDo هایشان مرتبط می‌کنیم، دسترسی به داده‌های کاربر ساده است. مثلا در متدی که مدیر سایت تمام آیتم‌ها را لیست می‌کند می‌توانیم به اطلاعات پروفایل تک تک کاربران دسترسی داشته باشیم و آنها را در نمای خود بگنجانیم. در این مثال تنها یک فیلد بنام **HomeTown** اضافه شده است، که آن را در کنار اطلاعات ToDo نمایش می‌دهیم.

```
@model IEnumerable<AspnetIdentitySample.Models.ToDo>

  @{
    ViewBag.Title = "Index";
  }

  <h2>List of ToDoes for all Users</h2>
  <p>
      Notice that we can see the User info (UserName) and profile info such as HomeTown for the user as
well.
      This was possible because we associated the User object with a ToDo object and hence
      we can get this rich behavior.
  12:  </p>

  <table class="table">
      <tr>
          <th>
              @Html.DisplayNameFor(model => model.Description)
          </th>
          <th>
              @Html.DisplayNameFor(model => model.IsDone)
          </th>
          <th>@Html.DisplayNameFor(model => model.User.UserName)</th>
          <th>@Html.DisplayNameFor(model => model.User.HomeTown)</th>
      </tr>
  25:
  26:      @foreach (var item in Model)
  27:      {
  28:          <tr>
  29:              <td>
  30:                  @Html.DisplayFor(modelItem => item.Description)
  31:              </td>
  32:              <td>
                  @Html.DisplayFor(modelItem => item.IsDone)
              </td>
              <td>
                  @Html.DisplayFor(modelItem => item.User.UserName)
              </td>
              <td>
                  @Html.DisplayFor(modelItem => item.User.HomeTown)
              </td>
          </tr>
      }

  </table>
```

صفحه Layout را بروز رسانی کنید تا به ToDoها لینک شود

```
<li>@Html.ActionLink("ToDo", "Index", "ToDo")</li>
 <li>@Html.ActionLink("ToDo for User In Role Admin", "All", "ToDo")</li>
```

حال اپلیکیشن را اجرا کنید. همانطور که مشاهده می‌کنید دو لینک جدید به منوی سایت اضافه شده اند.

ساخت یک ToDo بعنوان کاربر عادی

روی لینک ToDo کلیک کنید، باید به صفحه ورود هدایت شوید چرا که دسترسی تنها برای کاربران احراز هویت شده تعریف وجود دارد. می‌توانید یک حساب کاربری محلی ساخته، با آن وارد سایت شوید و یک ToDo بسازید.

پس از ساختن یک ToDo می‌توانید لیست رکوردهای خود را مشاهده کنید. دقت داشته باشید که رکوردهایی که کاربران دیگر ثبت کرده اند برای شما نمایش داده نخواهند شد.

مشاهده تمام ToDoها بعنوان مدیر سایت

روی لینک **ToDoes for User in Role Admin** کلیک کنید. در این مرحله باید مجددا به صفحه ورود هدایت شوید چرا که شما در نقش مدیر نیستید و دسترسی کافی برای مشاهده صفحه مورد نظر را ندارید. از سایت خارج شوید و توسط حساب کاربری مدیری که هنگام راه اندازی اولیه دیتابیس ساخته اید وارد سایت شوید.

```
User = Admin
Password = 123456
```

پس از ورود به سایت بعنوان یک مدیر، می‌توانید ToDoهای ثبت شده توسط تمام کاربران را مشاهده کنید.

نظرات خوانندگان

نویسنده: اس ام
تاریخ: ۱۳۹۳/۰۱/۰۱ ۲۰:۲۴

میشه این پروژه رو برا دانلود بزارید؟ ممنون.

نویسنده: میثم سلیمانی
تاریخ: ۱۳۹۳/۰۴/۲۸ ۱۷:۴۰

```
var currentUser = await manager.FindByIdAsync(User.Identity.GetUserId());
```

این GetUserId() چرا وجود نداره؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۲۸ ۱۷:۴۹

در فضای نام Microsoft.AspNet.Identity تعریف شده.

نویسنده: محمد رحیم پورابراهیم
تاریخ: ۱۳۹۳/۱۱/۲۳ ۱۹:۱۳

سلام
اگر پروژه ای داشته باشیم با ساختار زیر
MyProject
MyProject.Service
MyProject.Model
(MyProject.Data (Entity framework code first
اگر بخوایم تو لایه MyProject از asp .net identity استفاده کنیم، به چه نحوی باید این کار رو انجام داد که ساختار دیتابیس که در لایه دیتا دچار مشکل نشه و جداول مورد نیاز asp .net identity به دیتا بیس اضافه بشه؟

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۱/۲۳ ۲۰:۱۷

مطلب و پروژه اعمال تزریق وابستگی‌ها به مثال رسمی ASP.NET Identity رو مطالعه کنید.

عنوان: دریافت اطلاعات بیشتر از Social Provider ها در VS 2013
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۹ ۱۷:۳۵
آدرس: www.dotnettips.info
گروه‌ها: OpenID, ASP.NET MVC 5, ASP.NET Identity, OAuth, MVC, Security

هنگامی که یک پروژه جدید ASP.NET را در VS 2013 می‌سازید و متد احراز هویت آن را Individual User Accounts انتخاب می‌کنید، قالب پروژه، امکانات لازم را برای استفاده از تامین کنندگان ثالث، فراهم می‌کند، مثلا مایکروسافت، گوگل، توییتر و فیسبوک. هنگامی که توسط یکی از این تامین کننده‌ها کاربری را احراز هویت کردید، می‌توانید اطلاعات بیشتری درخواست کنید. مثلا عکس پروفایل کاربر یا لیست دوستان او. سپس اگر کاربر به اپلیکیشن شما سطح دسترسی کافی داده باشد می‌توانید این اطلاعات را دریافت کنید و تجربه کاربری قوی‌تر و بهتری ارائه کنید.

در این پست خواهید دید که چطور می‌شود از تامین کننده Facebook اطلاعات بیشتری درخواست کرد. پیش فرض این پست بر این است که شما با احراز هویت فیسبوک و سیستم کلی تامین کننده‌ها آشنایی دارید. برای اطلاعات بیشتر درباره راه اندازی احراز هویت فیسبوک به این لینک مراجعه کنید.

برای دریافت اطلاعات بیشتر از فیسبوک مراحل زیر را دنبال کنید.

یک اپلیکیشن جدید ASP.NET MVC با تنظیمات Individual User Accounts بسازید.

احراز هویت فیسبوک را توسط کلید هایی که از Facebook دریافت کرده اید فعال کنید. برای اطلاعات بیشتر در این باره می‌توانید به این لینک مراجعه کنید.

برای درخواست اطلاعات بیشتر از فیسبوک، فایل Startup.Auth.cs را مطابق لیست زیر ویرایش کنید.

```
List<string> scope = newList<string>() { "email", "user_about_me", "user_hometown", "friends_about_me",
"friends_photos" };
 var x = newFacebookAuthenticationOptions();
 x.Scope.Add("email");
 x.Scope.Add("friends_about_me");
 x.Scope.Add("friends_photos");
 x.AppId = "636919159681109";
 x.AppSecret = "f3c16511fe95e854cf5885c10f83f26f";
 x.Provider = newFacebookAuthenticationProvider()
{
    OnAuthenticated = async context =>
    {
         //Get the access token from FB and store it in the database and
        //use FacebookC# SDK to get more information about the user
        context.Identity.AddClaim(
        new System.Security.Claims.Claim("FacebookAccessToken",
                                                 context.AccessToken));
    }
};
 x.SignInAsAuthenticationType = DefaultAuthenticationTypes.ExternalCookie;
 app.UseFacebookAuthentication(x);
```

در خط 1 مشخص می‌کنیم که چه scope هایی از داده را می‌خواهیم درخواست کنیم.

از خط 10 تا 17 رویداد OnAuthenticated را مدیریت می‌کنیم که از طرف Facebook OWIN authentication اجرا می‌شود. این متد هر بار که کاربری با فیسبوک خودش را احراز هویت می‌کند فراخوانی می‌شود. پس از آنکه کاربر احراز هویت شد و به اپلیکیشن سطح دسترسی لازم را اعطا کرد، تمام داده‌ها در FacebookContext ذخیره می‌شوند.

خط 14 شناسه FacebookAccessToken را ذخیره می‌کند. ما این آبجکت را از فیسبوک دریافت کرده و از آن برای دریافت لیست دوستان کاربر استفاده می‌کنیم.

نکته: در این مثال تمام داده‌ها بصورت Claims ذخیره می‌شوند، اما اگر بخواهید می‌توانید از ASP.NET Identity برای ذخیره آنها در دیتابیس استفاده کنید.

در قدم بعدی لیست دوستان کاربر را از فیسبوک درخواست می‌کنیم. ابتدا فایل Views/Shared/_LoginPartial.cshtml را باز کنید و لینک زیر را به آن بیافزایید.

```
<li>
     @Html.ActionLink("FacebookInfo", "FacebookInfo","Account")
</li>
```

هنگامی که کاربری وارد سایت می‌شود و این لینک را کلیک می‌کند، ما لیست دوستان او را از فیسبوک درخواست می‌کنیم و بهمراه عکس‌های پروفایل شان آنها را لیست می‌کنیم.

تمام Claimها را از **UserIdentity** بگیرید و آنها را در دیتابیس ذخیره کنید. در این قطعه کد ما تمام Claim هایی که توسط OWIN دریافت کرده ایم را می‌خوانیم، و شناسه FacebookAccessToken را در دیتابیس عضویت ASP.NET Identity ذخیره می‌کنیم.

```
//
        // GET: /Account/LinkLoginCallback
        publicasyncTask<ActionResult> LinkLoginCallback()
        {
            var loginInfo = await AuthenticationManager.GetExternalLoginInfoAsync(XsrfKey,
User.Identity.GetUserId());
            if (loginInfo == null)
            {
                return RedirectToAction("Manage", new { Message = ManageMessageId.Error });
            }
            var result = await UserManager.AddLoginAsync(User.Identity.GetUserId(), loginInfo.Login);
            if (result.Succeeded)
            {
                var currentUser = await UserManager.FindByIdAsync(User.Identity.GetUserId());
                //Add the Facebook Claim
                await StoreFacebookAuthToken(currentUser);
                return RedirectToAction("Manage");
            }
            return RedirectToAction("Manage", new { Message = ManageMessageId.Error });
        }
```

خط 14-15 شناسه FacebookAccessToken را در دیتابیس ذخیره می‌کند.

StoreFacebookAuthToken تمام اختیارات (claim)های کاربر را از UserIdentity می‌گیرد و Access Token را در قالب یک User Claim در دیتابیس ذخیره می‌کند. اکشن LinkLoginCallback هنگامی فراخوانی می‌شود که کاربر وارد سایت شده و یک تامین کننده دیگر را می‌خواهد تنظیم کند.

اکشن ExternalLoginConfirmation هنگام اولین ورود شما توسط تامین کنندگان اجتماعی مانند فیسبوک فراخوانی می‌شود.

در خط 26 پس از آنکه کاربر ایجاد شد ما یک FacebookAccessToken را بعنوان یک Claim برای کاربر ذخیره می‌کنیم.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ExternalLoginConfirmation(ExternalLoginConfirmationViewModel model,
string returnUrl)
        {
            if (User.Identity.IsAuthenticated)
            {
                return RedirectToAction("Manage");
            }

            if (ModelState.IsValid)
            {
                // Get the information about the user from the external login provider
                var info = await AuthenticationManager.GetExternalLoginInfoAsync();
                if (info == null)
                {
                    return View("ExternalLoginFailure");
                }
                var user = newApplicationUser() { UserName = model.Email };
                var result = await UserManager.CreateAsync(user);
                if (result.Succeeded)
                {
                    result = await UserManager.AddLoginAsync(user.Id, info.Login);
                    if (result.Succeeded)
                    {
                        await StoreFacebookAuthToken(user);
                        await SignInAsync(user, isPersistent: false);
                        return RedirectToLocal(returnUrl);
                    }
                }
                AddErrors(result);
            }

            ViewBag.ReturnUrl = returnUrl;
```

```
            return View(model);
        }
```

اکشن **ExternalLoginCallback** هنگامی فراخوانی می‌شود که شما برای اولین بار یک کاربر را به یک تامین کننده اجتماعی اختصاص می‌دهید. در خط 17 شناسه دسترسی فیسبوک را بصورت یک claim برای کاربر ذخیره می‌کنیم.

```
//
        // GET: /Account/ExternalLoginCallback
        [AllowAnonymous]
        publicasyncTask<ActionResult> ExternalLoginCallback(string returnUrl)
        {
            var loginInfo = await AuthenticationManager.GetExternalLoginInfoAsync();
            if (loginInfo == null)
            {
                return RedirectToAction("Login");
            }

            // Sign in the user with this external login provider if the user already has a login
            var user = await UserManager.FindAsync(loginInfo.Login);
            if (user != null)
            {
                //Save the FacebookToken in the database if not already there
                await StoreFacebookAuthToken(user);
                await SignInAsync(user, isPersistent: false);
                return RedirectToLocal(returnUrl);
            }
            else
            {
                // If the user does not have an account, then prompt the user to create an account
                ViewBag.ReturnUrl = returnUrl;
                ViewBag.LoginProvider = loginInfo.Login.LoginProvider;
                return View("ExternalLoginConfirmation", newExternalLoginConfirmationViewModel { Email
= loginInfo.Email });
            }
        }
```

در آخر شناسه FacebookAccessToken را در دیتابیس ASP.NET Identity ذخیره کنید.

```
privateasyncTask StoreFacebookAuthToken(ApplicationUser user)
        {
            var claimsIdentity = await
AuthenticationManager.GetExternalIdentityAsync(DefaultAuthenticationTypes.ExternalCookie);
            if (claimsIdentity != null)
            {
                // Retrieve the existing claims for the user and add the FacebookAccessTokenClaim
                var currentClaims = await UserManager.GetClaimsAsync(user.Id);
                var facebookAccessToken = claimsIdentity.FindAll("FacebookAccessToken").First();
                if (currentClaims.Count() <=0 )
                {
                    await UserManager.AddClaimAsync(user.Id, facebookAccessToken);
                }
```

پکیج Facebook C#SDK را نصب کنید. http://nuget.org/packages/Facebook

فایل AccountViewModel.cs را باز کنید و کد زیر را اضافه کنید.

```
public class FacebookViewModel
     {
         [Required]
         [Display(Name = "Friend's name")]
         public string Name { get; set; }

        public string ImageURL { get; set; }
    }
```

کد زیر را به کنترلر Account اضافه کنید تا عکس‌های دوستان تان را دریافت کنید.

```
//GET: Account/FacebookInfo
[Authorize]
publicasyncTask<ActionResult> FacebookInfo()
{
    var claimsforUser = await UserManager.GetClaimsAsync(User.Identity.GetUserId());
    var access_token = claimsforUser.FirstOrDefault(x => x.Type == "FacebookAccessToken").Value;
    var fb = newFacebookClient(access_token);
    dynamic myInfo = fb.Get("/me/friends");
    var friendsList = newList<FacebookViewModel>();
    foreach (dynamic friend in myInfo.data)
    {
        friendsList.Add(newFacebookViewModel()
           {
               Name = friend.name,
               ImageURL = @"https://graph.facebook.com/" + friend.id + "/picture?type=large"
           });
    }

    return View(friendsList);
}
```

در پوشه Views/Account یک نمای جدید با نام FacebookInfo.cshtml بسازید و کد Markup آن را مطابق لیست زیر تغییر دهید.

```
@model IList<WebApplication96.Models.FacebookViewModel>
 @if (Model.Count > 0)
 {
     <h3>List of friends</h3>
     <div class="row">
             @foreach (var friend in Model)
             {
               <div class="col-md-3">
                <a href="#" class="thumbnail">
                  <img src=@friend.ImageURL alt=@friend.Name />
                 </a>
               </div>
              }
     </div>
 }
```

در این مرحله، شما می‌توانید لیست دوستان خود را بهمراه عکس‌های پروفایل شان دریافت کنید.

پروژه را اجرا کنید و توسط Facebook وارد سایت شوید. باید به سایت فیسبوک هدایت شوید تا احراز هویت کنید و دسترسی لازم را به اپلیکیشن اعطا کنید. پس از آن مجددا به سایت خودتان باید هدایت شوید.

حال هنگامی که روی لینک FacebookInfo کلیک می‌کنید باید صفحه ای مشابه تصویر زیر ببینید.

این یک مثال ساده از کار کردن با تامین کنندگان اجتماعی بود. همانطور که مشاهده می‌کنید، براحتی می‌توانید داده‌های بیشتری برای کاربر جاری درخواست کنید و تجربه کاربری و امکانات بسیار بهتری را در اپلیکیشن خود فراهم کنید.

نظرات خوانندگان

نویسنده: حامد شیربندی
تاریخ: ۱۳۹۳/۱۱/۲۹ ۱۹:۳

با سلام و تشکر
توی دریافت اطلاعات لیست دوستان از فیسبوک باید دقت داشت که طبق مستندات فیسبوک برای Graph API در اینجا ما فقط به اطلاعات دوستانی دسترسی خواهیم داشت که از Facebook Login استفاده کرده باشند.
البته این نکته رو هم باید اضافه کنم که برای دسترسی به هر اطلاعاتی از اکانت فیسبوک کاربر، باید مجوزش رو هم ارسال کنیم
بنابراین برای دریافت اطلاعات مربوط به لیست دوستان باید دستور زیر رو هم به کلاس startUp اضافه کنیم

```
x.Scope.Add("user_friends");
```

برای اطلاع از نام کل مجوزها و حتی تست اونها برای دریافت اطلاعات اکانت از Graph API Explorer استفاده کنید.

عنوان: **سفارشی کردن ASP.NET Identity در MVC 5**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۹ ۱۹:۵۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, MemberShip, ASP.NET MVC 5, ASP.NET Identity, MVC, Security

یکی از نیازهای رایج توسعه دهندگان هنگام استفاده از سیستم عضویت ASP.NET سفارشی کردن الگوی داده‌ها است. مثلا ممکن است بخواهید یک پروفایل سفارشی برای کاربران در نظر بگیرید، که شامل اطلاعات شخصی، آدرس و تلفن تماس و غیره می‌شود. یا ممکن است بخواهید به خود فرم ثبت نام فیلدهای جدیدی اضافه کنید و آنها را در رکورد هر کاربر ذخیره کنید.
یکی از مزایای ASP.NET Identity این است که بر پایه EF Code First نوشته شده است. بنابراین سفارشی سازی الگوی دیتابیس و اطلاعات کاربران ساده است.
یک اپلیکیشن جدید ASP.NET MVC بسازید و نوع احراز هویت را Individual User Accounts انتخاب کنید. پس از آنکه پروژه جدید ایجاد شد فایل IdentityModels.cs را در پوشه Models باز کنید. کلاسی با نام *ApplicationUser* مشاهده می‌کنید که همتای *UserProfile* در فریم ورک SimpleMembership است. این کلاس خالی است و از کلاس *IdentityUser* ارث بری می‌کند و شامل خواص زیر است.

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

        public virtual ICollection<identityuserclaim> Claims { get; }
        public virtual string Id { get; set; }
        public virtual ICollection<identityuserlogin> Logins { get; }
        public virtual string PasswordHash { get; set; }
        public virtual ICollection<identityuserrole> Roles { get; }
        public virtual string SecurityStamp { get; set; }
        public virtual string UserName { get; set; }
    }
```

اگر دقت کنید خواهید دید که فیلد Id بر خلاف SimpleMembership یک عدد صحیح یا int نیست، بلکه بصورت یک رشته ذخیره می‌شود. پیاده سازی پیش فرض ASP.NET Identity مقدار این فیلد را با یک GUID پر می‌کند. در این پست تنها یک فیلد آدرس ایمیل به کلاس کاربر اضافه می‌کنیم. با استفاده از همین فیلد در پست‌های آتی خواهیم دید چگونه می‌توان ایمیل‌های تایید ثبت نام برای کاربران ارسال کرد. کلاس ApplicationUser بدین شکل خواهد بود.

```
public class ApplicationUser : IdentityUser
{
    public string Email { get; set; }
}
```

حال برای آنکه کاربر بتواند هنگام ثبت نام آدرس ایمیل خود را هم وارد کند، باید مدل فرم ثبت نام را بروز رسانی کنیم.

```
public class RegisterViewModel
{
    [Required]
    [Display(Name = "User name")]
    public string UserName { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength =
6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }

    [Required]
    [Display(Name = "Email address")]
```

```
    public string Email { get; set; }

}
```

سپس فایل View را هم بروز رسانی می‌کنیم تا یک برچسب و تکست باکس برای آدرس ایمیل نمایش دهد.

```
<div class="form-group">
     @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
     <div class="col-md-10">
         @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
     </div>
 </div>
```

برای تست این تغییرات، صفحه About را طوری تغییر می‌دهید تا آدرس ایمیل کاربر جاری را نمایش دهد. این قسمت همچنین نمونه ای از نحوه دسترسی به اطلاعات کاربران است.

```
public ActionResult About()
{
    ViewBag.Message = "Your application description page.";
    UserManager<ApplicationUser> UserManager = new UserManager<ApplicationUser>(new
UserStore<ApplicationUser>(new ApplicationDbContext()));
    var user = UserManager.FindById(User.Identity.GetUserId());
    if (user != null)
        ViewBag.Email = user.Email;
    else
        ViewBag.Email = "User not found.";

    return View();
}
```

همین! تمام کاری که لازم بود انجام دهید همین بود. از آنجا که سیستم ASP.NET Identity توسط Entity Framework مدیریت می‌شود، روی الگوی دیتابیس سیستم عضویت کنترل کامل دارید. بنابراین به سادگی می‌توانید با استفاده از قابلیت Code First مدل‌های خود را سفارشی کنید.

در پست‌های آتی این مطلب را ادامه خواهیم داد تا ببینیم چگونه می‌توان ایمیل‌های تاییدیه برای کاربران ارسال کرد.

نظرات خوانندگان

نویسنده: رجایی
تاریخ: ۱۳۹۲/۱۱/۰۸ ۱۹:۲۰

سلام؛ از زحماتتون بسیار تشکر می‌کنم. من وب سایت را به روش چند لایه‌ی مرسوم می‌سازم:

data layer - domain models - website - service layer

با توجه به آن چگونه می‌توان از asp.net identity استفاده کرد؟ زیرا مثلا نیاز است از کلید جدول users در مدل‌های دیگه استفاده شود. آیا این کلید را باید در لایه دومین استفاده کرد؟

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۸ ۲۳:۴۵

موجودیت‌های مربوط به ASP.NET Identity رو در لایه مدل‌ها قرار بدین و از یک DbContext استفاده کنید. یعنی DbSetهای مدل برنامه و Identity رو در یک کانتکست تعریف کنید.

نویسنده: رجایی
تاریخ: ۱۳۹۲/۱۱/۱۰ ۱۲:۴

سلام...ممنون از پاسختون.

با توجه به راهنمایی شما در قسمت context در لایه data layer بدین صورت درج کردم

```
public class Context : DbContext
    {
        public DbSet<IdentityUserClaim> AspNetUserClaims { set; get; }
        public DbSet<IdentityRole> AspNetRoles { set; get; }
        public DbSet<IdentityUserLogin> AspNetUserLogins { set; get; }
        public DbSet<IdentityUserRole> AspNetUserRoles { set; get; }
        public DbSet<IdentityUser> AspNetUsers { set; get; }
        //-------------------------------------
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<IdentityRole>().HasKey<string>(r => r.Id).ToTable("AspNetRoles");
            modelBuilder.Entity<IdentityUser>().ToTable("AspNetUsers");
            modelBuilder.Entity<IdentityUserLogin>().HasKey(l => new { l.UserId, l.LoginProvider,
l.ProviderKey }).ToTable("AspNetUserLogins");
            modelBuilder.Entity<IdentityUserRole>().HasKey(r => new { r.RoleId, r.UserId
}).ToTable("AspNetUserRoles");
            modelBuilder.Entity<IdentityUserClaim>().ToTable("AspNetUserClaims");
        }
    }
```

ولی وقتی سایت اجرا می‌شود ایراد زیر نمایش داده می‌شود

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۱۰ ۱۲:۱۴

خطا رو فراموش کردید ارسال کنید.

نویسنده: مهرداد راهی
تاریخ: ۱۳۹۲/۱۱/۱۱ ۰:۵۵

سلام من MVC کار نمیکنم توی ASP.Net وب‌فرمز استفاده میکنم از این امکان. فایل IdentityModels.cs رو نداره توی پروژه .
مشکل کجاس؟
-enable migrations هم زدم خطا داد

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۱۱ ۱:۵۳

از VS 2013 استفاده می‌کنی ؟

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۱۱ ۲:۵۱

لازم نیست تمام این آبجکت‌ها رو به context نگاشت کنید. قالب پروژه‌های VS 2013 بصورت خودکار در پوشه Models کلاسی بنام IdentityModels میسازه. این کلاس شامل کلاسی بنام ApplicationDbContext میشه که تعریفی مانند لیست زیر داره:

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

این کلاس رو کلا حذف کنید، چون قراره از یک DbContext برای تمام موجودیت‌ها استفاده کنید.
کلاس ApplicationUser که معرف موجودیت کاربران هست رو در لایه دامنه‌ها تعریف کنید و دقت کنید که باید از IdentityUser ارث بری کنه، حال با نام پیش فرض یا با نام دلخواه. سپس باید کلاسی بسازید که از UserManager<T> مشتق میشه. با استفاده از این کلاس می‌تونید به موجودیت‌های کاربران دسترسی داشته باشید. بعنوان مثال:

```
public class AppUserManager : UserManager<AppUser>{
    public AppUserManager() : base(new UserStore<AppUser>(new ShirazBilitDbContext())) { }
}
```

همونطور که می‌بینید کلاس موجودیت کاربر در اینجا AppUser نام داره، پس هنگام استفاده از UserManager نوع داده رو بهش نگاشت می‌کنیم. کد کلاس AppUser هم مطابق لیست زیر خواهد بود.

```
public class AppUser : IdentityUser
{
    public string Email { get; set; }
    public string ConfirmationToken { get; set; }
    public bool IsConfirmed { get; set; }
}
```

همونطور که مشخصه کلاس کاربران سفارشی سازی شده و سه فیلد به جدول کاربران اضافه کردیم. فیلدهای بیشتر یا موجودیت پروفایل کاربران هم باید به همین کلاس افزوده بشن. اگر پست‌ها رو بیاد بیارید گفته شد که ASP.NET Identity با مدل EF Code-First کار میکنه.

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۱۱ ۲:۵۷

از VS 2013 استفاده کنید و .NET 4.5.
اگر این فایل برای شما ایجاد نمیشه پس در قالب پروژه‌های Web Forms وجود نداره. ارتباطی با مهاجرت‌ها هم نداره، کلاس موجودیت کاربر رو خودتون می‌تونید ایجاد کنید. اگر به نظرات بالا مراجعه کنید گفته شد که کلاس کاربران باید از

IdentityModels ارث بری کنه.

نویسنده: رجایی
تاریخ: ۱۳۹۲/۱۱/۱۲ ۹:۲۱

عذر خواهی می‌کنم فراموش کردم. ایراد بدین صورت است:

System.InvalidOperationException: The model backing the 'Context' context has changed since the database was created. Consider using Code First Migrations to update the database (http://go.microsoft.com/fwlink/?LinkId=238269).

مشخص است که می‌گوید context تغییر می‌کند.ولی من از migration استفاده می‌کنم و codefirst ولی باز هم این ایراد رو در اتصال به دیتابیس نشان می‌دهد. من از add-migration هم استفاده می‌کنم تا تغییرات موجودیت‌ها رو کامل به من نشان دهد که چیزی را عنوان نمی‌کند.

نویسنده: افتاب
تاریخ: ۱۳۹۲/۱۲/۲۷ ۲۳:۳۸

سلام و متشکر ،
دنبال آن میگشتم که چه طوری میشه دو تا صفحه لوگین در پروژه داشت که ورودی کاربران از مدیران جدا باشد

نویسنده: افتاب
تاریخ: ۱۳۹۲/۱۲/۲۷ ۲۳:۴۴

میشه در مورد «این کلاس رو کلا حذف کنید، چون قراره از یک DbContext برای تمام موجودیت‌ها استفاده کنید....» بیشتر توضیح بفرمایید؟

نویسنده: سعید رضایی
تاریخ: ۱۳۹۳/۰۲/۰۱ ۱۶:۴۱

با عرض سلام.تو Vs2012+mvc4 نمیشه از identity استفاده کرد؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۰۱ ۱۷:۲۶

خیر. با دات نت 4.5 کامپایل شده.

نویسنده: میثم سلیمانی
تاریخ: ۱۳۹۳/۰۵/۲۶ ۱۱:۷

با سلام و احترام
من دستور زیر رو تو asp.net mvc Identity sample 2 تو اکشن Login اضافه کردم

```
UserManager<ApplicationUser> UserManager = new UserManager<ApplicationUser>(new
UserStore<ApplicationUser>(new ApplicationDbContext()));
    var user = UserManager.FindById(User.Identity.GetUserId());
```

اما user و null میده !
اکشن login

```
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 SignInManager.PasswordSignInAsync(model.Email, model.Password,
model.RememberMe, shouldLockout: false);
            switch (result)
            {

                case SignInStatus.Success:
                    {
                        UserManager<ApplicationUser> UserManager = new UserManager<ApplicationUser>(new
UserStore<ApplicationUser>(new ApplicationDbContext()));
                        var user = UserManager.FindById(User.Identity.GetUserId());
                        return RedirectToLocal(returnUrl);
                    }
                case SignInStatus.LockedOut:
                    return View("Lockout");
                case SignInStatus.RequiresVerification:
                    return RedirectToAction("SendCode", new { ReturnUrl = returnUrl });
                case SignInStatus.Failure:
                default:
                    ModelState.AddModelError("", "Invalid login attempt.");
                    return View(model);
            }
        }
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۲۹ ۱۳:۴۴

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

نویسنده: ایمان صالحی
تاریخ: ۱۳۹۳/۱۰/۱۷ ۱۵:۵۰

من DataSetهای مربوط به پایگاه داده خودم رو در ApplicationDbContext مینویسم و پایگاه داده اجرا میشه و مشکلی پیش نمیاد..کار هم میکنه..اما سوالم اینه که کارم از نظر استاندارد بودن مشکلی داره یا نه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۷ ۱۵:۵۶

مشکلی ندارد. ApplicationDbContext از IdentityDbContext مشتق می‌شود که آن هم از DbContext مشتق شده:

```
public class IdentityDbContext<TUser, TRole, TKey, TUserLogin, TUserRole, TUserClaim> :
DbContext where TUser : IdentityUser<TKey, TUserLogin, TUserRole, TUserClaim>
where TRole : IdentityRole<TKey, TUserRole>
where TUserLogin : IdentityUserLogin<TKey>
where TUserRole : IdentityUserRole<TKey>
where TUserClaim : IdentityUserClaim<TKey>
```

یعنی نیازی به چند DbContext سفارشی در برنامه ندارید. همان ApplicationDbContext ، در حقیقت Context اصلی کاری برنامه است.

نویسنده: سانی
تاریخ: ۱۳۹۳/۱۱/۲۵ ۱۲:۴۶

سلام

این سوال رو توی asp.net , barnamenevis هم پرسیدم. من برای Identity از sample خود MVC استفاده کردم؛ به این صورت که با استفاده از nuget آن را نصب کرم. حال توی یک لیست بازشو میخوام فقط کاربرانی را نشان دهم که یک نقش را دارند.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۲۵ ۱۴:۵۱

مثال مطلب اعمال تزریق وابستگی‌ها به مثال رسمی ASP.NET Identity ، جهت تکمیل کلاس ApplicationRoleManager آن بهبود داده شد (^). برای نمونه :

```
public IList<ApplicationUser> GetApplicationUsersInRole(string roleName)
{
    var selectedUserIds = from role in this.Roles
                          where role.Name == roleName
                          from user in role.Users
                          select user.UserId;
    return _users.Where(applicationUser => selectedUserIds.Contains(applicationUser.Id)).ToList();
}
```

نویسنده: مهدی عطار
تاریخ: ۱۳۹۴/۰۲/۲۶ ۲۰:۱

با سلام وتشکر از مطالب خوبتون
یه سوال برام پیش اومده راجع به identity اگه قرار باشه به عنوان مثال به اکشن ویرایش محصول دسترسی مدیر بدیم و بعد بخواهیم آن را از داخل سایت تغییر دهیم نه در بخش کد نویسی چکار باید بکنیم. ممنون میشم اگه مطلب یا سایت و مقاله و راهنمایی نمایید. با تشکر

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۲/۲۷ ۱۲:۲۹

فیلتر Authorize را سفارشی سازی کنید .

نویسنده: مهدی عطار
تاریخ: ۱۳۹۴/۰۲/۲۷ ۲۲:۱۵

چطوری تو صفحه admin امکان این رو قرار بدیم که admin بتونه authorize واسه یه عملیات تعریف کنه تغییر بده حذف کنه و رولی رو به این authorize اضافه یا حذف کنه. با تشکر

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۲/۲۷ ۲۳:۳۰

سفارشی سازی فیلتر Authorize از ارث بری از AuthorizeAttribute و سپس override کردن متد

```
public override void OnAuthorization(AuthorizationContext filterContext)
```

آن شروع می‌شود. در اینجا به اطلاعاتی مانند

```
filterContext.ActionDescriptor.ControllerDescriptor.ControllerName
filterContext.ActionDescriptor.ActionName
```

و خیلی موارد دیگر (آدرس صفحه filterContext.HttpContext.Request.Url تا کاربر filterContext.HttpContext.User و غیره) دسترسی خواهید داشت.
سپس باید طراحی جدیدی را بر اساس ControllerName و ActionName پیاده سازی کنید (یک جدول جدید طراحی کنید) تا این

اکشن متدها یا کنترلرها امکان انتساب چندین Role متغیر را داشته باشند.
حالا زمانیکه این فیلتر Authorize سفارشی سازی شده بجای فیلتر Autorize اصلی استفاده می‌شود، نام اکشن متد و کنترلر جاری را از filterContext دریافت می‌کنید. سپس این دو مورد به همراه اطلاعات User جاری، پارامترهایی خواهند شد جهت کوئری گرفتن از بانک اطلاعاتی و جدولی که از آن صحبت شد.
در اینجا هر زمانیکه نیاز بود دسترسی کاربری را قطع کنید فقط کافی است نتیجه‌ی این فیلتر سفارشی را به نحو ذیل بازگردانید:

```
filterContext.Result = new HttpStatusCodeResult(403);
```

بنابراین در قسمت ادمین، یک صفحه‌ی جدید برای ثبت نام کنترلرها و اکشن متدها به همراه نقش‌های پویای آن‌ها خواهید داشت. سپس در این فیلتر Authorize سفارشی، دقیقا مشخص است که اکنون در کدام کنترلر و اکشن متد قرار داریم. بر این اساس (و سایر پارامترهایی که می‌توان از filterContext استخراج کرد) یک کوئری گرفته می‌شود و نقش‌های پویای فیلتر Authorize دریافت می‌شوند. نقش‌های کاربر جاری هم که مشخص هستند. این‌ها را با هم مقایسه می‌کنید و خروجی 403 را درصورت عدم تطابق، تنظیم خواهید کرد.
ضمنا در صفحه‌ی طراحی انتساب نقش‌های متغیر به اکشن متدها یا کنترلرها، امکان یافتن پویای لیست آن‌ها نیز وجود دارد.

عنوان: **افزودن تصدیق ایمیل به ASP.NET Identity در MVC 5**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۹ ۲۱:۴۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, ASP.NET MVC 5, ASP.NET Identity, MVC, Security

در پست قبلی نحوه سفارشی کردن پروفایل کاربران در ASP.NET Identity را مرور کردیم. اگر بیاد داشته باشید یک فیلد آدرس ایمیل به کلاس کاربر اضافه کردیم. در این پست از این فیلد استفاده میکنیم تا در پروسه ثبت نام ایمیل‌ها را تصدیق کنیم. بدین منظور پس از ثبت نام کاربران یک ایمیل فعالسازی برای آنها ارسال می‌کنیم که حاوی یک لینک است. کاربران با کلیک کردن روی این لینک پروسه ثبت نام خود را تایید می‌کنند و می‌توانند به سایت وارد شوند. پیش از تایید پروسه ثبت نام، کاربران قادر به ورود نیستند.

در ابتدا باید اطلاعات کلاس کاربر را تغییر دهید تا دو فیلد جدید را در بر گیرد. یک فیلد شناسه تایید (confirmation token) را ذخیره می‌کند، و دیگری فیلدی منطقی است که مشخص می‌کند پروسه ثبت نام تایید شده است یا خیر. پس کلاس *ApplicationUser* حالا باید بدین شکل باشد.

```
public class ApplicationUser : IdentityUser
{
    public string Email { get; set; }
    public string ConfirmationToken { get; set; }
    public bool IsConfirmed { get; set; }
}
```

اگر پیش از این کلاس ApplicationUser را تغییر داده اید، باید مهاجرت‌ها را فعال کنید و دیتابیس را بروز رسانی کنید. حالا می‌توانیم از این اطلاعات جدید در پروسه ثبت نام استفاده کنیم و برای کاربران ایمیل‌های تاییدیه را بفرستیم.

```
private string CreateConfirmationToken()
{
    return ShortGuid.NewGuid();
}

private void SendEmailConfirmation(string to, string username, string confirmationToken)
{
    dynamic email = new Email("RegEmail");
    email.To = to;
    email.UserName = username;
    email.ConfirmationToken = confirmationToken;
    email.Send();
}

//
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        string confirmationToken = CreateConfirmationToken();
        var user = new ApplicationUser()
        {
            UserName = model.UserName,
            Email = model.Email,
            ConfirmationToken = confirmationToken,
                IsConfirmed = false };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            SendEmailConfirmation(model.Email, model.UserName, confirmationToken);
            return RedirectToAction("RegisterStepTwo", "Account");
        }
        else
        {
            AddErrors(result);
        }
    }

    // If we got this far, something failed, redisplay form
```

```
    return View(model);
}
```

برای تولید شناسه‌های تایید (tokens) از کلاسی بنام *ShortGuid* استفاده شده است. این کلاس یک مقدار GUID را encode می‌کند که در نتیجه آن مقدار خروجی کوتاه‌تر بوده و برای استفاده در URLها ایمن است. کد این کلاس را از این وبلاگ گرفته ام. پس از ایجاد حساب کاربری باید شناسه تولید شده را به آن اضافه کنیم و مقدار فیلد IsConfirmed را به false تنظیم کنیم. برای تولید ایمیل‌ها من از Postal استفاده می‌کنم. Postal برای ساختن ایمیل‌های دینامیک شما از موتور Razor استفاده می‌کند. می‌توانید ایمیل‌های ساده (plain text) یا HTML بسازید، عکس و فایل در آن درج و ضمیمه کنید و امکانات بسیار خوب دیگر. اکشن متد RegisterStepTwo تنها کاربر را به یک View هدایت می‌کند که پیامی به او نشان داده می‌شود.

بعد از اینکه کاربر ایمیل را دریافت کرد و روی لینک تایید کلیک کرد به اکشن متد *RegisterConfirmation* باز می‌گردیم.

```
private bool ConfirmAccount(string confirmationToken)
{
    ApplicationDbContext context = new ApplicationDbContext();
    ApplicationUser user =  context.Users.SingleOrDefault(u => u.ConfirmationToken ==
confirmationToken);
    if (user != null)
    {
        user.IsConfirmed = true;
        DbSet<ApplicationUser> dbSet = context.Set<ApplicationUser>();
        dbSet.Attach(user);
        context.Entry(user).State = EntityState.Modified;
        context.SaveChanges();

        return true;
    }
    return false;
}

[AllowAnonymous]
public ActionResult RegisterConfirmation(string Id)
{
    if (ConfirmAccount(Id))
    {
        return RedirectToAction("ConfirmationSuccess");
    }
    return RedirectToAction("ConfirmationFailure");
}
```

متد *ConfirmAccount* سعی می‌کند کاربری را در دیتابیس پیدا کند که شناسه تاییدش با مقدار دریافت شده از URL برابر است. اگر این کاربر پیدا شود، مقدار خاصیت IsConfirmed را به true تغییر می‌دهیم و همین مقدار را به تابع باز می‌گردانیم. در غیر اینصورت false بر می‌گردانیم. اگر کاربر تایید شده است، می‌تواند به سایت وارد شود. برای اینکه مطمئن شویم کاربران پیش از تایید ایمیل شان نمی‌توانند وارد سایت شوند، باید اکشن متد *Login* را کمی تغییر دهیم.

```
[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 && user.IsConfirmed)
        {
            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);
}
```

تنها کاری که می‌کنیم این است که به دنبال کاربری می‌گردیم که فیلد IsConfirmed آن true باشد. اگر مقدار این فیلد false باشد کاربر را به سایت وارد نمی‌کنیم و پیغام خطایی نمایش می‌دهیم.
همین. این تمام چیزی بود که برای اضافه کردن تصدیق ایمیل به اپلیکیشن خود نیاز دارید. از آنجا که سیستم ASP.NET Identity با Entity Framework مدیریت می‌شود و با مدل Code First ساخته شده، سفارشی کردن اطلاعات کاربران و سیستم عضویت ساده‌تر از همیشه است.

توضیحاتی درباره کار با Postal

اگر به متد SendEmailConfirmation دقت کنید خواهید دید که آبجکتی از نوع Email می‌سازیم (که در اسمبلی‌های Postal وجود دارد) و از آن برای ارسال ایمیل استفاده می‌کنیم. عبارت "RegEmail" نام نمایی است که باید برای ساخت ایمیل استفاده شود. این متغیر از نوع dynamic است، مانند خاصیت ViewBag. بدین معنا که می‌توانید مقادیر مورد نظر خود را بصورت خواص دینامیک روی این آبجکت تعریف کنید. از آنجا که Postal از موتور Razor استفاده می‌کند، بعدا در View ایمیل خود می‌توانید به این مقادیر دسترسی داشته باشید.
در پوشه Views پوشه جدیدی بنام Emails بسازید. سپس یک فایل جدید با نام RegEmail.cshtml در آن ایجاد کنید. کد این فایل را با لیست زیر جایگزین کنید.

```
To: @ViewBag.To
From: YOURNAME@gmail.com
Subject: Confirm your registration

Hello @ViewBag.UserName,
Please confirm your registration by following the link bellow.

@Html.ActionLink(Url.Action("RegisterConfirmation", "Account", new { id = @ViewBag.ConfirmationToken
}), "RegisterConfirmation", "Account", new { id = @ViewBag.ConfirmationToken }, null)
```

این فایل، قالب ایمیل‌های شما خواهد بود. ایمیل‌ها در حال حاظر بصورت plain text ارسال می‌شوند. برای اطلاعات بیشتر درباره ایمیل‌های HTML و امکانات پیشرفته‌تر به سایت پروژه Postal مراجعه کنید.
همانطور که مشاهده می‌کنید در این نما همان خاصیت‌های دینامیک تعریف شده را فراخوانی می‌کنیم تا مقادیر لازم را بدست آوریم.

To. ViewBag آدرس ایمیل گیرنده را نشان می‌دهد.
UserName. ViewBag نام کاربر جاری را نمایش می‌دهد.
ConfirmationToken. ViewBag شناسه تولید شده برای تایید کاربر است.

در این قالب لینکی به متد RegisterConfirmation در کنترلر Account وجود دارد که شناسه تایید را نیز با پارامتری بنام id انتقال می‌دهد.
یک فایل _ViewStart.cshtml هم در این پوشه بسازید و کد آن را با لیست زیر جایگزین کنید.

```
@{ Layout = null; /* Overrides the Layout set for regular page views. */ }
```

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۰/۱۹ ۲۲:۱۴

با تشکر از شما. لطفا View ایمیل ارسالی را (متن حاوی لینک) که توسط کتابخانه Postal پردازش می‌شود، نیز ارسال نمائید. چون الان به نظر متد SendEmailConfirmation مشخص نیست چه متنی را ارسال می‌کند و چطور آن متن را دریافت می‌کند.

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۱۹ ۲۳:۷

با تشکر از شما، پست بروز رسانی شد.

نویسنده: کاربر
تاریخ: ۱۳۹۲/۱۱/۰۵ ۱۶:۵۲

با تشکر لطفا تنظیمات smtp مربوطه رو هم قرار بدید
سایت سازنده تنظیمات رو در وب کانفیگ قرار داده بود، برای جیمیل من تنظیمات زیر رو مینویسم ولی خطای timed out میده

```
<system.net>
<mailSettings>
<smtp >
  <network host="smtp.gmail.com" userName="My Gmail Email"
  password="my Password" enableSsl="true" port="465" defaultCredentials="true"/>
</smtp>
</mailSettings>
  </system.net>
```

لطفا راهنمایی بفرمایید با تشکر

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۵ ۱۹:۵۲

```
<system.net>
  <mailSettings>
    <smtp deliveryMethod="Network" from="armin.zia@gmail.com">
      <network host="smtp.gmail.com" port="587" defaultCredentials="false" enableSsl="true"
userName="YOUR-EMAIL" password="YOUR-PASSWORD" />
    </smtp>
  </mailSettings>
</system.net>
```

نویسنده: داود
تاریخ: ۱۳۹۲/۱۲/۰۵ ۰:۲۷

اگر بخوایم کاربر در فرم لاگین هم با username و هم با email که تأیید شده وارد بشه باید چه کار کنیم؟

نویسنده: Mr.J
تاریخ: ۱۳۹۳/۰۲/۰۱ ۱۵:۱۳

سلام
من دقیقا طبق دستورات بالا کدهام رو نوشتم اما این خطارو میگیره...

```
The SMTP host was not specified.
```

و در قسمت web.config اصلی سایت هم این کدهارو اضافه کردم

```
<system.net>
<mailSettings>
<smtp from="my_gmail">
  <network host="smtp.gmail.com" port="587" defaultCredentials="false" enableSsl="true"
userName="my_gmail" password="mypassword" />
</smtp>
</mailSettings>
  </system.net>
```

مشکل از کجاست.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۰۱ ۱۷:۱۲

نباید مشکلی باشد. مگر اینکه محل قرارگیری تنظیمات system.net شما توسط برنامه قابل یافت شدن نباشد. مثلا آن‌را داخل system.web قرار داده باشید یا مکان دیگری. system.net یک مدخل مجزا و مستقل است. همچنین اگر سایت چندین وب کانفیگ دارد (مانند برنامه‌های ASP.NET MVC)، وب کانفیگ موجود در ریشه سایت باید تنظیم شود و نه مورد موجود در پوشه‌ی Views برنامه.

نویسنده: ذهن خوان
تاریخ: ۱۳۹۴/۰۶/۰۹ ۱۵:۵۲

سلام؛ در قسمت RegEmail.cshtml خط آخر دارای ارور هنگام اجرا هستم.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۰۹ ۱۶:۳۸

- ابتدا بررسی کنید مقادیر viewbagهای دریافتی در این View نال هستند یا خیر؟
- سپس تعریف ActionLink موجود در متن فوق را به صورت زیر اصلاح کنید (قسمت آخر آن در متن، دوبار تکرار شده‌است):

```
Html.ActionLink(".....Click Here for Confirmation.....",
                "RegisterConfirmation", "Account", new { id = @ViewBag.ConfirmationToken }, null)
```

نویسنده: ذهن خوان
تاریخ: ۱۳۹۴/۰۶/۰۹ ۱۹:۵۷

بسیار متشکرم بابت پاسخ، همه چی عالیه جز اینکه متد send پیغام timeout می‌دهد و تمامی پورت‌های SMTP رو چک کردم. به نظر شما امکانش هست مشکل از Portal باشه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۰۹ ۲۰:۳۷

Portal بیشتر یک واسطه و یک لایه مخفی ساز کار با مباحث ارسال ایمیل است. این موارد را بهتر است بررسی کنید:

« چگونه بررسی کنیم smtp server مورد نظر ما کار می‌کند؟ »

« تنظیمات امنیتی SMTP Server متعلق به IIS 6.0 جهت قرارگیری بر روی اینترنت »

برای آزمایش‌های محلی:

« شبیه سازی ارسال ایمیل در ASP.Net »

« شبیه ساز میل سرور برای برنامه نویس‌ها »

عنوان: **مهاجرت از SQL Membership به ASP.NET Identity**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۲۱ ۱۸:۳۰
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, Security, MemberShip, ASP.NET Identity

در این مقاله مهاجرت یک اپلیکیشن وب که توسط SQL Membership ساخته شده است را به سیستم جدید ASP.NET Identity بررسی می‌کنیم. برای این مقاله از یک قالب اپلیکیشن وب (Web Forms) که توسط Visual Studio 2010 ساخته شده است برای ساختن کاربران و نقش‌ها استفاده می‌کنیم. سپس با استفاده از یک SQL Script دیتابیس موجود را به دیتابیسی که ASP.NET Identity نیاز دارد تبدیل می‌کنیم. در قدم بعدی پکیج‌های مورد نیاز را به پروژه اضافه می‌کنیم و صفحات جدیدی برای مدیریت حساب‌های کاربری خواهیم ساخت. بعنوان یک تست، کاربران قدیمی که توسط SQL Membership ساخته شده بودند باید قادر باشند به سایت وارد شوند. همچنین کاربران جدید باید بتوانند بدون هیچ مشکلی در سیستم ثبت نام کنند. سورس کد کامل این مقاله را می‌توانید از این لینک دریافت کنید.

یک اپلیکیشن با SQL Membership بسازید

برای شروع به اپلیکیشنی نیاز داریم که از SQL Membership استفاده می‌کند و دارای داده هایی از کاربران و نقش‌ها است. برای این مقاله، بگذارید پروژه جدیدی توسط VS 2010 بسازیم.

حال با استفاده از ابزار ASP.NET Configuration دو کاربر جدید بسازید: **oldAdminUser** و **oldUser** .

Project
Build
Debug
Team
XML
Data
Tools
Architecture
Test
Add Class...
Add New Item...
Ctrl+Shift+A
Add Existing Item...
Shift+Alt+A
Add ASP.NET Folder
Build Deployment Package
Package/Publish Settings
Exclude From Project
Show All Files
Add Reference...
Add Service Reference...
Set as StartUp Project
Use IIS Express...
Add Deployable Dependencies...
Manage NuGet Packages...
Enable NuGet Package Restore
Refresh Project Toolbox Items
SQLMembership-Identity-OWIN Properties...
ASP.NET Configuration

نقش جدیدی با نام Admin بسازید و کاربر oldAdminUser را به آن اضافه کنید.

بخش جدیدی با نام Admin در سایت خود بسازید و فرمی بنام Default.aspx به آن اضافه کنید. همچنین فایل web.config این قسمت را طوری پیکربندی کنید تا تنها کاربرانی که در نقش Admin هستند به آن دسترسی داشته باشند. برای اطلاعات بیشتر به این لینک مراجعه کنید.

پنجره Server Explorer را باز کنید و جداول ساخته شده توسط SQL Membership را بررسی کنید. اطلاعات اصلی کاربران که برای ورود به سایت استفاده می‌شوند، در جداول **aspnet_Users** و **aspnet_Membership** ذخیره می‌شوند. داده‌های مربوط به نقش‌ها نیز در جدول **aspnet_Roles** ذخیره خواهند شد. رابطه بین کاربران و نقش‌ها نیز در جدول **aspnet_UsersInRoles** ذخیره می‌شود، یعنی اینکه هر کاربری به چه نقش هایی تعلق دارد.

برای مدیریت اساسی سیستم عضویت، مهاجرت جداول ذکر شده به سیستم جدید ASP.NET Identity کفایت می‌کند.

مهاجرت به Visual Studio 2013

برای شروع ابتدا Visual Studio Express 2013 for Web یا Visual Studio 2013 را نصب کنید.

حال پروژه ایجاد شده را در نسخه جدید ویژوال استودیو باز کنید. اگر نسخه ای از SQL Server Express را روی سیستم خود نصب نکرده باشید، هنگام باز کردن پروژه پیغامی به شما نشان داده می‌شود. دلیل آن وجود رشته اتصالی است که از SQL Server Express استفاده می‌کند. برای رفع این مساله می‌توانید SQL Express را نصب کنید، و یا رشته اتصال را طوری تغییر دهید که از LocalDB استفاده کند.

فایل web.config را باز کرده و رشته اتصال را مانند تصویر زیر ویرایش کنید.

```
<configuration>
  <connectionStrings>
    <add name="ApplicationServices"
         connectionString="data source=(LocalDb)\v11.0;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\aspnetdb.mdf
         providerName="System.Data.SqlClient" />
  </connectionStrings>
```

پنجره Server Explorer را باز کنید و مطمئن شوید که الگوی جداول و داده‌ها قابل رویت هستند.
سیستم ASP.NET Identity با نسخه 4.5 دات نت فریم ورک و بالاتر سازگار است. پس نسخه فریم ورک پروژه را به آخرین نسخه (4.5.1) تغییر دهید.

پروژه را Build کنید تا مطمئن شوید هیچ خطایی وجود ندارد.

نصب پکیج‌های NuGet

در پنجره Solution Explorer روی نام پروژه خود کلیک راست کرده، و گزینه Manage NuGet Packages را انتخاب کنید. در قسمت جستجوی دیالوگ باز شده، عبارت "Microsoft.AspNet.Identity.EntityFramework" را وارد کنید. این پکیج را در لیست نتایج انتخاب کرده و آن را نصب کنید. نصب این بسته، نیازمندهای موجود را بصورت خودکار دانلود و نصب می‌کند:

EntityFramework و **ASP.NET Identity Core** . حال پکیج‌های زیر را هم نصب کنید (اگر نمی‌خواهید OAuth را فعال کنید، 4 پکیج آخر را نادیده بگیرید).

Microsoft.AspNet.Identity.Owin

Microsoft.Owin.Host.SystemWeb

Microsoft.Owin.Security.Facebook

Microsoft.Owin.Security.Google

Microsoft.Owin.Security.MicrosoftAccount

Microsoft.Owin.Security.Twitter

مهاجرت دیتابیس فعلی به سیستم ASP.NET Identity

قدم بعدی مهاجرت دیتابیس فعلی به الگویی است، که سیستم ASP.NET Identity به آن نیاز دارد. بدین منظور ما یک اسکریپت SQL را اجرا می‌کنیم تا جداول جدیدی بسازد و اطلاعات کاربران را به آنها انتقال دهد. فایل این اسکریپت را می‌توانید از لینک https://github.com/suhasj/SQLMembership-Identity-OWIN دریافت کنید.

این اسکریپت مختص این مقاله است. اگر الگوی استفاده شده برای جداول سیستم عضویت شما ویرایش/سفارشی-سازی شده باید این اسکریپت را هم بر اساس این تغییرات بروز رسانی کنید.

پنجره Server Explorer را باز کنید. گره اتصال ApplicationServices را باز کنید تا جداول را مشاهده کنید. روی گره Tables کلیک راست کرده و گزینه New Query را انتخاب کنید.

در پنجره کوئری باز شده، تمام محتویات فایل *Migrations.sql* را کپی کنید. سپس اسکریپت را با کلیک کردن دکمه Execute اجرا کنید.

```sql
DROP TABLE AspNetUserRoles;

DROP TABLE AspNetUserClaims;

DROP TABLE AspNetUserLogins;

DROP TABLE AspNetRoles;

DROP TABLE AspNetUsers;

CREATE TABLE [dbo].[AspNetUsers] (
    [Id]                    NVARCHAR (128) NOT NULL,
    [UserName]              NVARCHAR (MAX) NULL,
    [PasswordHash]          NVARCHAR (MAX) NULL,
    [SecurityStamp]         NVARCHAR (MAX) NULL,
    [Discriminator]         NVARCHAR (128) NOT NULL,
    [ApplicationId]                        UNIQUEIDENTIFIER NOT NULL,
    [PasswordFormat]                       INT              DEFAULT ((0)) NULL,
    [PasswordSalt]                         NVARCHAR (128)   NULL,
    [LegacyPasswordHash]  NVARCHAR (MAX) NULL,
    [LoweredUserName]  NVARCHAR (256)   NOT NULL,
    [MobileAlias]      NVARCHAR (16)    DEFAULT (NULL) NULL,
    [IsAnonymous]      BIT              DEFAULT ((0)) NOT NULL,
    [LastActivityDate] DATETIME         NOT NULL,
    [MobilePIN]                            NVARCHAR (16)    NULL,
    [Email]                                NVARCHAR (256)   NULL,
    [LoweredEmail]                         NVARCHAR (256)   NULL,
    [PasswordQuestion]                     NVARCHAR (256)   NULL,
    [PasswordAnswer]                       NVARCHAR (128)   NULL,
    [IsApproved]                           BIT              NOT NULL,
    [IsLockedOut]                          BIT              NOT NULL,
    [CreateDate]                           DATETIME         NOT NULL,
    [LastLoginDate]                        DATETIME         NOT NULL,
    [LastPasswordChangedDate]              DATETIME         NOT NULL,
    [LastLockoutDate]                      DATETIME         NOT NULL,
    [FailedPasswordAttemptCount]           INT              NOT NULL,
    [FailedPasswordAttemptWindowStart]     DATETIME         NOT NULL,
    [FailedPasswordAnswerAttemptCount]     INT              NOT NULL,
```

ممکن است با اخطاری مواجه شوید مبنی بر آنکه امکان حذف (drop) بعضی از جداول وجود نداشت. دلیلش آن است که چهار عبارت اولیه در این اسکریپت، تمام جداول مربوط به Identity را در صورت وجود حذف می‌کنند. از آنجا که با اجرای اولیه این اسکریپت چنین جداولی وجود ندارند، می‌توانیم این خطاها را نادیده بگیریم. حال پنجره Server Explorer را تازه (refresh) کنید و خواهید دید که پنج جدول جدید ساخته شده اند.

لیست زیر نحوه Map کردن اطلاعات از جداول SQL Membership به سیستم Identity را نشان می‌دهد.

aspnet_Roles --> **AspNetRoles**

aspnet_Users, aspnet_Membership --> **AspNetUsers**

aspnet_UsersInRoles --> **AspNetUserRoles**

جداول AspNetUserClaims و AspNetUserLogins خالی هستند. فیلد تفکیک کننده (Discriminator) در جدول AspNetUsers باید مطابق نام کلاس مدل باشد، که در مرحله بعدی تعریف خواهد شد. همچنین ستون PasswordHash به فرم 'encrypted password salt|password format|password' می‌باشد. این شما را قادر می‌سازد تا از رمزنگاری برای ذخیره و بازیابی کلمه‌های عبور استفاده کنید. این مورد نیز در ادامه مقاله بررسی شده است.

ساختن مدل‌ها و صفحات عضویت

بصورت پیش فرض سیستم ASP.NET Identity برای دریافت و ذخیره اطلاعات در دیتابیس عضویت از Entity Framework استفاده می‌کند. برای آنکه بتوانیم با جداول موجود کار کنیم، می‌بایست ابتدا مدل هایی که الگوی دیتابیس را نمایندگی می‌کنند ایجاد کنیم. برای این کار مدل‌های ما یا باید اینترفیس‌های موجود در Identity.Core را پیاده سازی کنند، یا می‌توانند پیاده سازی‌های پیش فرض را توسعه دهند. پیاده سازی‌های پیش فرض در Microsoft.AspNet.Identity.EntityFramework وجود دارند.

در نمونه ما، جداول AspNetRoles, AspNetUserClaims, AspNetLogins و AspNetUserRole ستون هایی دارند که شباهت زیادی به پیاده سازی‌های پیش فرض سیستم Identity دارند. در نتیجه می‌توانیم از کلاس‌های موجود، برای Map کردن الگوی جدید استفاده کنیم. جدول AspNetUsers ستون‌های جدیدی نیز دارد. می‌توانیم کلاس جدیدی بسازیم که از **IdentityUser** ارث بری کند و آن را گسترش دهیم تا این فیلدهای جدید را پوشش دهد.

پوشه ای با نام Models بسازید (در صورتی که وجود ندارد) و کلاسی با نام User به آن اضافه کنید.

کلاس User باید کلاس IdentityUser را که در اسمبلی Microsoft.AspNet.Identity.EntityFramework وجود دارد گسترش دهد. خاصیت هایی را تعریف کنید که نماینده الگوی جدول AspNetUser هستند. خواص ID, Username, PasswordHash و SecurityStamp در کلاس IdentityUser تعریف شده اند، بنابراین این خواص را در لیست زیر نمی‌بینید.

```
public class User : IdentityUser
    {
        public User()
        {
            CreateDate = DateTime.Now;
            IsApproved = false;
            LastLoginDate = DateTime.Now;
            LastActivityDate = DateTime.Now;
            LastPasswordChangedDate = DateTime.Now;
            LastLockoutDate = DateTime.Parse("1/1/1754");
            FailedPasswordAnswerAttemptWindowStart = DateTime.Parse("1/1/1754");
            FailedPasswordAttemptWindowStart = DateTime.Parse("1/1/1754");
        }

        public System.Guid ApplicationId { get; set; }
        public string MobileAlias { get; set; }
        public bool IsAnonymous { get; set; }
        public System.DateTime LastActivityDate { get; set; }
        public string MobilePIN { get; set; }
        public string Email { get; set; }
        public string LoweredEmail { get; set; }
        public string LoweredUserName { get; set; }
        public string PasswordQuestion { get; set; }
        public string PasswordAnswer { get; set; }
        public bool IsApproved { get; set; }
        public bool IsLockedOut { get; set; }
        public System.DateTime CreateDate { get; set; }
        public System.DateTime LastLoginDate { get; set; }
        public System.DateTime LastPasswordChangedDate { get; set; }
        public System.DateTime LastLockoutDate { get; set; }
        public int FailedPasswordAttemptCount { get; set; }
        public System.DateTime FailedPasswordAttemptWindowStart { get; set; }
        public int FailedPasswordAnswerAttemptCount { get; set; }
        public System.DateTime FailedPasswordAnswerAttemptWindowStart { get; set; }
        public string Comment { get; set; }
    }
```

حال برای دسترسی به دیتابیس مورد نظر، نیاز به یک DbContext داریم. اسمبلی Microsoft.AspNet.Identity.EntityFramework کلاسی با نام **IdentityDbContext** دارد که پیاده سازی پیش فرض برای دسترسی به دیتابیس ASP.NET Identity است. نکته قابل توجه این است که IdentityDbContext آبجکتی از نوع **TUser** را می‌پذیرد. TUser می‌تواند هر کلاسی باشد که از IdentityUser ارث بری کرده و آن را گسترش می‌دهد.

در پوشه Models کلاس جدیدی با نام **ApplicationDbContext** بسازید که از IdentityDbContext ارث بری کرده و از کلاس

User استفاده می‌کند.

```
public class ApplicationDbContext : IdentityDbContext<User>
{

}
```

مدیریت کاربران در ASP.NET Identity توسط کلاسی با نام UserManager انجام می‌شود که در اسمبلی Microsoft.AspNet.Identity.EntityFramework قرار دارد. چیزی که ما در این مرحله نیاز داریم، کلاسی است که از UserManager ارث بری می‌کند و آن را طوری توسعه می‌دهد که از کلاس User استفاده کند.

در پوشه Models کلاس جدیدی با نام UserManager بسازید.

```
public class UserManager : UserManager<User>
{

}
```

کلمه عبور کاربران بصورت رمز نگاری شده در دیتابیس ذخیره می‌شوند. الگوریتم رمز نگاری SQL Membership با سیستم ASP.NET Identity تفاوت دارد. هنگامی که کاربران قدیمی به سایت وارد می‌شوند، کلمه عبورشان را توسط الگوریتم‌های قدیمی SQL Membership رمزگشایی می‌کنیم، اما کاربران جدید از الگوریتم‌های ASP.NET Identity استفاده خواهند کرد.

کلاس UserManager خاصیتی با نام **PasswordHasher** دارد. این خاصیت نمونه ای از یک کلاس را ذخیره می‌کند، که اینترفیس IPasswordHasher را پیاده سازی کرده است. این کلاس هنگام تراکنش‌های احراز هویت کاربران استفاده می‌شود تا کلمه‌های عبور را رمزنگاری/رمزگشایی شوند. در کلاس UserManager کلاس جدیدی بنام SQLPasswordHasher بسازید. کد کامل را در لیست زیر مشاهده می‌کنید.

```
public class SQLPasswordHasher : PasswordHasher
{
        public override string HashPassword(string password)
        {
            return base.HashPassword(password);
        }

        public override PasswordVerificationResult VerifyHashedPassword(string hashedPassword, string
providedPassword)
        {
            string[] passwordProperties = hashedPassword.Split('|');
            if (passwordProperties.Length != 3)
            {
                return base.VerifyHashedPassword(hashedPassword, providedPassword);
            }
            else
            {
                string passwordHash = passwordProperties[0];
                int passwordformat = 1;
                string salt = passwordProperties[2];
                if (String.Equals(EncryptPassword(providedPassword, passwordformat, salt),
passwordHash, StringComparison.CurrentCultureIgnoreCase))
                {
                    return PasswordVerificationResult.SuccessRehashNeeded;
                }
                else
                {
                    return PasswordVerificationResult.Failed;
                }
            }
        }

        //This is copied from the existing SQL providers and is provided only for back-compat.
        private string EncryptPassword(string pass, int passwordFormat, string salt)
        {
            if (passwordFormat == 0) // MembershipPasswordFormat.Clear
                return pass;

            byte[] bIn = Encoding.Unicode.GetBytes(pass);
```

```
            byte[] bSalt = Convert.FromBase64String(salt);
            byte[] bRet = null;

            if (passwordFormat == 1)
            { // MembershipPasswordFormat.Hashed
                HashAlgorithm hm = HashAlgorithm.Create("SHA1");
                if (hm is KeyedHashAlgorithm)
                {
                    KeyedHashAlgorithm kha = (KeyedHashAlgorithm)hm;
                    if (kha.Key.Length == bSalt.Length)
                    {
                        kha.Key = bSalt;
                    }
                    else if (kha.Key.Length < bSalt.Length)
                    {
                        byte[] bKey = new byte[kha.Key.Length];
                        Buffer.BlockCopy(bSalt, 0, bKey, 0, bKey.Length);
                        kha.Key = bKey;
                    }
                    else
                    {
                        byte[] bKey = new byte[kha.Key.Length];
                        for (int iter = 0; iter < bKey.Length; )
                        {
                            int len = Math.Min(bSalt.Length, bKey.Length - iter);
                            Buffer.BlockCopy(bSalt, 0, bKey, iter, len);
                            iter += len;
                        }
                        kha.Key = bKey;
                    }
                    bRet = kha.ComputeHash(bIn);
                }
                else
                {
                    byte[] bAll = new byte[bSalt.Length + bIn.Length];
                    Buffer.BlockCopy(bSalt, 0, bAll, 0, bSalt.Length);
                    Buffer.BlockCopy(bIn, 0, bAll, bSalt.Length, bIn.Length);
                    bRet = hm.ComputeHash(bAll);
                }
            }

            return Convert.ToBase64String(bRet);
    }
}
```

دقت کنید تا فضاهای نام System.Text و System.Security.Cryptography را وارد کرده باشید.

متد EncodePassword کلمه عبور را بر اساس پیاده سازی پیش فرض SQL Membership رمزنگاری می‌کند. این الگوریتم از System.Web گرفته می‌شود. اگر اپلیکیشن قدیمی شما از الگوریتم خاصی استفاده می‌کرده است، همینجا باید آن را منعکس کنید. دو متد دیگر نیز بنام‌های *HashPassword* و *VerifyHashedPassword* نیاز داریم. این متدها از EncodePassword برای رمزنگاری کلمه‌های عبور و تایید آنها در دیتابیس استفاده می‌کنند.

سیستم SQL Membership برای رمزنگاری (Hash) کلمه‌های عبور هنگام ثبت نام و تغییر آنها توسط کاربران، از PasswordHash, PasswordSalt و PasswordFormat استفاده می‌کرد. در روند مهاجرت، این سه فیلد در ستون PasswordHash جدول AspNetUsers ذخیره شده و با کاراکتر '|' جدا شده اند. هنگام ورود کاربری به سایت، اگر کله عبور شامل این فیلدها باشد از الگوریتم SQL Membership برای بررسی آن استفاده می‌کنیم. در غیر اینصورت از پیاده سازی پیش فرض ASP.NET Identity استفاده خواهد شد. با این روش، کاربران قدیمی لازم نیست کلمه‌های عبور خود را صرفا بدلیل مهاجرت اپلیکیشن ما تغییر دهند.

کلاس UserManager را مانند قطعه کد زیر بروز رسانی کنید.

```
public UserManager()
            : base(new UserStore<User>(new ApplicationDbContext()))
        {
            this.PasswordHasher = new SQLPasswordHasher();
 }
```

ایجاد صفحات جدید مدیریت کاربران

قدم بعدی ایجاد صفحاتی است که به کاربران اجازه ثبت نام و ورود را می‌دهند. صفحات قدیمی SQL Membership از کنترل هایی استفاده می‌کنند که با ASP.NET Identity سازگار نیستند. برای ساختن این صفحات جدید به این مقاله مراجعه کنید. از آنجا که در این مقاله پروژه جدید را ساخته ایم و پکیج‌های لازم را هم نصب کرده ایم، می‌توانید مستقیما به قسمت Adding Web Forms for registering users to your application بروید.

چند تغییر که باید اعمال شوند:

فایل‌های Register.aspx.cs و Login.aspx.cs از کلاس UserManager استفاده می‌کنند. این ارجاعات را با کلاس UserManager جدیدی که در پوشه Models ساختید جایگزین کنید.

همچنین ارجاعات استفاده از کلاس IdentityUser را به کلاس User که در پوشه Models ساختید تغییر دهید.

لازم است توسعه دهنده مقدار ApplicationId را برای کاربران جدید طوری تنظیم کند که با شناسه اپلیکیشن جاری تطابق داشته باشد. برای این کار می‌توانید پیش از ساختن حساب‌های کاربری جدید در فایل Register.aspx.cs ابتدا شناسه اپلیکیشن را بدست آورید و اطلاعات کاربر را بدرستی تنظیم کنید.

مثال: در فایل Register.aspx.cs متد جدیدی تعریف کنید که جدول aspnet_Applications را بررسی میکند و شناسه اپلیکیشن را بر اساس نام اپلیکیشن بدست می‌آورد.

```
private Guid GetApplicationID()
{
    using (SqlConnection connection = new
SqlConnection(ConfigurationManager.ConnectionStrings["ApplicationServices"].ConnectionString))
            {
                string queryString = "SELECT ApplicationId from aspnet_Applications WHERE
ApplicationName = '/'"; //Set application name as in database

                SqlCommand command = new SqlCommand(queryString, connection);
                command.Connection.Open();

                var reader = command.ExecuteReader();
                while (reader.Read())
                {
                    return reader.GetGuid(0);
                }

            return Guid.NewGuid();
        }
}
```

حال می‌توانید این مقدار را برای آبجکت کاربر تنظیم کنید.

```
var currentApplicationId = GetApplicationID();

User user = new User() { UserName = Username.Text,
ApplicationId=currentApplicationId, …};
```

در این مرحله می‌توانید با استفاده از اطلاعات پیشین وارد سایت شوید، یا کاربران جدیدی ثبت کنید. همچنین اطمینان حاصل کنید که کاربران پیشین در نقش‌های مورد نظر وجود دارند.

مهاجرت به ASP.NET Identity مزایا و قابلیت‌های جدیدی را به شما ارائه می‌کند. مثلا کاربران می‌توانند با استفاده از تامین کنندگان ثالثی مثل Facebook, Google, Microsoft, Twitter و غیره به سایت وارد شوند. اگر به سورس کد این مقاله مراجعه کنید خواهید دید که امکانات OAuth نیز فعال شده اند.

در این مقاله انتقال داده‌های پروفایل کاربران بررسی نشد. اما با استفاده از نکات ذکر شده می‌توانید پروفایل کاربران را هم بسادگی منتقل کنید. کافی است مدل‌های لازم را در پروژه خود تعریف کرده و با استفاده از اسکریپت‌های SQL داده‌ها را انتقال دهید.

نظرات خوانندگان

نویسنده: سمیرا قادری
تاریخ: ۱۳۹۲/۱۰/۲۳ ۱۳:۳۳

سلام ممنون از مطالب خوبتون .
زمانیکه در اپلیکیشن خودم asp.net Configuration و تب Security را انتخاب می‌نمایم با پیغام زیر مواجه می‌شوم Unable to connect to SQL Server database.
دستور زیر را هم اجرا کردم
C:\Windows\Microsoft.NET\Framework\v4.0.30319\aspnet_regsql.exe -S . -A all -d tt -U sa_tt -P asd123
در صورتیکه از login و password آن مطمئن هستم ولی مشکل حل نشد
در صورتیکه مشکل از permission هست چگونه باید آن را حل کنم
webconfig برنامه به شرح زیر است
Data Source=.;Initial Catalog=tt;User ID=sa_tt;Password=asd123
با تشکر

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۰/۲۳ ۱۴:۴۵

تنظیمات وب کانفیگ خودتون رو با نمونه MSDN برای تعریف membership مطابقت بدید. جزئیات قسمت‌های مختلف آن

نویسنده: حمید
تاریخ: ۱۳۹۲/۱۰/۲۳ ۲۱:۱۰

خیلی ممنون آموزش‌های مفیدتون!
فرض کنید ما یک سیستم Authentication مثل Membership داریم با 2 تا نقش. حالا می‌خواهیم نقش A به فرم F1 و بعضی از متدهاش دسترسی داشته و در فرم F2، نقش B به آن دسترسی داشته باشه. خوب تا اینجا فکر کنم پیاده سازیش راحت باشه. ولی مساله از جایی پیچیده میشه که ما بخواهیم در زمان اجرا این تنظیمات رو عوض کنیم مثلا: در فرم F1، نقش A دسترسیش به متدها و حتی به کل فرم تغییر کنه و یا اجازه دسترسی به فرم B بهش داده بشه و این شرایط رو برای چندین نقش و یا در سطح کاربر داشت توصیه چیه؟ از چه تکنولوژی و یا راهکاری میشه به این مقصود رسید.
خیلی ممنون

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۰/۲۳ ۲۱:۳۷

پایه‌اش همین مسایل هست. فقط قسمتی که کار Authorization انجام میشه رو می‌توان سفارشی کرد. مثلا:

Dynamic Controller/Action Authorization in ASP.NET MVC
MVC Dynamic Authorization
ASP.NET MVC Custom Authorize Attribute with Roles Parser

نویسنده: کامران سادین
تاریخ: ۱۳۹۲/۱۱/۰۸ ۲:۱۳

ممنون از مطلب خوبتون.
من جداول خودم رو برای احراز هویت دارم آیا میشه همین جداول رو با ASP.NET Identity کار کنم؟
اگر نمیشه، این بانک ASP.NET Identity که توی SQL Server نمیسازه بانک Database.mdf رو میسازه، میشه توی SQL بسازیم از

همون ابتدای پروژه؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۰۸ ۹:۲۰

سیستم کارش EF Code first هست. این سیستم کدهاش گره خورده به بانک اطلاعاتی خاصی نیست. الان در این مثال رشته اتصالی به یک localdb اشاره می‌کنه. شما می‌تونید کلا این رشته و نحوه‌ی تعریف اون رو برای کار با SQL Server یا SQL CE یا هر بانک اطلاعاتی دیگری که پروایدر code first داره، تغییر بدید و استفاده کنید. (و اگر با ef code first آشنایی ندارید، کم کم در آینده نمی‌تونید با کتابخانه‌های کمکی و جانبی دات نت کار کنید)

نویسنده: کامران سادین
تاریخ: ۱۳۹۲/۱۱/۰۸ ۱۰:۲۷

مقاله ای نیز راجع به تغییر فیلدهاش هم بنویسید بد نیست دوستان علاقه دارند. با تشکر.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۰۸ ۱۰:۵۸

« سفارشی کردن ASP.NET Identity در MVC 5 » بیشتر در اینجا

نویسنده: مهدی
تاریخ: ۱۳۹۳/۰۱/۱۹ ۱۲:۳۴

دوست عزیز برای حل این مشکل شما باید پوشه سورس برنامه رو تغییر بدین. این مشکل به دلیل وجود خط فاصله تو آدرس (دایرکتوری) محل قرار گیری پروژه هست. اگه محل پروژه رو در جایی قرار دهید که در آدرس آن از فاصله استفاده نشده باشد به خوبی کار خواهد کرد.

عنوان: **مهاجرت داده عضویت و پروفایل از Universal Providers به ASP.NET Identity**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۲۲ ۶:۵۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, MemberShip, Identity, ASP.NET Identity, Universal Providers

در این مقاله مهاجرت داده‌های سیستم عضویت، نقش‌ها و پروفایل‌های کاربران که توسط Universal Providers ساخته شده اند به مدل ASP.NET Identity را بررسی می‌کنیم. رویکردی که در این مقاله استفاده شده و قدم‌های لازمی که توضیح داده شده اند، برای اپلیکیشنی که با SQL Membership کار می‌کند هم می‌توانند کارساز باشند.
با انتشار Visual Studio 2013، تیم ASP.NET سیستم جدیدی با نام ASP.NET Identity معرفی کردند. می‌توانید در این لینک بیشتر درباره این انتشار بخوانید.
در ادامه مقاله قبلی تحت عنوان مهاجرت از SQL Membership به ASP.NET Identity ، در این پست به مهاجرت داده‌های یک اپلیکیشن که از مدل Providers برای مدیریت اطلاعات کاربران، نقش‌ها و پروفایل‌ها استفاده می‌کند به مدل جدید ASP.NET Identity می‌پردازیم. تمرکز این مقاله اساسا روی مهاجرت داده‌های پروفایل کاربران خواهد بود، تا بتوان به سرعت از آنها در اپلیکیشن استفاده کرد. مهاجرت داده‌های عضویت و نقش ها، شبیه پروسه مهاجرت SQL Membership است. رویکردی که در ادامه برای مهاجرت داده پروفایل‌ها دنبال شده است، می‌تواند برای اپلیکیشنی با SQL Membership نیز استفاده شود.
بعنوان یک مثال، با اپلیکیشن وبی شروع می‌کنیم که توسط Visual Studio 2012 ساخته شده و از مدل Providers استفاده می‌کند. پس از آن یک سری کد برای مدیریت پروفایل ها، ثبت نام کاربران، افزودن اطلاعات پروفایل به کاربران و مهاجرت الگوی دیتابیس می‌نویسیم و نهایتا اپلیکیشن را بروز رسانی می‌کنیم تا برای استفاده از سیستم Identity برای مدیریت کاربران و نقش‌ها آماده باشد. و بعنوان یک تست، کاربرانی که قبلا توسط Universal Providers ساخته شده اند باید بتوانند به سایت وارد شوند، و کاربران جدید هم باید قادر به ثبت نام در سایت باشند.
سورس کد کامل این مثال را می‌توانید از این لینک دریافت کنید.

خلاصه مهاجرت داده پروفایل ها
قبل از آنکه با مهاجرت‌ها شروع کنیم، بگذارید تا نگاهی به تجربه مان از ذخیره اطلاعات پروفایل‌ها در مدل Providers بیاندازیم.
اطلاعات پروفایل کاربران یک اپلیکیشن به طرق مختلفی می‌تواند ذخیره شود. یکی از رایج‌ترین این راه ها، استفاده از تامین کننده‌های پیش فرضی است که بهمراه Universal Providers منتشر شدند. بدین منظور انجام مراحل زیر لازم است
کلاس جدیدی بسازید که دارای خواصی برای ذخیره اطلاعات پروفایل است.
کلاس جدیدی بسازید که از 'ProfileBase' ارث بری می‌کند و متدهای لازم برای دریافت پروفایل کاربران را پیاده سازی می‌کند.
استفاده از تامین کننده‌های پیش فرض را، در فایل web.config فعال کنید. و کلاسی که در مرحله 2 ساختید را بعنوان کلاس پیش فرض برای خواندن اطلاعات پروفایل معرفی کنید.

اطلاعات پروفایل‌ها بصورت serialized xml و binary در جدول 'Profiles' ذخیره می‌شوند.

پس از آنکه به سیستم ASP.NET Identity مهاجرت کردیم، اطلاعات پروفایل deserialized شده و در قالب خواص کلاس User ذخیره می‌شوند. هر خاصیت، بعدا می‌تواند به یک ستون در دیتابیس متصل شود. مزیت بدست آمده این است که مستقیما از کلاس User به اطلاعات پروفایل دسترسی داریم. ناگفته نماند که دیگر داده‌ها serialize/deserialize هم نمی‌شوند.

شروع به کار در Visual Studio 2012 پروژه جدیدی از نوع ASP.NET 4.5 Web Forms application بسازید. مثال جاری از یک قالب Web Forms استفاده می‌کند، اما می‌توانید از یک قالب MVC هم استفاده کنید.

پوشه جدیدی با نام 'Models' بسازید تا اطلاعات پروفایل را در آن قرار دهیم.

بعنوان یک مثال، بگذارید تا تاریخ تولد کاربر، شهر سکونت، قد و وزن او را در پروفایلش ذخیره کنیم. قد و وزن بصورت یک کلاس سفارشی (custom class) بنام 'PersonalStats' ذخیره می‌شوند. برای ذخیره و بازیابی پروفایل ها، به کلاسی احتیاج داریم که 'ProfileBase' را ارث بری می‌کند. پس کلاس جدیدی با نام 'AppProfile' بسازید.

```
public class ProfileInfo
{
    public ProfileInfo()
    {
        UserStats = new PersonalStats();
    }
    public DateTime? DateOfBirth { get; set; }
    public PersonalStats UserStats { get; set; }
    public string City { get; set; }
}

public class PersonalStats
{
    public int? Weight { get; set; }
    public int? Height { get; set; }
}

public class AppProfile : ProfileBase
{
    public ProfileInfo ProfileInfo
```

```
    {
        get { return (ProfileInfo)GetPropertyValue("ProfileInfo"); }
    }
    public static AppProfile GetProfile()
    {
        return (AppProfile)HttpContext.Current.Profile;
    }
    public static AppProfile GetProfile(string userName)
    {
        return (AppProfile)Create(userName);
    }
}
```

پروفایل را در فایل web.config خود فعال کنید. نام کلاسی را که در مرحله قبل ساختید، بعنوان کلاس پیش فرض برای ذخیره و بازیابی پروفایل‌ها معرفی کنید.

```
<profile defaultProvider="DefaultProfileProvider" enabled="true"
    inherits="UniversalProviders_ProfileMigrations.Models.AppProfile">
  <providers>
    .....
  </providers>
</profile>
```

برای دریافت اطلاعات پروفایل از کاربر، فرم وب جدیدی در پوشه Account بسازید و آنرا 'AddProfileData.aspx' نامگذاری کنید.

```
<h2> Add Profile Data for <%# User.Identity.Name %></h2>
<asp:Label Text="" ID="Result" runat="server" />
<div>
    Date of Birth:
    <asp:TextBox runat="server" ID="DateOfBirth"/>
</div>
<div>
    Weight:
    <asp:TextBox runat="server" ID="Weight"/>
</div>
<div>
    Height:
    <asp:TextBox runat="server" ID="Height"/>
</div>
<div>
    City:
    <asp:TextBox runat="server" ID="City"/>
</div>
<div>
    <asp:Button Text="Add Profile" ID="Add" OnClick="Add_Click" runat="server" />
</div>
```

کد زیر را هم به فایل code-behind اضافه کنید.

```
protected void Add_Click(object sender, EventArgs e)
{
    AppProfile profile = AppProfile.GetProfile(User.Identity.Name);
    profile.ProfileInfo.DateOfBirth = DateTime.Parse(DateOfBirth.Text);
    profile.ProfileInfo.UserStats.Weight = Int32.Parse(Weight.Text);
    profile.ProfileInfo.UserStats.Height = Int32.Parse(Height.Text);
    profile.ProfileInfo.City = City.Text;
    profile.Save();
}
```

دقت کنید که فضای نامی که کلاس AppProfile در آن قرار دارد را وارد کرده باشید.

اپلیکیشن را اجرا کنید و کاربر جدیدی با نام ' **olduser** ' بسازید. به صفحه جدید 'AddProfileData' بروید و اطلاعات پروفایل کاربر را وارد کنید.

با استفاده از پنجره Server Explorer می‌توانید تایید کنید که اطلاعات پروفایل با فرمت xml در جدول 'Profiles' ذخیره می‌شوند.

مهاجرت الگوی دیتابیس

برای اینکه دیتابیس فعلی بتواند با سیستم ASP.NET Identity کار کند، باید الگوی ASP.NET Identity را بروز رسانی کنیم تا فیلدهای جدیدی که اضافه کردیم را هم در نظر بگیرد. این کار می‌تواند توسط اسکریپت‌های SQL انجام شود، باید جداول جدیدی بسازیم و اطلاعات موجود را به آنها انتقال دهیم. در پنجره 'Server Explorer' گره 'DefaultConnection' را باز کنید تا جداول لیست شوند. روی Tables کلیک راست کنید و 'New Query' را انتخاب کنید.

اسکریپت مورد نیاز را از آدرس https://raw.github.com/suhasj/UniversalProviders-Identity-Migrations/master/Migration.txt دریافت کرده و آن را اجرا کنید. اگر اتصال خود به دیتابیس را تازه کنید خواهید دید که جداول جدیدی اضافه شده اند. می‌توانید داده‌های این جداول را بررسی کنید تا ببینید چگونه اطلاعات منتقل شده اند.

مهاجرت اپلیکیشن برای استفاده از ASP.NET Identity

پکیج‌های مورد نیاز برای ASP.NET Identity را نصب کنید:

Microsoft.AspNet.Identity.EntityFramework
Microsoft.AspNet.Identity.Owin
Microsoft.Owin.Host.SystemWeb
Microsoft.Owin.Security.Facebook
Microsoft.Owin.Security.Google
Microsoft.Owin.Security.MicrosoftAccount
Microsoft.Owin.Security.Twitter

اطلاعات بیشتری درباره مدیریت پکیج‌های NuGet از اینجا قابل دسترسی هستند.
برای اینکه بتوانیم از الگوی جاری دیتابیس استفاده کنیم، ابتدا باید مدل‌های لازم ASP.NET Identity را تعریف کنیم تا موجودیت‌های دیتابیس را Map کنیم. طبق قرارداد سیستم Identity کلاس‌های مدل یا باید اینترفیس‌های تعریف شده در Identity.Core dll را پیاده سازی کنند، یا می‌توانند پیاده سازی‌های پیش فرضی را که در Microsoft.AspNet.Identity.EntityFramework وجود دارند گسترش دهند. ما برای نقش ها، اطلاعات ورود کاربران و claimها از پیاده سازی‌های پیش فرض استفاده خواهیم کرد. نیاز به استفاده از یک کلاس سفارشی User داریم. پوشه جدیدی در پروژه با نام 'IdentityModels' بسازید. کلاسی با نام 'User' در این پوشه بسازید و کد آن را با لیست زیر تطابق دهید.

```
using Microsoft.AspNet.Identity.EntityFramework;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using UniversalProviders_ProfileMigrations.Models;

namespace UniversalProviders_Identity_Migrations
{
    public class User : IdentityUser
    {
        public User()
        {
            CreateDate = DateTime.UtcNow;
            IsApproved = false;
            LastLoginDate = DateTime.UtcNow;
            LastActivityDate = DateTime.UtcNow;
            LastPasswordChangedDate = DateTime.UtcNow;
            Profile = new ProfileInfo();
        }

        public System.Guid ApplicationId { get; set; }
        public bool IsAnonymous { get; set; }
        public System.DateTime? LastActivityDate { get; set; }
        public string Email { get; set; }
        public string PasswordQuestion { get; set; }
        public string PasswordAnswer { get; set; }
        public bool IsApproved { get; set; }
        public bool IsLockedOut { get; set; }
        public System.DateTime? CreateDate { get; set; }
        public System.DateTime? LastLoginDate { get; set; }
        public System.DateTime? LastPasswordChangedDate { get; set; }
        public System.DateTime? LastLockoutDate { get; set; }
        public int FailedPasswordAttemptCount { get; set; }
        public System.DateTime? FailedPasswordAttemptWindowStart { get; set; }
        public int FailedPasswordAnswerAttemptCount { get; set; }
        public System.DateTime? FailedPasswordAnswerAttemptWindowStart { get; set; }
        public string Comment { get; set; }
        public ProfileInfo Profile { get; set; }
    }
}
```

دقت کنید که 'ProfileInfo' حالا بعنوان یک خاصیت روی کلاس User تعریف شده است. بنابراین می‌توانیم مستقیما از کلاس کاربر با اطلاعات پروفایل کار کنیم.
محتویات پوشه‌های **IdentityModels** و **IdentityAccount** را از آدرس https://github.com/suhasj/UniversalProviders-Identity-Migrations/tree/master/UniversalProviders-Identity-Migrations دریافت و کپی کنید. این فایل‌ها مابقی مدل ها، و صفحاتی برای مدیریت کاربران و نقش‌ها در سیستم جدید ASP.NET Identity هستند.

انتقال داده پروفایل‌ها به جداول جدید

همانطور که گفته شد ابتدا باید داده‌های پروفایل را deserialize کرده و از فرمت xml خارج کنیم، سپس آنها را در ستون‌های جدول AspNetUsers ذخیره کنیم. ستون‌های جدید در مرحله قبل به دیتابیس اضافه شدند، پس تنها کاری که باقی مانده پر کردن این ستون‌ها با داده‌های ضروری است. بدین منظور ما از یک اپلیکیشن کنسول استفاده می‌کنیم که تنها یک بار اجرا خواهد شد، و ستون‌های جدید را با داده‌های لازم پر می‌کند.

در solution جاری یک پروژه اپلیکیشن کنسول بسازید.

آخرین نسخه پکیج Entity Framework را نصب کنید. همچنین یک رفرنس به اپلیکیشن وب پروژه بدهید (کلیک راست روی پروژه و گزینه 'Add Reference').

کد زیر را در کلاس Program.cs وارد کنید. این قطعه کد پروفایل تک تک کاربران را می‌خواند و در قالب 'ProfileInfo' آنها را serialize می‌کند و در دیتابیس ذخیره می‌کند.

```
public class Program
{
    var dbContext = new ApplicationDbContext();
    foreach (var profile in dbContext.Profiles)
    {
        var stringId = profile.UserId.ToString();
        var user = dbContext.Users.Where(x => x.Id == stringId).FirstOrDefault();
        Console.WriteLine("Adding Profile for user:" + user.UserName);
        var serializer = new XmlSerializer(typeof(ProfileInfo));
        var stringReader = new StringReader(profile.PropertyValueStrings);
        var profileData = serializer.Deserialize(stringReader) as ProfileInfo;
        if (profileData == null)
        {
            Console.WriteLine("Profile data deserialization error for user:" + user.UserName);
        }
        else
```

```
        {
            user.Profile = profileData;
        }
    }
    dbContext.SaveChanges();
}
```

برخی از مدل‌های استفاده شده در پوشه 'IdentityModels' تعریف شده اند که در پروژه اپلیکیشن وبمان قرار دارند، بنابراین افزودن فضاهای نام مورد نیاز فراموش نشود.

کد بالا روی دیتابیسی که در پوشه App_Data وجود دارد کار می‌کند، این دیتابیس در مراحل قبلی در اپلیکیشن وب پروژه ایجاد شد. برای اینکه این دیتابیس را رفرنس کنیم باید رشته اتصال فایل app.config اپلیکیشن کنسول را بروز رسانی کنید. از همان رشته اتصال web.config در اپلیکیشن وب پروژه استفاده کنید. همچنین آدرس فیزیکی کامل را در خاصیت 'AttachDbFilename' وارد کنید.

یک Command Prompt باز کنید و به پوشه bin اپلیکیشن کنسول بالا بروید. فایل اجرایی را اجرا کنید و نتیجه را مانند تصویر زیر بررسی کنید.

در پنجره Server Explorer جدول 'AspNetUsers' را باز کنید. حال ستون‌های این جدول باید خواص کلاس مدل را منعکس کنند.

کارایی سیستم را تایید کنید

با استفاده از صفحات جدیدی که برای کار با ASP.NET Identity پیاده سازی شده اند سیستم را تست کنید. با کاربران قدیمی که در دیتابیس قبلی وجود دارند وارد شوید. کاربران باید با همان اطلاعات پیشین بتوانند وارد سیستم شوند. مابقی قابلیت‌ها را هم بررسی کنید. مثلا افزودن OAuth، ثبت کاربر جدید، تغییر کلمه عبور، افزودن نقش ها، تخصیص کاربران به نقش‌ها و غیره.
داده‌های پروفایل کاربران قدیمی و جدید همگی باید در جدول کاربران ذخیره شده و بازیابی شوند. جدول قبلی دیگر نباید رفرنس شود.

عنوان: پیاده سازی Password Policy سفارشی توسط ASP.NET Identity
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۲۲ ۲۱:۱۰
آدرس: www.dotnettips.info
گروه‌ها: Tips, Identity, Visual Studio 2013, ASP.NET Identity

برای فراهم کردن یک تجربه کاربری ایمن‌تر و بهتر، ممکن است بخواهید پیچیدگی password policy را سفارشی سازی کنید. مثلا ممکن است بخواهید حداقل تعداد کاراکترها را تنظیم کنید، استفاده از چند حروف ویژه را اجباری کنید، جلوگیری از استفاده نام کاربر در کلمه عبور و غیره. برای اطلاعات بیشتر درباره سیاست‌های کلمه عبور به این لینک مراجعه کنید. بصورت پیش فرض ASP.NET Identity کاربران را وادار می‌کند تا کلمه‌های عبوری بطول حداقل 6 کاراکتر وارد نمایند. در ادامه نحوه افزودن چند خط مشی دیگر را هم بررسی می‌کنیم.

با استفاده از ویژوال استودیو 2013 پروژه جدیدی خواهیم ساخت تا از ASP.NET Identity استفاده کند. مواردی که درباره کلمه‌های عبور می‌خواهیم اعمال کنیم در زیر لیست شده اند.

تنظیمات پیش فرض باید تغییر کنند تا کلمات عبور حداقل 10 کاراکتر باشند
کلمه عبور حداقل یک عدد و یک کاراکتر ویژه باید داشته باشد
امکان استفاده از 5 کلمه عبور اخیری که ثبت شده وجود ندارد

در آخر اپلیکیشن را اجرا می‌کنیم و عملکرد این قوانین جدید را بررسی خواهیم کرد.

ایجاد اپلیکیشن جدید

در Visual Studio 2013 اپلیکیشن جدیدی از نوع ASP.NET MVC 4.5 بسازید.

در پنجره Solution Explorer روی نام پروژه کلیک راست کنید و گزینه Manage NuGet Packages را انتخاب کنید. به قسمت **Update** بروید و تمام انتشارات جدید را در صورت وجود نصب کنید.

بگذارید تا به روند کلی ایجاد کاربران جدید در اپلیکیشن نگاهی بیاندازیم. این به ما در شناسایی نیازهای جدیدمان کمک می‌کند. در پوشه Controllers فایلی بنام AccountController.cs وجود دارد که حاوی متدهایی برای مدیریت کاربران است.

کنترلر Account از کلاس UserManager استفاده می‌کند که در فریم ورک Identity تعریف شده است. این کلاس به نوبه خود از کلاس دیگری بنام UserStore استفاده می‌کند که برای دسترسی و مدیریت داده‌های کاربران استفاده می‌شود. در مثال ما این کلاس از Entity Framework استفاده می‌کند که پیاده سازی پیش فرض است.

متد Register POST یک کاربر جدید می‌سازد. متد CreateAsync به طبع متد 'ValidateAsync' را روی خاصیت PasswordValidator فراخوانی می‌کند تا کلمه عبور دریافتی اعتبارسنجی شود.

```
var user = new ApplicationUser() { UserName = model.UserName };
var result = await UserManager.CreateAsync(user, model.Password);

if (result.Succeeded)
{
    await SignInAsync(user, isPersistent: false);
    return RedirectToAction("Index", "Home");
}
```

در صورت موفقیت آمیز بودن عملیات ایجاد حساب کاربری، کاربر به سایت وارد می‌شود.

قانون 1: کلمه‌های عبور باید حداقل 10 کاراکتر باشند

بصورت پیش فرض خاصیت PasswordValidator در کلاس UserManager به کلاس MinimumLengthValidator تنظیم شده است، که اطمینان حاصل می‌کند کلمه عبور حداقل 6 کاراکتر باشد. هنگام وهله سازی UserManager می‌توانید این مقدار را تغییر دهید.
مقدار حداقل کاراکترهای کلمه عبور به دو شکل می‌تواند تعریف شود. راه اول، تغییر کنترلر Account است. در متد سازنده این کنترلر کلاس UserManager وهله سازی می‌شود، همینجا می‌توانید این تغییر را اعمال کنید. راه دوم، ساختن کلاس جدیدی است که از UserManager ارث بری می‌کند. سپس می‌توان این کلاس را در سطح global تعریف کرد. در پوشه IdentityExtensions کلاس جدیدی با نام ApplicationUserManager بسازید.

```
public class ApplicationUserManager : UserManager<ApplicationUser>
{
    public ApplicationUserManager(): base(new UserStore<ApplicationUser>(new ApplicationDbContext()))
    {
        PasswordValidator = new MinimumLengthValidator (10);
    }
}
```

کلاس UserManager یک نمونه از کلاس IUserStore را دریافت می‌کند که پیاده سازی APIهای مدیریت کاربران است. از آنجا که کلاس UserStore مبتنی بر Entity Framework است، باید آبجکت DbContext را هم پاس دهیم. این کد در واقع همان کدی است که در متد سازنده کنترلر Account وجود دارد.
یک مزیت دیگر این روش این است که می‌توانیم متدهای UserManager را بازنویسی (overwrite) کنیم. برای پیاده سازی نیازمندهای بعدی دقیقا همین کار را خواهیم کرد.

حال باید کلاس ApplicationUserManager را در کنترلر Account استفاده کنیم. متد سازنده و خاصیت UserManager را مانند زیر تغییر دهید.

```
public AccountController() : this(new ApplicationUserManager())
        {
        }

        public AccountController(ApplicationUserManager userManager)
        {
            UserManager = userManager;
        }
        public ApplicationUserManager UserManager { get; private set; }
```

حالا داریم از کلاس سفارشی جدیدمان استفاده می‌کنیم. این به ما اجازه می‌دهد مراحل بعدی سفارشی سازی را انجام دهیم، بدون آنکه کدهای موجود در کنترلر از کار بیافتند.
اپلیکیشن را اجرا کنید و سعی کنید کاربر محلی جدیدی ثبت نمایید. اگر کلمه عبور وارد شده کمتر از 10 کاراکتر باشد پیغام خطای زیر را دریافت می‌کنید.

قانون 2: کلمه‌های عبور باید حداقل یک عدد و یک کاراکتر ویژه داشته باشند

چیزی که در این مرحله نیاز داریم کلاس جدیدی است که اینترفیس IIdentityValidator را پیاده سازی می‌کند. چیزی که ما می‌خواهیم اعتبارسنجی کنیم، وجود اعداد و کاراکترهای ویژه در کلمه عبور است، همچنین طول مجاز هم بررسی می‌شود. نهایتا این قوانین اعتبارسنجی در متد 'ValidateAsync' بکار گرفته خواهند شد.

در پوشه IdentityExtensions کلاس جدیدی بنام CustomPasswordValidator بسازید و اینترفیس مذکور را پیاده سازی کنید. از آنجا که نوع کلمه عبور رشته (string) است از IIdentityValidator<string> استفاده می‌کنیم.

```
public class CustomPasswordValidator : IIdentityValidator<string>
{
    public int RequiredLength { get; set; }

    public CustomPasswordValidator(int length)
    {
        RequiredLength = length;
    }

    public Task<IdentityResult> ValidateAsync(string item)
    {
        if (String.IsNullOrEmpty(item) || item.Length < RequiredLength)
        {
            return Task.FromResult(IdentityResult.Failed(String.Format("Password should be of length
{0}",RequiredLength)));
        }

    string pattern = @"^(?=.*[0-9])(?=.*[!@#$%^&*])[0-9a-zA-Z!@#$%^&*0-9]{10,}$";

    if (!Regex.IsMatch(item, pattern))
    {
        return Task.FromResult(IdentityResult.Failed("Password should have one numeral and one special
character"));
    }
```

```
    return Task.FromResult(IdentityResult.Success);
 }
```

در متد **ValidateAsync** بررس می‌کنیم که طول کلمه عبور معتبر و مجاز است یا خیر. سپس با استفاده از یک RegEx وجود کاراکترهای ویژه و اعداد را بررسی می‌کنیم. دقت کنید که regex استفاده شده تست نشده و تنها بعنوان یک مثال باید در نظر گرفته شود.

قدم بعدی تعریف این اعتبارسنج سفارشی در کلاس UserManager است. باید مقدار خاصیت PasswordValidator را به این کلاس تنظیم کنیم. به کلاس ApplicationUserManager که پیشتر ساختید بروید و مقدار خاصیت PasswordValidator را به CustomPasswordValidator تغییر دهید.

```
public class ApplicationUserManager : UserManager<ApplicationUser>
{
    public ApplicationUserManager() : base(new UserStore<ApplicationUser(new ApplicationDbContext()))
    {
        PasswordValidator = new CustomPasswordValidator(10);
    }
 }
```

هیچ تغییر دیگری در کلاس AccountController لازم نیست. حال سعی کنید کاربر جدید دیگری بسازید، اما اینبار کلمه عبوری وارد کنید که خطای اعتبارسنجی تولید کند. پیغام خطایی مشابه تصویر زیر باید دریافت کنید.

قانون 3: امکان استفاده از 5 کلمه عبور اخیر ثبت شده وجود ندارد

هنگامی که کاربران سیستم، کلمه عبور خود را بازنشانی (reset) می‌کنند یا تغییر می‌دهند، می‌توانیم بررسی کنیم که آیا مجددا از یک کلمه عبور پیشین استفاده کرده اند یا خیر. این بررسی بصورت پیش فرض انجام نمی‌شود، چرا که سیستم Identity تاریخچه کلمه‌های عبور کاربران را ذخیره نمی‌کند. می‌توانیم در اپلیکیشن خود جدول جدیدی بسازیم و تاریخچه کلمات عبور کاربران را در آن ذخیره کنیم. هربار که کاربر سعی در بازنشانی یا تغییر کلمه عبور خود دارد، مقدار Hash شده را در جدول تاریخچه بررسی میکنیم.

فایل IdentityModels.cs را باز کنید. مانند لیست زیر، کلاس جدیدی بنام 'PreviousPassword' بسازید.

```
public class PreviousPassword
{

 public PreviousPassword()
 {
    CreateDate = DateTimeOffset.Now;
 }

 [Key, Column(Order = 0)]

 public string PasswordHash { get; set; }

 public DateTimeOffset CreateDate { get; set; }

 [Key, Column(Order = 1)]

 public string UserId { get; set; }

 public virtual ApplicationUser User { get; set; }

 }
```

در این کلاس، فیلد 'Password' مقدار Hash شده کلمه عبور را نگاه میدارد و توسط فیلد 'UserId' رفرنس می‌شود. فیلد 'CreateDate' یک مقدار timestamp ذخیره می‌کند که تاریخ ثبت کلمه عبور را مشخص می‌نماید. توسط این فیلد می‌توانیم تاریخچه کلمات عبور را فیلتر کنیم و مثلا 5 رکورد آخر را بگیریم.

Entity Framework Code First جدول 'PreviousPasswords' را می‌سازد و با استفاده از فیلدهای 'UserId' و 'Password' کلید اصلی (composite primary key) را ایجاد می‌کند. برای اطلاعات بیشتر درباره قرارداهای EF Code First به این لینک مراجعه کنید.

خاصیت جدیدی به کلاس ApplicationUser اضافه کنید تا لیست آخرین کلمات عبور استفاده شده را نگهداری کند.

```
public class ApplicationUser : IdentityUser
{
    public ApplicationUser() : base()
    {
        PreviousUserPasswords = new List<PreviousPassword>();
     }

     public virtual IList<PreviousPassword> PreviousUserPasswords { get; set; }
}
```

همانطور که پیشتر گفته شد، کلاس UserStore پیاده سازی APIهای لازم برای مدیریت کاربران را در بر می‌گیرد. هنگامی که کاربر برای نخستین بار در سایت ثبت می‌شود باید مقدار Hash کلمه عبورش را در جدول تاریخچه کلمات عبور ذخیره کنیم. از آنجا که UserStore بصورت پیش فرض متدی برای چنین عملیاتی معرفی نمی‌کند، باید یک override تعریف کنیم تا این مراحل را انجام دهیم. پس ابتدا باید کلاس سفارشی جدیدی بسازیم که از UserStore ارث بری کرده و آن را توسعه می‌دهد. سپس از این کلاس سفارشی در ApplicationUserManager بعنوان پیاده سازی پیش فرض UserStore استفاده می‌کنیم. پس کلاس جدیدی در پوشه IdentityExtensions ایجاد کنید.

```
public class ApplicationUserStore : UserStore<ApplicationUser>
{
    public ApplicationUserStore(DbContext context) : base(context)  { }
```

```
    public override async Task CreateAsync(ApplicationUser user)
    {
        await base.CreateAsync(user);
        await AddToPreviousPasswordsAsync(user, user.PasswordHash);
    }

     public Task AddToPreviousPasswordsAsync(ApplicationUser user, string password)
     {
        user.PreviousUserPasswords.Add(new PreviousPassword() { UserId = user.Id, PasswordHash =
password });

        return UpdateAsync(user);
     }
 }
```

متد 'AddToPreviousPasswordsAsync' کلمه عبور را در جدول 'PreviousPasswords' ذخیره می‌کند.

هرگاه کاربر سعی در بازنشانی یا تغییر کلمه عبورش دارد باید این متد را فراخوانی کنیم. APIهای لازم برای این کار در کلاس UserManager تعریف شده اند. باید این متدها را override کنیم و فراخوانی متد مذکور را پیاده کنیم. برای این کار کلاس ApplicationUserManager را باز کنید و متدهای ChangePassword و ResetPassword را بازنویسی کنید.

```
public class ApplicationUserManager : UserManager<ApplicationUser>
    {
        private const int PASSWORD_HISTORY_LIMIT = 5;

        public ApplicationUserManager() : base(new ApplicationUserStore(new ApplicationDbContext()))
        {
            PasswordValidator = new CustomPasswordValidator(10);
        }

        public override async Task<IdentityResult> ChangePasswordAsync(string userId, string
currentPassword, string newPassword)
        {
            if (await IsPreviousPassword(userId, newPassword))
            {
                return await Task.FromResult(IdentityResult.Failed("Cannot reuse old password"));
            }

            var result = await base.ChangePasswordAsync(userId, currentPassword, newPassword);

            if (result.Succeeded)
            {
                var store = Store as ApplicationUserStore;
                await store.AddToPreviousPasswordsAsync(await FindByIdAsync(userId),
PasswordHasher.HashPassword(newPassword));
            }

            return result;
        }

        public override async Task<IdentityResult> ResetPasswordAsync(string userId, string token,
string newPassword)
        {
            if (await IsPreviousPassword(userId, newPassword))
            {
                return await Task.FromResult(IdentityResult.Failed("Cannot reuse old password"));
            }

            var result = await base.ResetPasswordAsync(userId, token, newPassword);

            if (result.Succeeded)
            {
                var store = Store as ApplicationUserStore;
                await store.AddToPreviousPasswordsAsync(await FindByIdAsync(userId),
PasswordHasher.HashPassword(newPassword));
            }

            return result;
        }

        private async Task<bool> IsPreviousPassword(string userId, string newPassword)
        {
            var user = await FindByIdAsync(userId);
```

```
            if (user.PreviousUserPasswords.OrderByDescending(x => x.CreateDate).
                Select(x => x.PasswordHash).Take(PASSWORD_HISTORY_LIMIT)
                .Where(x => PasswordHasher.VerifyHashedPassword(x, newPassword) !=
PasswordVerificationResult.Failed).Any())
            {
                return true;
            }

            return false;
        }
    }
```

فیلد 'PASSWORD_HISTORY_LIMIT' برای دریافت X رکورد از جدول تاریخچه کلمه عبور استفاده می‌شود. همانطور که می‌بینید از متد سازنده کلاس ApplicationUserStore برای گرفتن متد جدیدمان استفاده کرده ایم. هرگاه کاربری سعی می‌کند کلمه عبورش را بازنشانی کند یا تغییر دهد، کلمه عبورش را با 5 کلمه عبور قبلی استفاده شده مقایسه می‌کنیم و بر این اساس مقدار true/false بر می‌گردانیم.

کاربر جدیدی بسازید و به صفحه **Manage** بروید. حال سعی کنید کلمه عبور را تغییر دهید و از کلمه عبور فعلی برای مقدار جدید استفاده کنید تا خطای اعتبارسنجی تولید شود. پیغامی مانند تصویر زیر باید دریافت کنید.

سورس کد این مثال را می‌توانید از این لینک دریافت کنید. نام پروژه **Identity-PasswordPolicy** است، و زیر قسمت Samples/Identity قرار دارد.

عنوان: **انتشار پیش نمایش ASP.NET Identity 2.0.0-alpha1**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۲۳ ۱۶:۳۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, MemberShip, Visual Studio 2013, ASP.NET Identity

مایکروسافت در تاریخ 20 دسامبر 2013 پیش نمایش نسخه جدید ASP.NET Identity را معرفی کرد. تمرکز اصلی در این انتشار، رفع مشکلات نسخه 1.0 بود. امکانات جدیدی هم مانند Account Confirmation و Password Reset اضافه شده اند.

دانلود این انتشار

ASP.NET Identity را می‌توانید در قالب یک پکیج NuGet دریافت کنید. در پنجره Manage NuGet Packages می‌توانید پکیج‌های Preview را لیست کرده و گزینه مورد نظر را

نصب کنید. برای نصب پکیج‌های pre-release توسط Package Manager Console از فرامین زیر استفاده کنید.

```
Install-Package Microsoft.AspNet.Identity.EntityFramework -Version 2.0.0-alpha1 -Pre
Install-Package Microsoft.AspNet.Identity.Core -Version 2.0.0-alpha1 -Pre
Install-Package Microsoft.AspNet.Identity.OWIN -Version 2.0.0-alpha1 -Pre
```

دقت کنید که حتما از گزینه "Include Prerelease" استفاده می‌کنید. برای اطلاعات بیشتر درباره نصب پکیج‌های Pre-release لطفا به این لینک و یا این لینک مراجعه کنید.

در ادامه لیست امکانات جدید و مشکلات رفع شده را می‌خوانید.

Account Confirmation

سیستم ASP.NET Identity حالا از Account Confirmation پشتیبانی می‌کند. این یک سناریوی بسیار رایج است. در اکثر وب سایت‌های امروزی پس از ثبت نام، حتما باید ایمیل خود را تایید کنید. پیش از تایید ثبت نام قادر به انجام هیچ کاری در وب سایت نخواهید بود، یعنی نمی‌توانید Login کنید. این روش مفید است، چرا که از ایجاد حساب‌های کاربری نامعتبر (bogus) جلوگیری می‌کند. همچنین این روش برای برقراری ارتباط با کاربران هم بسیار کارآمد است. از آدرس‌های ایمیل کاربران می‌توانید در وب سایت‌های فروم، شبکه‌های اجتماعی، تجارت آنلاین و بانکداری برای اطلاع رسانی و دیگر موارد استفاده کنید.

نکته: برای ارسال ایمیل باید تنظیمات SMTP را پیکربندی کنید. مثلا می‌توانید از سرویس‌های ایمیل محبوبی مانند SendGrid استفاده کنید، که با Windows Azure براحتی یکپارچه می‌شود و از طرف توسعه دهنده اپلیکیشن هم نیاز به پیکربندی ندارد.

در مثال زیر نیاز دارید تا یک سرویس ایمیل برای ارسال ایمیل‌ها پیکربندی کنید. همچنین کاربران پیش از تایید ایمیل شان قادر به بازنشانی کلمه عبور نیستند.

Password Reset

این هم یک سناریوی رایج و استاندارد است. کاربران در صورتی که کلمه عبورشان را فراموش کنند، می‌توانند از این قابلیت برای بازنشانی آن استفاده کنند. کلمه عبور جدیدی بصورت خودکار تولید شده و برای آنها ارسال می‌شود. کاربران با استفاده از این رمز عبور جدید می‌توانند وارد سایت شوند و سپس آن را تغییر دهند.

Security Token Provider

هنگامی که کاربران کلمه عبورشان را تغییر می‌دهند، یا اطلاعات امنیتی خود را بروز رسانی می‌کنند (مثلا حذف کردن لاگین‌های خارجی مثل فیسبوک، گوگل و غیره) باید شناسه امنیتی (security token) کاربر را بازتولید کنیم و مقدار قبلی را Invalidate یا بی اعتبار سازیم. این کار بمنظور حصول اطمینان از بی اعتبار بودن تمام شناسه‌های قبلی است که توسط کلمه عبور پیشین تولید شده بودند. این قابلیت، یک لایه امنیتی بیشتر برای اپلیکیشن شما فراهم می‌کند. چرا که وقتی کاربری کلمه عبورش را تغییر بدهد از همه جا logged-out می‌شود. یعنی از تمام مرورگرهایی که برای استفاده از اپلیکیشن استفاده کرده خارج خواهد شد.

برای پیکربندی تنظیمات این قابلیت می‌توانید از فایل Startup.Auth.cs استفاده کنید. می‌توانید مشخص کنید که میان افزار OWIN cookie هر چند وقت یکبار باید شناسه امنیتی کاربران را بررسی کند. به لیست زیر دقت کنید.

```
// 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(newCookieAuthenticationOptions {
    AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
    LoginPath = newPathString("/Account/Login"),
    Provider = newCookieAuthenticationProvider {
        OnValidateIdentity = SecurityStampValidator.OnValidateIdentity<ApplicationUserManager,
ApplicationUser>(
```

```
            validateInterval: TimeSpan.FromSeconds(5),
            regenerateIdentity: (manager, user) => user.GenerateUserIdentityAsync(manager))
    }
});
```

امکان سفارشی کردن کلیدهای اصلی Users و Roles

در نسخه 1.0 نوع فیلدهای کلید اصلی در جداول Users و Roles از نوع رشته (string) بود. این بدین معنا است که وقتی از Entity Framework و Sql Server برای ذخیره داده‌های ASP.NET Identity استفاده می‌کنیم داده‌های این فیلدها بعنوان nvarchar ذخیره می‌شوند. درباره این پیاده سازی پیش فرض در فروم هایی مانند سایت StackOverflow بسیار بحث شده است. و در آخر با در نظر گرفتن تمام بازخورد ها، تصمیم گرفته شد یک نقطه توسعه پذیری (extensibility) اضافه شود که توسط آن بتوان نوع فیلدهای اصلی را مشخص کرد. مثلا شاید بخواهید کلیدهای اصلی جداول Users و Roles از نوع int باشند. این نقطه توسعه پذیری مخصوصا هنگام مهاجرت داده‌های قبلی بسیار مفید است، مثلا ممکن است دیتابیس قبلی فیلدهای UserId را با فرمت GUID ذخیره کرده باشد.

اگر نوع فیلدهای کلید اصلی را تغییر دهید، باید کلاس‌های مورد نیاز برای Claims و Logins را هم اضافه کنید تا کلید اصلی معتبری دریافت کنند. قطعه کد زیر نمونه ای از نحوه استفاده این قابلیت برای تعریف کلیدهای int را نشان می‌دهد.

```
de Snippet
publicclassApplicationUser : IdentityUser<int, CustomUserLogin, CustomUserRole, CustomUserClaim>
{
}
publicclassCustomRole : IdentityRole<int, CustomUserRole>
{
    public CustomRole() { }
    public CustomRole(string name) { Name = name; }
}
publicclassCustomUserRole : IdentityUserRole<int> { }
publicclassCustomUserClaim : IdentityUserClaim<int> { }
publicclassCustomUserLogin : IdentityUserLogin<int> { }

publicclassApplicationDbContext : IdentityDbContext<ApplicationUser, CustomRole, int, CustomUserLogin,
CustomUserRole, CustomUserClaim>
{
}
```

پشتیبانی از IQueryable روی Users و Roles

کلاس‌های UserStore و RoleStore حالا از IQueryable پشتیبانی می‌کنند، بنابراین می‌توانید براحتی لیست کاربران و نقش‌ها را کوئری کنید.

بعنوان مثال قطعه کد زیر دریافت لیست کاربران را نشان می‌دهد. از همین روش برای دریافت لیست نقش‌ها از RoleManager می‌توانید استفاده کنید.

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

پشتیبانی از عملیات Delete از طریق UserManager

در نسخه 1.0 اگر قصد حذف یک کاربر را داشتید، نمی‌توانستید این کار را از طریق UserManager انجام دهید. اما حالا می‌توانید مانند قطعه کد زیر عمل کنید.

```
var user = await UserManager.FindByIdAsync(id);

if (user == null)
{
    return HttpNotFound();
```

```
}
var result = await UserManager.DeleteAsync(user);
```

میان افزار UserManagerFactory

شما می‌توانید با استفاده از یک پیاده سازی Factory، وهله ای از UserManager را از OWIN context دریافت کنید. این الگو مشابه چیزی است که برای گرفتن AuthenticationManager در OWIN context استفاده می‌کنیم. این الگو همچنین روش توصیه شده برای گرفتن یک نمونه از UserManager به ازای هر درخواست در اپلیکیشن است.

قطعه کد زیر نحوه پیکربندی این میان افزار در فایل StartupAuth.cs را نشان می‌دهد.

```
// Configure the UserManager
app.UseUserManagerFactory(newUserManagerOptions<ApplicationUserManager>()
{
    DataProtectionProvider = app.GetDataProtectionProvider(),
    Provider = newUserManagerProvider<ApplicationUserManager>()
    {
        OnCreate = ApplicationUserManager.Create
    }
});
```

و برای گرفتن یک وهله از UserManager:

```
HttpContext.GetOwinContext().GetUserManager<ApplicationUserManager>();
```

میان افزار DbContextFactory

سیستم ASP.NET Identity از Entity Framework برای ذخیره داده هایش در Sql Server استفاده می‌کند. بدین منظور، ASP.NET Identity کلاس ApplicationDbContext را رفرنس می‌کند. میان افزار DbContextFactory به ازای هر درخواست در اپلیکیشن یک وهله از ApplicationDbContext را به شما تحویل می‌دهد.

می توانید پیکربندی لازم را در StartupAuth.cs انجام دهید.

```
app.UseDbContextFactory(ApplicationDbContext.Create);
```

Samples

امکانات جدید را می‌توانید در پروژه https://aspnet.codeplex.com پیدا کنید. لطفا به پوشه Identity در سورس کد مراجعه کنید. برای اطلاعاتی درباره نحوه اجرای پروژه هم فایل readme را بخوانید.

برای مستندات ASP.NET Identity 1.0 هم به http://www.asp.net/identity سر بزنید. هنوز مستنداتی برای نسخه 2.0 منتشر نشده، اما بزودی با انتشار نسخه نهایی مستندات و مثال‌های جدیدی به سایت اضافه خواهند شد.

Known Issues

در کنار قابلیت‌های جدیدی مانند Account Confirmation و Password Reset، دو خاصیت جدید به کلاس IdentityUser اضافه شده اند: 'Email' و 'IsConfirmed'. این تغییرات الگوی دیتابیسی که توسط ASP.NET Identity 1.0 ساخته شده است را تغییر می‌دهد. بروز رسانی پکیج‌ها از نسخه 1.0 به 2.0 باعث می‌شود که اپلیکیشن شما دیگر قادر به دسترسی به دیتابیس عضویت نباشد، چرا که مدل دیتابیس تغییر کرده. برای بروز رسانی الگوی دیتابیس می‌توانید از Code First Migrations استفاده کنید.

نکته: نسخه جدید به EntityFramework 6.1.0-alpha1 وابستگی دارد، که در همین تاریخ (20 دسامبر 2013) پیش نمایش شد. http://blogs.msdn.com/b/adonet/archive/2013/12/20/ef-6-1-alpha-1-available.aspx

EntityFramework 6.1.0-alpha1 بروز رسانی هایی دارد که سناریوی مهاجرت در ASP.NET Identity را تسهیل می‌کند، به همین دلیل از نسخه جدید EF استفاده شده. تیم ASP.NET هنوز باگ‌های زیادی را باید رفع کند و قابلیت‌های جدیدی را هم باید پیاده سازی کند. بنابراین پیش از نسخه نهایی RTM شاهد پیش نمایش‌های دیگری هم خواهیم بود که در ماه‌های آتی منتشر می‌شوند.

برای اطلاعات بیشتر درباره آینده ASP.NET Identity به لینک زیر سری بزنید.

https://aspnetidentity.codeplex.com/wikipage?title=Roadmap&version=1

نظرات خوانندگان

نویسنده: ابوالفضل رجب پور
تاریخ: ۱۳۹۲/۱۰/۳۰ ۲۰:۴۴

سلام

پیاده سازی Single Sign on در این سیستم کجا کار قرار داره؟ در واقع چطور میشه پیاده سازی ش کرد؟
در سیستم membership قبلی، اگر کلید اپلیکیشن رو در وب کانفیگ برنامه هاتون که دامین هاشون مشترک بود (در واقع ساب دامین ها)، یکسان وارد میکردی، برنامه‌ها بصورت SSO کار می‌کرد و احتیاجی به هیچ کاری نداشت. حالا در سیستم جدید همون روش جواب میده؟ برای برنامه‌های با دامین‌های متفاوت چطور؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۰/۳۰ ۲۳:۷

CookieDomain رو باید تنظیم کنید.

عنوان: **پیاده سازی یک تامین کننده MySQL برای ASP.NET Identity**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۲۴ ۱۷:۵۵
آدرس: www.dotnettips.info
گروه‌ها: MySQL, Security, ASP.NET Identity, Windows Azure

در این مقاله جایگزینی پیاده سازی پیش فرض ASP.NET Identity را بررسی می‌کنیم. در ادامه خواهید خواند:
جزئیات نحوه پیاده سازی یک Storage Provider برای ASP.NET Identity
تشریح اینترفیس هایی که باید پیاده سازی شوند، و نحوه استفاده از آنها در ASP.NET Identity
ایجاد یک دیتابیس MySQL روی Windows Azure
نحوه استفاده از یک ابزار کلاینت (MySQL Workbench) برای مدیریت دیتابیس مذکور
نحوه جایگزینی پیاده سازی سفارشی با نسخه پیش فرض در یک اپلیکیشن ASP.NET MVC

در انتهای این مقاله یک اپلیکیشن ASP.NET MVC خواهیم داشت که از ASP.NET Identity و تامین کننده سفارشی جدید استفاده می‌کند. دیتابیس اپلیکیشن MySQL خواهد بود و روی Windows Azure میزبانی می‌شود. سورس کد کامل این مثال را هم می‌توانید از این لینک دریافت کنید.

پیاده سازی یک Storage Provider سفارشی برای ASP.NET Identity

ASP.NET Identity سیستم توسعه پذیری است که می‌توانید بخش‌های مختلف آن را جایگزین کنید.در این سیستم بناهای سطح بالایی مانند Managers و Stores وجود دارند.

Managers کلاس‌های سطح بالایی هستند که توسعه دهندگان از آنها برای اجرای عملیات مختلف روی ASP.NET Identity استفاده می‌کنند. مدیریت کننده‌های موجود عبارتند از UserManager و RoleManager. کلاس UserManager برای اجرای عملیات مختلف روی کاربران استفاده می‌شود، مثلا ایجاد کاربر جدید یا حذف آنها. کلاس RoleManager هم برای اجرای عملیات مختلف روی نقش‌ها استفاده می‌شود.

Stores کلاس‌های سطح پایین‌تری هستند که جزئیات پیاده سازی را در بر می‌گیرند، مثلا اینکه موجودیت‌های کاربران و نقش‌ها چگونه باید ذخیره و بازیابی شوند. این کلاس‌ها با مکانیزم ذخیره و بازیابی تلفیق شده اند. مثلا Microsoft.AspNet.Identity.EntityFramework کلاسی با نام UserStore دارد که برای ذخیره و بازیابی Userها و داده‌های مربوطه توسط EntityFramework استفاده می‌شود.

Managers از Stores تفکیک شده اند و هیچ وابستگی ای به یکدیگر ندارند. این تفکیک بدین منظور انجام شده که بتوانید مکانیزم ذخیره و بازیابی را جایگزین کنید، بدون اینکه اپلیکیشن شما از کار بیافتد یا نیاز به توسعه بیشتر داشته باشد. کلاس‌های Manager می‌توانند با هر Store ای ارتباط برقرار کنند. از آنجا که شما از APIهای سطح بالای UserManager برای انجام عملیات CRUD روی کاربران استفاده می‌کنید، اگر UserStore را با پیاده سازی دیگری جایگزین کنید، مثلا AzureTable Storage یا MySql، نیازی به بازنویسی اپلیکیشن نیست.

در مثال جاری پیاده سازی پیش فرض Entity Framework را با یک تامین کننده MySQL جایگزین می‌کنیم.

پیاده سازی کلاس‌های Storage

برای پیاده سازی تامین کننده‌های سفارشی، باید کلاس هایی را پیاده سازی کنید که همتای آنها در Microsoft.AspNet.Identity.EntityFramework وجود دارند:

```
UserStore<TUser>
IdentityUser
RoleStore<TRole>
IdentityRole
```

پیاده سازی پیش فرض Entity Framework را در تصاویر زیر مشاهده می‌کنید.

Users

Roles

در مخزن پیش فرض ASP.NET Identity EntityFramework کلاس‌های بیشتری برای موجودیت‌ها مشاهده می‌کنید.

IdentityUserClaim

IdentityUserLogin

IdentityUserRole

همانطور که از نام این کلاس‌ها مشخص است، اختیارات، نقش‌ها و اطلاعات ورود کاربران توسط این کلاس‌ها معرفی می‌شوند. در مثال جاری این کلاس‌ها را پیاده سازی نخواهیم کرد، چرا که بارگذاری اینگونه رکوردها از دیتابیس به حافظه برای انجام عملیات پایه (مانند افزودن و حذف اختیارات کاربران) سنگین است. در عوض کلاس‌های backend store اینگونه عملیات را بصورت مستقیم روی دیتابیس اجرا خواهند کرد. بعنوان نمونه متد UserStore.GetClaimsAsync() را در نظر بگیرید. این متد به نوبه خود متد userClaimTable.FindByUserId(user.Id) را فراخوانی می‌کند که یک کوئری روی جدول مربوطه اجرا می‌کند و لیستی از اختیارات کاربر را بر می‌گرداند.

```
public Task<IList<Claim>> GetClaimsAsync(IdentityUser user)
{
    ClaimsIdentity identity = userClaimsTable.FindByUserId(user.Id);
    return Task.FromResult<IList<Claim>>(identity.Claims.ToList());
```

```
}
```

برای پیاده سازی یک تامین کننده سفارشی MySQL مراحل زیر را دنبال کنید.

1. کلاس کاربر را ایجاد کنید، که اینترفیس **IUser** را پیاده سازی می‌کند.

```
public class IdentityUser : IUser
{
    public IdentityUser(){...}

    public IdentityUser(string userName) (){...}

    public string Id { get; set; }

    public string UserName { get; set; }

    public string PasswordHash { get; set; }

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

2. کلاس User Store را ایجاد کنید، که اینترفیس‌های **IUserRoleStore** , **IUserLoginStore** , **IUserClaimStore** , **IUserStore** و **IUserPasswordStore** را پیاده سازی می‌کند. توجه کنید که تنها اینترفیس IUserStore را باید پیاده سازی کنید، مگر آنکه بخواهید از امکاناتی که دیگر اینترفیس‌ها ارائه می‌کنند هم استفاده کنید.

```
public class UserStore : IUserStore<IdentityUser>,
                         IUserClaimStore<IdentityUser>,
                         IUserLoginStore<IdentityUser>,
                         IUserRoleStore<IdentityUser>,
                         IUserPasswordStore<IdentityUser>
{
    public UserStore(){...}

    public Task CreateAsync(IdentityUser user){...}

    public Task<IdentityUser> FindByIdAsync(string userId){...}
...
}
```

3. کلاس Role را ایجاد کنید که اینترفیس **IRole** را پیاده سازی می‌کند.

```
public class IdentityRole : IRole
{
    public IdentityRole(){...}

    public IdentityRole(string roleName) (){...}

    public string Id { get; set; }

    public string Name { get; set; }
}
```

4. کلاس Role Store را ایجاد کنید که اینترفیس **IRoleStore** را پیاده سازی می‌کند. توجه داشته باشید که پیاده سازی این مخزن اختیاری است و در صورتی لازم است که بخواهید از نقش‌ها در سیستم خود استفاده کنید.

```
public class RoleStore : IRoleStore<IdentityRole>
{
    public RoleStore(){...}

    public Task CreateAsync(IdentityRole role){...}

    public Task<IdentityRole> FindByIdAsync(string roleId){...}
....
}
```

کلاس‌های بیشتری هم وجود دارند که مختص پیاده سازی مثال جاری هستند.

MySQLDatabase: این کلاس اتصال دیتابیس MySql و کوئری‌ها را کپسوله می‌کند. کلاس‌های UserStore و RoleStore توسط نمونه ای از این کلاس وهله سازی می‌شوند.
RoleTable: این کلاس جدول Roles و عملیات CRUD مربوط به آن را کپسوله می‌کند.
UserClaimsTable: این کلاس جدول UserClaims و عملیات CRUD مربوط به آن را کپسوله می‌کند.
UserLoginsTable: این کلاس جدول UserLogins و عملیات CRUD مربوط به آن را کپسوله می‌کند.
UserRolesTable: این کلاس جدول UserRoles و عملیات CRUD مربوطه به آن را کپسوله می‌کند.
UserTable: این کلاس جدول Users و عملیات CRUD مربوط به آن را کپسوله می‌کند.

ایجاد یک دیتابیس MySQL روی Windows Azure

1. به پورتال مدیریتی Windows Azure وارد شوید.
2. در پایین صفحه روی **+NEW** کلیک کنید و گزینه **STORE** را انتخاب نمایید.

در ویزارد **Choose Add-on** به سمت پایین اسکرول کنید و گزینه **ClearDB MySQL Database** را انتخاب کنید. سپس به مرحله بعد بروید.

Choose an Add-on

ALL

APP SERVICES

DATA

ClearDB MySQL Database

SuccessBricks, Inc.

ClearDB is a powerful, fault-tolerant database-as-a-service in the cloud for your MySQL powered applications.

PUBLISHED DATE 10/10/2012

2 3

4. راهکار **Free** بصورت پیش فرض انتخاب شده، همین گزینه را انتخاب کنید و نام دیتابیس را به **IdentityMySQLDatabase** تغییر دهید. نزدیک‌ترین ناحیه (region) به خود را انتخاب کنید و به مرحله بعد بروید.

5. روی علامت checkmark کلیک کنید تا دیتابیس شما ایجاد شود. پس از آنکه دیتابیس شما ساخته شد می‌توانید از قسمت **ADD-**

ONS آن را مدیریت کنید.

6. همانطور که در تصویر بالا می‌بینید، می‌توانید اطلاعات اتصال دیتابیس (connection info) را از پایین صفحه دریافت کنید.

7. اطلاعات اتصال را با کلیک کردن روی دکمه مجاور کپی کنید تا بعدا در اپلیکیشن MVC خود از آن استفاده کنیم.

ایجاد جداول ASP.NET Identity در یک دیتابیس MySQL

ابتدا ابزار MySQL Workbench را نصب کنید.

1. ابزار مذکور را از اینجا دانلود کنید.
2. هنگام نصب، گزینه **Setup Type: Custom** را انتخاب کنید.
3. در قسمت انتخاب قابلیت ها، گزینه‌های **Applications** و **MySQLWorkbench** را انتخاب کنید و مراحل نصب را به اتمام برسانید.
4. اپلیکیشن را اجرا کرده و روی **MySQLConnection** کلیک کنید تا رشته اتصال جدیدی تعریف کنید. رشته اتصالی که در مراحل قبل از Azure MySQL Database کپی کردید را اینجا استفاده کنید. بعنوان مثال:

Connection Name

```
: AzureDB;
```

Host Name

```
: us-cdbr-azure-west-b.cleardb.com;
```

Username

```
: <username>;
```

Password

```
: <password>;
```

Default Schema

```
: IdentityMySQLDatabase
```

5. پس از برقراری ارتباط با دیتابیس، یک برگ **Query** جدید باز کنید. فرامین زیر را برای ایجاد جداول مورد نیاز کپی کنید.

```
CREATE TABLE `IdentityMySQLDatabase`.`users` (
  `Id` VARCHAR(45) NOT NULL,
  `UserName` VARCHAR(45) NULL,
  `PasswordHash` VARCHAR(100) NULL,
  `SecurityStamp` VARCHAR(45) NULL,
  PRIMARY KEY (`id`));

CREATE TABLE `IdentityMySQLDatabase`.`roles` (
  `Id` VARCHAR(45) NOT NULL,
  `Name` VARCHAR(45) NULL,
  PRIMARY KEY (`Id`));

CREATE TABLE `IdentityMySQLDatabase`.`userclaims` (
  `Id` INT NOT NULL AUTO_INCREMENT,
  `UserId` VARCHAR(45) NULL,
  `ClaimType` VARCHAR(100) NULL,
  `ClaimValue` VARCHAR(100) NULL,
  PRIMARY KEY (`Id`),
  FOREIGN KEY (`UserId`)
    REFERENCES `IdentityMySQLDatabase`.`users` (`Id`) on delete cascade);

CREATE TABLE `IdentityMySQLDatabase`.`userlogins` (
  `UserId` VARCHAR(45) NOT NULL,
  `ProviderKey` VARCHAR(100) NULL,
  `LoginProvider` VARCHAR(100) NULL,
  FOREIGN KEY (`UserId`)
    REFERENCES `IdentityMySQLDatabase`.`users` (`Id`) on delete cascade);

CREATE TABLE `IdentityMySQLDatabase`.`userroles` (
  `UserId` VARCHAR(45) NOT NULL,
  `RoleId` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`UserId`, `RoleId`),
  FOREIGN KEY (`UserId`)
    REFERENCES `IdentityMySQLDatabase`.`users` (`Id`)
on delete cascade
on update cascade,
  FOREIGN KEY (`RoleId`)
    REFERENCES `IdentityMySQLDatabase`.`roles` (`Id`)
on delete cascade
on update cascade);
```

6. حالا تمام جداول لازم برای ASP.NET Identity را در اختیار دارید، دیتابیس ما MySQL است و روی Windows Azure میزبانی شده.

ایجاد یک اپلیکیشن ASP.NET MVC و پیکربندی آن برای استفاده از MySQL Provider

1. به مخزن https://github.com/raquelsa/AspNet.Identity.MySQL بروید.
2. در گوشه سمت راست پایین صفحه روی دکمه Download Zip کلیک کنید تا کل پروژه را دریافت کنید.
3. محتوای فایل دریافتی را در یک پوشه محلی استخراج کنید.
4. پروژه AspNet.Identity.MySQL را باز کرده و آن را کامپایل (build) کنید.
5. روی نام پروژه کلیک راست کنید و گزینه **Add, New Project** را انتخاب نمایید. پروژه جدیدی از نوع ASP.NET Web Application بسازید و نام آن را به **IdentityMySQLDemo** تغییر دهید.

6. در پنجره New ASP.NET Project قالب MVC را انتخاب کنید و تنظیمات پیش فرض را بپذیرید.

7. در پنجره Solution Explorer روی پروژه IdentityMySQLDemo کلیک راست کرده و **Manage NuGet Packages** را انتخاب کنید. در قسمت جستجوی دیالوگ باز شده عبارت "*Identity.EntityFramework* " را وارد کنید. در لیست نتایج این پکیج را انتخاب کرده و آن را حذف (Uninstall) کنید. پیغامی مبنی بر حذف وابستگی‌ها باید دریافت کنید که مربوط به پکیج EntityFramework است، گزینه Yes را انتخاب کنید. از آنجا که کاری با پیاده سازی فرض نخواهیم داشت، این پکیج‌ها را حذف می‌کنیم.

8. روی پروژه IdentityMySQLDemo کلیک راست کرده و **Add, Reference, Solution, Projects** را انتخاب کنید. در دیالوگ باز شده پروژه **AspNet.Identity.MySQL** را انتخاب کرده و OK کنید.

9. در پروژه IdentityMySQLDemo پوشه Models را پیدا کرده و کلاس **IdentityModels.cs** را حذف کنید.

10. در پروژه IdentityMySQLDemo تمام ارجاعات "using Microsoft.AspNet.Identity.EntityFramework;" را با "using AspNet.Identity.MySQL;" جایگزین کنید.

11. در پروژه IdentityMySQLDemo تمام ارجاعات به کلاس "ApplicationUser" را با "IdentityUser" جایگزین کنید.

12. کنترلر Account را باز کنید و متد سازنده آنرا مطابق لیست زیر تغییر دهید.

```
public AccountController() : this(new UserManager<IdentityUser>(new UserStore(new MySQLDatabase())))
{

}
```

13. فایل web.config را باز کنید و رشته اتصال DefaultConnection را مطابق لیست زیر تغییر دهید.

```
<add name="DefaultConnection" connectionString="Database=IdentityMySQLDatabase;Data
Source=<DataSource>;User Id=<UserID>;Password=<Password>" providerName="MySql.Data.MySqlClient" />
```

مقادیر <UserId>, <DataSource> و <Password> را با اطلاعات دیتابیس خود جایگزین کنید.

اجرای اپلیکیشن و اتصال به دیتابیس MySQL

1. روی پروژه IdentityMySQLDemo کلیک راست کرده و **Set as Startup Project** را انتخاب کنید.
2. اپلیکیشن را با Ctrl + F5 کامپایل و اجرا کنید.
3. در بالای صفحه روی **Register** کلیک کنید.
4. حساب کاربری جدیدی بسازید.

5. در این مرحله کاربر جدید باید ایجاد شده و وارد سایت شود.

6. به ابزار MySQL Workbench بروید و محتوای جداول **IdentityMySQLDatabase** را بررسی کنید. جدول users را باز کنید و اطلاعات کاربر جدید را بررسی نمایید.

برای ساده نگاه داشتن این مقاله از بررسی تمام کدهای لازم خودداری شده، اما اگر مراحل را دنبال کنید و سورس کد نمونه را دریافت و بررسی کنید خواهید دید که پیاده سازی تامین کنندگان سفارشی برای ASP.NET Identity کار نسبتا ساده ای است.

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 EF Code-first 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 »

نویسنده: آذرنیوا
تاریخ: ۱۳۹۳/۱۱/۲۷ ۷:۲۴

نحوه تعیین سطح دسترسی در asp form برای متدها و فرم‌ها ها چه گونه است مثل کاری که فیلتر authorize در mvc انجام می‌ده

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۱/۲۷ ۱۰:۱۹

افزودن ASP.NET Identity به یک پروژه Web Forms

عنوان: **Identity 2.0 : تایید حساب های کاربری و احراز هویت دو مرحله ای**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۳/۰۲/۱۰ ۱:۳۰
آدرس: www.dotnettips.info
گروه‌ها: MVC, Security, ASP.NET Identity

در پست قبلی نگاهی اجمالی به انتشار نسخه جدید Identity Framework داشتیم. نسخه جدید تغییرات چشمگیری را در فریم ورک بوجود آورده و قابلیت‌های جدیدی نیز عرضه شده‌اند. دو مورد از این قابلیت‌ها که پیشتر بسیار درخواست شده بود، تایید حساب‌های کاربری (Account Validation) و احراز هویت دو مرحله ای (Two-Factor Authorization) بود. در این پست راه اندازی این دو قابلیت را بررسی می‌کنیم.

تیم ASP.NET Identity پروژه نمونه ای را فراهم کرده است که می‌تواند بعنوان نقطه شروعی برای اپلیکیشن‌های MVC استفاده شود. پیکربندی‌های لازم در این پروژه انجام شده‌اند و برای استفاده از فریم ورک جدید آماده است.

شروع به کار : پروژه نمونه را توسط NuGet ایجاد کنید

برای شروع یک پروژه ASP.NET خالی ایجاد کنید (در دیالوگ قالب‌ها گزینه Empty را انتخاب کنید). سپس کنسول Package Manager را باز کرده و دستور زیر را اجرا کنید.

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

پس از اینکه NuGet کارش را به اتمام رساند باید پروژه ای با ساختار متداول پروژه‌های ASP.NET MVC داشته باشید. به تصویر زیر دقت کنید.

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Identity2Examples' (1 project)
Identity2Examples
Properties
References
App_Start
BundleConfig.cs
FilterConfig.cs
IdentityConfig.cs
RouteConfig.cs
Startup.Auth.cs
Content
Controllers
AccountController.cs
HomeController.cs
ManageController.cs
RolesAdminController.cs
UserAdminController.cs
fonts
Models
AccountViewModels.cs
AdminViewModel.cs
IdentityModels.cs
ManageViewModels.cs
Scripts
Views
Global.asax
packages.config
Startup.cs
Web.config
Solution Explorer
Team Explorer

همانطور که می‌بینید ساختار پروژه بسیار مشابه پروژه‌های معمول MVC است، اما آیتم‌های جدیدی نیز وجود دارند. فعلا تمرکز اصلی ما روی فایل *IdentityConfig.cs* است که در پوشه *App_Start* قرار دارد.

اگر فایل مذکور را باز کنید و کمی اسکرول کنید تعاریف دو کلاس سرویس را مشاهده می‌کنید: EmailService و SmsService.

```
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);
    }
}
```

اگر دقت کنید هر دو کلاس قرارداد IIdentityMessageService را پیاده سازی می‌کنند. می‌توانید از این قرارداد برای پیاده سازی سرویس‌های اطلاع رسانی ایمیلی، پیامکی و غیره استفاده کنید. در ادامه خواهیم دید چگونه این دو سرویس را بسط دهیم.

یک حساب کاربری مدیریتی پیش فرض ایجاد کنید

پیش از آنکه بیشتر جلو رویم نیاز به یک حساب کاربری در نقش مدیریتی داریم تا با اجرای اولیه اپلیکیشن در دسترس باشد. کلاسی بنام ApplicationDbInitializer در همین فایل وجود دارد که هنگام اجرای اولیه و یا تشخیص تغییرات در مدل دیتابیس، اطلاعاتی را Seed می‌کند.

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

    //Create User=Admin@Admin.com with password=Admin@123456 in the Admin role
    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);
        }
    }
}
```

همانطور که می‌بینید این قطعه کد ابتدا نقشی بنام Admin می‌سازد. سپس حساب کاربری ای، با اطلاعاتی پیش فرض ایجاد شده و بدین نقش منتسب می‌گردد. اطلاعات کاربر را به دلخواه تغییر دهید و ترجیحا از یک آدرس ایمیل زنده برای آن استفاده کنید.

تایید حساب‌های کاربری : چگونه کار می‌کند

بدون شک با تایید حساب‌های کاربری توسط ایمیل آشنا هستید. حساب کاربری ای ایجاد می‌کنید و ایمیلی به آدرس شما ارسال می‌شود که حاوی لینک فعالسازی است. با کلیک کردن این لینک حساب کاربری شما تایید شده و می‌توانید به سایت وارد شوید.

اگر به کنترلر AccountController در این پروژه نمونه مراجعه کنید متد Register را مانند لیست زیر می‌یابید.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);

        if (result.Succeeded)
        {
            var code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);

            var callbackUrl = Url.Action(
                "ConfirmEmail",
                "Account",
                new { userId = user.Id, code = code },
                protocol: Request.Url.Scheme);

            await UserManager.SendEmailAsync(
                user.Id,
                "Confirm your account",
                "Please confirm your account by clicking this link: <a href=\""
                + callbackUrl + "\">link</a>");

            ViewBag.Link = callbackUrl;

            return View("DisplayEmail");
        }
        AddErrors(result);
    }

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

اگر به قطعه کد بالا دقت کنید فراخوانی متد UserManager.SendEmailAsync را می‌یابید که آرگومانهایی را به آن پاس می‌دهیم. در کنترلر جاری، آبجکت UserManager یک خاصیت (Property) است که وهله ای از نوع ApplicationUserManager را باز می‌گرداند. اگر به فایل *IdentityConfig.cs* مراجعه کنید تعاریف این کلاس را خواهید یافت. در این کلاس، متد استاتیک ()Create وهله ای از ApplicationUserManager را می‌سازد که در همین مرحله سرویس‌های پیام رسانی پیکربندی می‌شوند.

```
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;
}
```

در قطعه کد بالا کلاس‌های EmailService و SmsService روی وهله ApplicationUserManager تنظیم می‌شوند.

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

درست در بالای این کدها می‌بینید که چگونه تامین کنندگان احراز هویت دو مرحله ای (مبتنی بر ایمیل و پیامک) رجیستر می‌شوند.

```
// 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}"
    });
```

تایید حساب‌های کاربری توسط ایمیل و احراز هویت دو مرحله ای توسط ایمیل و/یا پیامک نیاز به پیاده سازی هایی معتبر از قراردارد IIdentityMessageService دارند.

پیاده سازی سرویس ایمیل توسط ایمیل خودتان

پیاده سازی سرویس ایمیل نسبتا کار ساده ای است. برای ارسال ایمیل‌ها می‌توانید از اکانت ایمیل خود و یا سرویس هایی مانند SendGrid استفاده کنید. بعنوان مثال اگر بخواهیم سرویس ایمیل را طوری پیکربندی کنیم که از یک حساب کاربری Outlook استفاده کند، مانند زیر عمل خواهیم کرد.

```
public class EmailService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        // Credentials:
        var credentialUserName = "yourAccount@outlook.com";
        var sentFrom = "yourAccount@outlook.com";
        var pwd = "yourApssword";

        // Configure the client:
        System.Net.Mail.SmtpClient client =
            new System.Net.Mail.SmtpClient("smtp-mail.outlook.com");

        client.Port = 587;
        client.DeliveryMethod = System.Net.Mail.SmtpDeliveryMethod.Network;
        client.UseDefaultCredentials = false;

        // Creatte the credentials:
        System.Net.NetworkCredential credentials =
            new System.Net.NetworkCredential(credentialUserName, pwd);

        client.EnableSsl = true;
        client.Credentials = credentials;

        // Create the message:
        var mail =
            new System.Net.Mail.MailMessage(sentFrom, message.Destination);

        mail.Subject = message.Subject;
        mail.Body = message.Body;

        // Send:
        return client.SendMailAsync(mail);
    }
}
```

تنظیمات SMTP میزبان شما ممکن است متفاوت باشد اما مطمئنا می‌توانید مستندات لازم را پیدا کنید. اما در کل رویکرد مشابهی خواهید داشت.

پیاده سازی سرویس ایمیل با استفاده از SendGrid

سرویس‌های ایمیل متعددی وجود دارند اما یکی از گزینه‌های محبوب در جامعه دات نت SendGrid است. این سرویس API قدرتمندی برای زبان‌های برنامه نویسی مختلف فراهم کرده است. همچنین یک Web API مبتنی بر HTTP نیز در دسترس است. قابلیت دیگر اینکه این سرویس مستقیما با Windows Azure یکپارچه می‌شود.

می توانید در سایت SendGrid یک حساب کاربری رایگان بعنوان توسعه دهنده بسازید. پس از آن پیکربندی سرویس ایمیل با مرحله قبل تفاوت چندانی نخواهد داشت. پس از ایجاد حساب کاربری توسط تیم پشتیبانی SendGrid با شما تماس گرفته خواهد شد تا از صحت اطلاعات شما اطمینان حاصل شود. برای اینکار چند گزینه در اختیار دارید که بهترین آنها ایجاد یک اکانت ایمیل در دامنه وب سایتتان است. مثلا اگر هنگام ثبت نام آدرس وب سایت خود را www.yourwebsite.com وارد کرده باشید، باید ایمیلی مانند info@yourwebsite.com ایجاد کنید و توسط ایمیل فعالسازی آن را تایید کند تا تیم پشتیبانی مطمئن شود صاحب امتیاز این دامنه خودتان هستید.

تنها چیزی که در قطعه کد بالا باید تغییر کند اطلاعات حساب کاربری و تنظیمات SMTP است. توجه داشته باشید که نام کاربری و آدرس فرستنده در اینجا متفاوت هستند. در واقع می‌توانید از هر آدرسی بعنوان آدرس فرستنده استفاده کنید.

```
public class EmailService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        // Credentials:
        var sendGridUserName = "yourSendGridUserName";
        var sentFrom = "whateverEmailAdressYouWant";
        var sendGridPassword = "YourSendGridPassword";

        // Configure the client:
        var client =
            new System.Net.Mail.SmtpClient("smtp.sendgrid.net", Convert.ToInt32(587));

        client.Port = 587;
        client.DeliveryMethod = System.Net.Mail.SmtpDeliveryMethod.Network;
        client.UseDefaultCredentials = false;

        // Creatte the credentials:
        System.Net.NetworkCredential credentials =
            new System.Net.NetworkCredential(credentialUserName, pwd);

        client.EnableSsl = true;
        client.Credentials = credentials;

        // Create the message:
        var mail =
            new System.Net.Mail.MailMessage(sentFrom, message.Destination);

        mail.Subject = message.Subject;
        mail.Body = message.Body;

        // Send:
        return client.SendMailAsync(mail);
    }
}
```

حال می‌توانیم سرویس ایمیل را تست کنیم.

آزمایش تایید حساب‌های کاربری توسط سرویس ایمیل

ابتدا اپلیکیشن را اجرا کنید و سعی کنید یک حساب کاربری جدید ثبت کنید. دقت کنید که از آدرس ایمیلی زنده که به آن دسترسی دارید استفاده کنید. اگر همه چیز بدرستی کار کند باید به صفحه ای مانند تصویر زیر هدایت شوید.

همانطور که مشاهده می‌کنید پاراگرافی در این صفحه وجود دارد که شامل لینک فعالسازی است. این لینک صرفا جهت تسهیل کار توسعه دهندگان درج می‌شود و هنگام توزیع اپلیکیشن باید آن را حذف کنید. در ادامه به این قسمت باز می‌گردیم. در این مرحله ایمیلی حاوی لینک فعالسازی باید برای شما ارسال شده باشد.

پیاده سازی سرویس SMS

برای استفاده از احراز هویت دو مرحله ای پیامکی نیاز به یک فراهم کننده SMS دارید، مانند Twilio . مانند SendGrid این سرویس نیز در جامعه دات نت بسیار محبوب است و یک C# API قدرتمند ارائه می‌کند. می‌توانید حساب کاربری رایگانی بسازید و شروع به کار کنید.

پس از ایجاد حساب کاربری یک شماره SMS، یک شناسه SID و یک شناسه Auth Token به شما داده می‌شود. شماره پیامکی خود را می‌توانید پس از ورود به سایت و پیمایش به صفحه Numbers مشاهده کنید.

شناسه‌های SID و Auth Token نیز در صفحه Dashboard قابل مشاهده هستند.

اگر دقت کنید کنار شناسه Auth Token یک آیکون قفل وجود دارد که با کلیک کردن روی آن شناسه مورد نظر نمایان می‌شود.

حال می‌توانید از سرویس Twilio در اپلیکیشن خود استفاده کنید. ابتدا بسته NuGet مورد نیاز را نصب کنید.

```
PM> Install-Package Twilio
```

پس از آن فضای نام Twilio را به بالای فایل *IdentityConfig.cs* اضافه کنید و سرویس پیامک را پیاده سازی کنید.

```
public class SmsService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        string AccountSid = "YourTwilioAccountSID";
        string AuthToken = "YourTwilioAuthToken";
        string twilioPhoneNumber = "YourTwilioPhoneNumber";

        var twilio = new TwilioRestClient(AccountSid, AuthToken);

        twilio.SendSmsMessage(twilioPhoneNumber, message.Destination, message.Body);

        // Twilio does not return an async Task, so we need this:
        return Task.FromResult(0);
```

```
    }
}
```

حال که سرویس‌های ایمیل و پیامک را در اختیار داریم می‌توانیم احراز هویت دو مرحله ای را تست کنیم.

آزمایش احراز هویت دو مرحله ای

پروژه نمونه جاری طوری پیکربندی شده است که احراز هویت دو مرحله ای اختیاری است و در صورت لزوم می‌تواند برای هر کاربر بصورت جداگانه فعال شود. ابتدا توسط حساب کاربری مدیر، یا حساب کاربری ای که در قسمت تست تایید حساب کاربری ایجاد کرده اید وارد سایت شوید. سپس در سمت راست بالای صفحه روی نام کاربری خود کلیک کنید. باید صفحه ای مانند تصویر زیر را مشاهده کنید.

در این قسمت باید احراز هویت دو مرحله ای را فعال کنید و شماره تلفن خود را ثبت نمایید. پس از آن یک پیام SMS برای شما ارسال خواهد شد که توسط آن می‌توانید پروسه را تایید کنید. اگر همه چیز بدرستی کار کند این مراحل چند ثانیه بیشتر نباید زمان بگیرد، اما اگر مثلا بیش از 30 ثانیه زمان برد احتمالا اشکالی در کار است.

حال که احراز هویت دو مرحله ای فعال شده از سایت خارج شوید و مجددا سعی کنید به سایت وارد شوید. در این مرحله یک

انتخاب به شما داده می‌شود. می‌توانید کد احراز هویت دو مرحله ای خود را توسط ایمیل یا پیامک دریافت کنید.

پس از اینکه گزینه خود را انتخاب کردید، کد احراز هویت دو مرحله ای برای شما ارسال می‌شود که توسط آن می‌توانید پروسه ورود به سایت را تکمیل کنید.

حذف میانبرهای آزمایشی

همانطور که گفته شد پروژه نمونه شامل میانبرهایی برای تسهیل کار توسعه دهندگان است. در واقع اصلا نیازی به پیاده سازی سرویس‌های ایمیل و پیامک ندارید و می‌توانید با استفاده از این میانبرها حساب‌های کاربری را تایید کنید و کدهای احراز هویت دو مرحله ای را نیز مشاهده کنید. اما قطعا این میانبرها پیش از توزیع اپلیکیشن باید حذف شوند.

بدین منظور باید نماها و کدهای مربوطه را ویرایش کنیم تا اینگونه اطلاعات به کلاینت ارسال نشوند. اگر کنترلر AccountController را باز کنید و به متد Register() بروید با کد زیر مواجه خواهید شد.

```
if (result.Succeeded)
{
    var code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
    var callbackUrl =
        Url.Action("ConfirmEmail", "Account",
            new { userId = user.Id, code = code }, protocol: Request.Url.Scheme);

    await UserManager.SendEmailAsync(user.Id, "Confirm your account",
        "Please confirm your account by clicking this link: <a href=\"" + callbackUrl + "\">link</a>");

    // This should not be deployed in production:
    ViewBag.Link = callbackUrl;

    return View("DisplayEmail");
}

AddErrors(result);
```

همانطور که می‌بینید پیش از بازگشت از این متد، متغیر callbackUrl به ViewBag اضافه می‌شود. این خط را Comment کنید یا به کلی حذف نمایید.

نمایی که این متد باز می‌گرداند یعنی *DisplayEmail.cshtml* نیز باید ویرایش شود.

```
@{
    ViewBag.Title = "DEMO purpose Email Link";
}

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

<p class="text-info">
    Please check your email and confirm your email address.
</p>

<p class="text-danger">
    For DEMO only: You can click this link to confirm the email: <a href="@ViewBag.Link">link</a>
    Please change this code to register an email service in IdentityConfig to send an email.
</p>
```

متد دیگری که در این کنترلر باید ویرایش شود VerifyCode() است که کد احراز هویت دو مرحله ای را به صفحه مربوطه پاس می‌دهد.

```
[AllowAnonymous]
public async Task<ActionResult> VerifyCode(string provider, string returnUrl)
{
    // Require that the user has already logged in via username/password or external login
    if (!await SignInHelper.HasBeenVerified())
    {
        return View("Error");
    }

    var user =
        await UserManager.FindByIdAsync(await SignInHelper.GetVerifiedUserIdAsync());

    if (user != null)
    {
        ViewBag.Status =
            "For DEMO purposes the current "
            + provider
            + " code is: "
            + await UserManager.GenerateTwoFactorTokenAsync(user.Id, provider);
    }

    return View(new VerifyCodeViewModel { Provider = provider, ReturnUrl = returnUrl });
}
```

همانطور که می‌بینید متغیری بنام Status به ViewBag اضافه می‌شود که باید حذف شود.

نمای این متد یعنی *VerifyCode.cshtml* نیز باید ویرایش شود.

```
@model IdentitySample.Models.VerifyCodeViewModel

@{
    ViewBag.Title = "Enter Verification Code";
}

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

@using (Html.BeginForm("VerifyCode", "Account", new { ReturnUrl = Model.ReturnUrl }, FormMethod.Post,
new { @class = "form-horizontal", role = "form" })) {
    @Html.AntiForgeryToken()
    @Html.ValidationSummary("", new { @class = "text-danger" })
    @Html.Hidden("provider", @Model.Provider)
    <h4>@ViewBag.Status</h4>
    <hr />

    <div class="form-group">
        @Html.LabelFor(m => m.Code, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
```

```
            @Html.TextBoxFor(m => m.Code, new { @class = "form-control" })
        </div>
    </div>

    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <div class="checkbox">
                @Html.CheckBoxFor(m => m.RememberBrowser)
                @Html.LabelFor(m => m.RememberBrowser)
            </div>
        </div>
    </div>

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

در این فایل کافی است ViewBag.Status را حذف کنید.

از تنظیمات ایمیل و SMS محافظت کنید

در مثال جاری اطلاعاتی مانند نام کاربری و کلمه عبور، شناسه‌های SID و Auth Token همگی در کد برنامه نوشته شده اند. بهتر است چنین مقادیری را بیرون از کد اپلیکیشن نگاه دارید، مخصوصا هنگامی که پروژه را به سرویس کنترل ارسال می‌کند (مثلا مخازن عمومی مثل GitHub). بدین منظور می‌توانید یکی از پست‌های اخیر را مطالعه کنید.

نظرات خوانندگان

نویسنده: دات نت کدینگ
تاریخ: ۱۳۹۳/۰۲/۱۱ ۱۳:۱

سلام
مقاله مفیدی بود برای من، ظاهرا نمیتوان نام جداول ایجادی توسط identity را تغییر داد، من از متد OnModelCreating() هم کمک گرفتم ولی بی نتیجه بود. به شکل زیر

```
modelBuilder.Entity<IdentityUser>().ToTable("MyUserRoles");
```

ممنون میشوم راهنمایی نمایید.

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۲/۱۱ ۱۳:۲۶

از ef-migrations برای اعمال تغییرات انجام شده استفاده کردید؟ بدون اینکار تغییری اعمال نمیشه.

نویسنده: دات نت کدینگ
تاریخ: ۱۳۹۳/۰۲/۱۴ ۰:۳۳

منظورتون اجرا کردن migration باشه، بله،
اینکار رو انجام دادین؟ شدنیه؟

نویسنده: علی
تاریخ: ۱۳۹۳/۰۳/۰۵ ۱۴:۱۹

توی این سری کدها، اون قسمت فعال سازی توسط ایمیل هنگام ثبت نام فعال هستش. اما مسئله ای که وجود داره کاربری که ثبت نام میکنه به صورت خودکار فعال هستش. حتی اگه روی اون لینکی که به ایمیل فرستاده میشه کلیک نکنه. راحت میتونه لاگین کنه. باید کجا این مسئله چک شود و چگونه؟ ممنون

نویسنده: Sam
تاریخ: ۱۳۹۳/۰۵/۰۴ ۲۳:۳۲

من از مدل لایه ای که آقای نصیری در مباحث ان تی تی کد فرست گفتن استفاد می‌کنم ، می‌خواستم بدونم که چطور میشه کلاس‌ها user identity رو به sql server به وسیله code first و migraiton انتقال داد البته با نام‌های دلخواه جداول در sql...
ممنون میشم اگر توضیح بدید،

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۵/۰۵ ۱۰:۲

در Asp.Net Identity نباید یک DbContext جداگانه ایجاد کنید. از همان ApplicationDbContext آن جهت اضافه کردن سایر مدل‌های برنامه استفاده کنید؛ به همراه سفارشی سازی و توسعه آن. مابقی مسایل و نکات آن مانند سایر مباحث متداول EF Code first است و تفاوتی نمی‌کند.

نویسنده: سام میرزاقراچه
تاریخ: ۱۳۹۳/۰۶/۰۴ ۲۲:۳

با سلام مجدد
چگونه می‌توان در Role Manager یک Property جدید اضافه کرد و یا همان بخش Custom کردن Role در Identity 2 .....

نویسنده: ربال
تاریخ: ۱۳۹۳/۰۸/۱۸ ۰:۵۰

با سلام.آیا استفاد ه از Twilio محدودیت دارد؟ یعنی اگر تو پروژه استفاده کنیم بعدا ب مشکل بر نخواهیم خورد (از جانب رایگان بودنش)
ممنون .

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۲۹ ۱۳:۴۲

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

نویسنده: عباس معینی
تاریخ: ۱۳۹۴/۰۱/۱۵ ۱۲:۱

با سلام؛ من یک پروژه همانطور که در ابتدا توضیح دادید ایجاد کردم. وقتی کاربر جدیدی تعریف میکنم بدون اینکه روی لینک فعال سازی کلیک کنم، لاگین میشه! ممنون میشم راهنماییم کنید.

عنوان: **نگاهی به هویت سنجی کاربران در ASP.NET MVC 5**
نویسنده: مرتضی دلیل
تاریخ: ۱۳۹۳/۰۳/۲۲ ۱۱:۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, MVC, Security, ASP.NET Identity

در مقاله پیش رو، سعی شده‌است به شکلی تقریبا عملی، کلیاتی در مورد Authentication در MVC5 توضیح داده شود. هدف روشن شدن ابهامات اولیه در هویت سنجی MVC5 و حل شدن مشکلات اولیه برای ایجاد یک پروژه است. در MVC 4 برای دسترسی به جداول مرتبط با اعتبار سنجی (مثلا لیست کاربران) مجبور به استفاده از متدهای از پیش تعریف شده‌ی رفرنس‌هایی که برای آن نوع اعتبار سنجی وجود داشت، بودیم. راه حلی نیز برای دسترسی بهتر وجود داشت و آن هم ساختن مدل‌های مشابه آن جدول‌ها و اضافه کردن چند خط کد به برنامه بود. با اینکار دسترسی ساده به Roles و Users برای تغییر و اضافه کردن محتوای آنها ممکن می‌شد. در لینک زیر توضیحاتی در مورد روش اینکار وجود دارد.
[اینجا]

در MVC5 داستان کمی فرق کرده است. برای درک موضوع پروژه ای بسازید و حالت پیش فرض آن را تغییر ندهید و آن را اجرا کنید و ثبت نام را انجام دهید، بلافاصله تصویر زیر در دیتابیس نمایان خواهد شد.

دقت کنید بعد از ایجاد پروژه در MVC5 دو پکیج بصورت اتوماتیک از طریق Nuget به پروژه شما اضافه میشود:

```
Microsoft.AspNet.Identity.Core
Microsoft.AspNet.Identity.EntityFrameWork
```

عامل اصلی تغییرات جدید، همین دو پکیج فوق است.
اولین پکیج شامل اینترفیس‌های IUser و IRole است که شامل فیلدهای مرتبط با این دو می‌باشد. همچنین اینترفیسی به نام IUserStore وجود دارد که چندین متد داشته و وظیفه اصلی هر نوع اضافه و حذف کردن یا تغییر در کاربران، بر دوش آن است.

دومین پکیج هم وظیفه پیاده سازی آن‌چیزی را دارد که در پکیج اول معرفی شده است. کلاس‌های موجود در این پکیج ابزارهایی برای ارتباط EntityFramework با دیتابیس هستند.
اما از مقدمات فوق که بگذریم برای درک بهتر رفتار با دیتابیس یک مثال را پیاده سازی خواهیم کرد.

فرض کنید میخواهیم چنین ارتباطی را بین سه جدول در دیتابیس برقرار کنیم، فقط به منظور یادآوری، توجه کنید که جدول ASPNetUsers جدولی است که به شکل اتوماتیک پیش از این تولید شد و ما قرار است به کمک یک جدول واسط (AuthorProduct) آن را به جدول Product مرتبط سازیم تا مشخص شود هر کتاب (به عنوان محصول) به کدام کاربر (به عنوان نویسنده) مرتبط است.
بعد از اینکه مدل‌های مربوط به برنامه خود را ساختیم، اولا نیاز به ساخت کلاس کانتکست نداریم چون خود MVC5 کلاس کانتکست را دارد؛ ثانیا نیاز به ایجاد مدل برای جداول اعتبارسنجی نیست، چون کلاسی برای فیلدهای اضافی ما که علاقمندیم به جدول Users اضافه شود، از پیش تعیین گردیده است.

دو کلاسی که با فلش علامت گذاری شده اند، تنها فایل‌های موجود در پوشه مدل، بعد از ایجاد یک پروژه هستند. فایل IdentityModel را به عنوان فایل کانتکست خواهیم شناخت (چون یکی از کلاسهایش Context است). همانطور که پیش از این گفتیم با وجود این فایل نیازی به ایجاد یک کلاس مشتق شده از DbContext نیست. همانطور که در کد زیر میبینید این فایل دارای دو کلاس است:

```
namespace MyShop.Models
{
    // You can add profile data for the user by adding more properties to your ApplicationUser class,
please visit http://go.microsoft.com/fwlink/?LinkID=317594 to learn more.
    public class ApplicationUser : IdentityUser
    {
    }

    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext()
            : base("DefaultConnection")
        {
        }

}
```

کلاس اول همان کلاسی است که اگر به آن پراپرتی اضافه کنیم، بطور اتوماتیک آن پراپرتی به جدول ASPNetUsers در دیتابیس اضافه می‌شود و دیگر نگران فیلدهای نداشته‌ی جدول کاربران ASP.NET نخواهیم بود. مثلا در کد زیر چند عنوان به این جدول اضافه کرده ایم.

```
namespace MyShop.Models
{
    // You can add profile data for the user by adding more properties to your ApplicationUser class,
please visit http://go.microsoft.com/fwlink/?LinkID=317594 to learn more.
    public class ApplicationUser : IdentityUser
    {
        [Display(Name = "نام انگلیسی")]
        public string EnglishName { get; set; }

        [Display(Name = "نام سیستمی")]
        public string NameInSystem { get; set; }

        [Display(Name = "نام فارسی")]
        public string PersianName { get; set; }

        [Required]
        [DataType(DataType.EmailAddress)]
        [Display(Name = "آدرس ایمیل")]
        public string Email { get; set; }
     }
}
```

کلاس دوم نیز محل معرفی مدلها به منظور ایجاد در دیتابیس است. به ازای هر مدل یک جدول در دیتابیس خواهیم داشت. مثلا در

شکل فوق سه پراپرتی به جدول کاربران اضافه میشود. دقت داشته باشید با اینکه هیچ مدلی برای جدول کاربران نساخته ایم اما کلاس ApplicatioUsers کلاسی است که به ما امکان دسترسی به مقادیر این جدول را می‌دهد(دسترسی به معنای اضافه و حذف وتغییر مقادیر این جدول است) (در MVC4 به کمک کلاس membership کارهای مشابهی انجام میدادیم)

در ساختن مدل هایمان نیز اگر نیاز به ارتباط با جدول کاربران باشد، از همین کلاس فوق استفاده میکنیم. کلاس واسط(مدل واسط) بین AspNetUsers و Product در کد زیر زیر نشان داده شده است :

```
namespace MyShop.Models
{
    public class AuthorProduct
    {
        [Key]
        public int AuthorProductId { get; set; }
       /* public int UserId { get; set; }*/

        [Display(Name = "User")]
        public string ApplicationUserId { get; set; }

        public int ProductID { get; set; }

        public virtual Product Product { get; set; }

        public virtual ApplicationUser ApplicationUser { get; set; }
    }
}
```

همانطور که مشاهده میکنید، به راحتی ارتباط را برقرار کردیم و برای برقراری این ارتباط از کلاس ApplicationUser استفاده کردیم. پراپرتی ApplicationUserId نیز فیلد ارتباطی ما با جدول کاربران است. جدول product هم نکته خاصی ندارد و به شکل زیر مدل خواهد شد.

```
namespace MyShop.Models
{
    [DisplayName("محصول")]
    [DisplayPluralName("محصولات")]
    public class Product
    {
        [Key]
        public int ProductID { get; set; }

        [Display(Name = "گروه محصول")]
        [Required(ErrorMessage = "لطفا {0} را وارد کنید")]
        public int ProductGroupID { get; set; }

        [Display(Name = "مدت زمان")]
        public string Duration { get; set; }


        [Display(Name = "نام تهیه کننده")]
        public string Producer { get; set; }

        [Display(Name = "عنوان محصول")]
        [Required(ErrorMessage = "لطفا {0} را وارد کنید")]
        public string ProductTitle { get; set; }

        [StringLength(200)]
        [Display(Name = "کلید واژه")]
        public string MetaKeyword { get; set; }

        [StringLength(200)]
        [Display(Name = "توضیح")]
        public string MetaDescription { get; set; }

        [Display(Name = "شرح محصول")]
        [UIHint("RichText")]
        [AllowHtml]
        public string ProductDescription { get; set; }

        [Display(Name = "قیمت محصول")]
        [DisplayFormat(ApplyFormatInEditMode = true, DataFormatString = "{0:#,0 ریال}")]
        [UIHint("Integer")]
        [Required(ErrorMessage = "لطفا {0} را وارد کنید")]
        public int ProductPrice { get; set; }
        [Display(Name = "تاریخ ثبت محصول")]
```

```
        public DateTime? RegisterDate { get; set; }

    }
}
```

به این ترتیب هم ارتباطات را برقرار کرده‌ایم و هم از ساختن یک UserProfile اضافی خلاص شدیم.

برای پر کردن مقادیر اولیه نیز به راحتی از seed موجود در Configuration.cs مربوط به migration استفاده میکنیم. نمونه‌ی اینکار در کد زیر موجود است:

```
protected override void Seed(MyShop.Models.ApplicationDbContext context)
        {
            context.Users.AddOrUpdate(u => u.Id,
                        new ApplicationUser() {  Id = "1",EnglishName = "MortezaDalil", PersianName =
"مرتضی دلیل", UserDescription = "توضیح در مورد مرتضی", Email = "mm@mm.com", Phone = "2323", Address =
"test", NationalCode = "2222222222", ZipCode = "2222222222" },
                                new ApplicationUser() { Id = "2", EnglishName = "MarhamatZeinali",
PersianName = "محسن احمدی", UserDescription = "توضیح در مورد محسن", Email = "mm@mm.com", Phone =
"2323", Address = "test", NationalCode = "2222222222", ZipCode = "2222222222" },
                                new ApplicationUser() { Id = "3", EnglishName = "MahdiMilani", PersianName
= "مهدی محمدی", UserDescription = "توضیح در مورد مهدی", Email = "mm@mm.com", Phone = "2323", Address
= "test", NationalCode = "2222222222", ZipCode = "2222222222" },
                                new ApplicationUser() { Id = "4", EnglishName = "Babak", PersianName =
"بابک", UserDescription = "کاربر معمولی بدون توضیح", Email = "mm@mm.com", Phone = "2323", Address =
"test", NationalCode = "2222222222", ZipCode = "2222222222" }

                          );


            context.AuthorProducts.AddOrUpdate(u => u.AuthorProductId,
              new AuthorProduct() { AuthorProductId = 1, ProductID = 1, ApplicationUserId = "2" },
              new AuthorProduct() { AuthorProductId = 2, ProductID = 2, ApplicationUserId = "1" },
              new AuthorProduct() { AuthorProductId = 3, ProductID = 3, ApplicationUserId = "3" }

          );
```

می‌توانیم از کلاس‌های خود Identity برای انجام روش فوق استفاده کنیم؛ فرض کنید بخواهیم یک کاربر به نام admin و با نقش admin به سیستم اضافه کنیم.

```
if (!context.Users.Where(u => u.UserName == "Admin").Any())
            {
                var roleStore = new RoleStore<IdentityRole>(context);
                var rolemanager = new RoleManager<IdentityRole>(roleStore);

                var userstore = new UserStore<ApplicationUser>(context);
                var usermanager = new UserManager<ApplicationUser>(userstore);

                var user = new ApplicationUser {UserName = "Admin"};

                usermanager.Create(user, "121212");
                rolemanager.Create(new IdentityRole {Name = "admin"});

                usermanager.AddToRole(user.Id, "admin");
            }
```

در عبارت شرطی موجود کد فوق، ابتدا چک کردیم که چنین یوزری در دیتابیس نباشد، سپس از کلاس RoleStore که پیاده سازی شده‌ی اینترفیس IRoleStore است استفاده کردیم. سازنده این کلاس به کانتکست نیاز دارد؛ پس به آن context را به عنوان ورودی می‌دهیم. در خط بعد، کلاس rolemanager را داریم که بخشی از پکیج Core است و پیش از این درباره اش توضیح دادیم (یکی از دو رفرنسی که خوبخود به پروژه اضافه میشوند) و از ویژگی‌های Identity است. به آن آبجکتی که از RoleStore ساختیم را پاس میدهیم و خود کلاس میداند چه چیز را کجا ذخیره کند.

برای ایجاد کاربر نیز همین روند را انجام می‌دهیم. سپس یک آبجکت به نام user را از روی کلاس ApplicationUser میسازیم. برای آن پسورد 121212 سِت میکنیم و نقش ادمین را به آن نسبت میدهیم. این روش قابل تسری به تمامی بخش‌های برنامه شماست. میتوانید عملیات کنترل و مدیریت اکانت را نیز به همین شکل انجام دهید. ساخت کاربر و لاگین کردن یا مدیریت پسورد

نیز به همین شکل قابل انجام است.

بعد از آپدیت دیتابیس تغییرات را مشاهده خواهیم کرد.

نظرات خوانندگان

نویسنده: هما
تاریخ: ۱۳۹۴/۰۳/۱۲ ۲۰:۳۲

چطور میشه برای role کاربر تعریف کنیم. بعد برای هر کاربری که مدیر خواست چند کار غیر فعال کنیم. سطح دسترسی‌های مختلف منظورم هست.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۱۲ ۲۰:۳۶

یک پروژه‌ی خالی را ایجاد کنید. سپس این بسته‌ی نیوگت را نصب کنید:

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

در این مثال رسمی، صفحات افزودن role، کاربر و غیره وجود دارد. از آن ایده بگیرید.

نویسنده: هما
تاریخ: ۱۳۹۴/۰۳/۱۳ ۱۸:۳

تو این پروژه اینکه برای هر رول بیایم کار یا action تعریف کنیم وجود نداره
منظورم اینه که مثلا یه رول تعریف که میشه براش تعریف کنیم که چه کارایی می‌تونه انجام بده و حالا ممکن برای یه کاربر خاص که رول براش تعریف شده بتونیم چند کار غیر فعال کنیم
من که قبلا از این سیستم استفاده نمی‌کردم یه جدول task و UserTask تعریف می‌کردم

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۱۳ ۱۸:۵۹

- ابتدا باید با فیلتر Authorize و نحوه‌ی طراحی یک برنامه بر اساس آن، آشنا باشید.
- سپس: « ... سفارشی سازی فیلتر Authorize از ارث بری از AuthorizeAttribute و سپس override کردن متد OnAuthorization آن شروع می‌شود ... »

نویسنده: احمد نواصری
تاریخ: ۱۳۹۴/۰۳/۱۴ ۰:۳۷

برای فهم هر چه بهتر مبحث Identity از کتاب Pro Asp.Net MVC5 Platform کمک بگیرید. مبحث Identity رو به صورت مفصل و روان توضیح داده.

عنوان: **اعمال تزریق وابستگی‌ها به مثال رسمی ASP.NET Identity**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۲۹ ۱۳:۳۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, Entity framework, MVC, Dependency Injection, ASP.NET Identity

پروژه‌ی ASP.NET Identity که نسل جدید سیستم Authentication و Authorization مخصوص ASP.NET است، دارای دو سری مثال رسمی است:

الف) مثال‌های کدپلکس

ب) مثال نیوگت

در ادامه قصد داریم مثال نیوگت آن‌را که مثال کاملی است از نحوه‌ی استفاده از ASP.NET Identity در ASP.NET MVC، جهت اعمال الگوی واحد کار و تزریق وابستگی‌ها، بازنویسی کنیم.

پیشنیازها

- برای درک مطلب جاری نیاز است ابتدا دوره‌ی مرتبطی را در سایت مطالعه کنید و همچنین با نحوه‌ی پیاده سازی الگوی واحد کار در EF Code First آشنا باشید.
- به علاوه فرض بر این است که یک پروژه‌ی خالی ASP.NET MVC 5 را نیز آغاز کرده‌اید و توسط کنسول پاور شل نیوگت، فایل‌های مثال Microsoft.AspNet.Identity.Samples را به آن افزوده‌اید:

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

ساختار پروژه‌ی تکمیلی

همانند مطلب پیاده سازی الگوی واحد کار در EF Code First ، این پروژه‌ی جدید را با چهار اسمبلی class library دیگر به نام‌های

AspNetIdentityDependencyInjectionSample.DataLayer
AspNetIdentityDependencyInjectionSample.DomainClasses
AspNetIdentityDependencyInjectionSample.IocConfig
AspNetIdentityDependencyInjectionSample.ServiceLayer

تکمیل می‌کنیم.

ساختار پروژه‌ی AspNetIdentityDependencyInjectionSample.DomainClasses

مثال Microsoft.AspNet.Identity.Samples بر مبنای primary key از نوع string است. برای نمونه کلاس کاربران آن‌را به نام ApplicationUser در فایل Models\IdentityModels.cs می‌توانید مشاهده کنید. در مطلب جاری، این نوع پیش فرض، به نوع متداول int تغییر خواهد یافت. به همین جهت نیاز است کلاس‌های ذیل را به پروژه‌ی DomainClasses اضافه کرد:

```
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.AspNet.Identity.EntityFramework;

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
  public class ApplicationUser : IdentityUser<int, CustomUserLogin, CustomUserRole, CustomUserClaim>
  {
   // سایر خواص اضافی در اینجا

   [ForeignKey("AddressId")]
   public virtual Address Address { get; set; }
   public int? AddressId { get; set; }
  }
}

using System.Collections.Generic;
```

```
namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
  public class Address
  {
   public int Id { get; set; }
   public string City { get; set; }
   public string State { get; set; }

   public virtual ICollection<ApplicationUser> ApplicationUsers { set; get; }
  }
}

using Microsoft.AspNet.Identity.EntityFramework;

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
  public class CustomRole : IdentityRole<int, CustomUserRole>
  {
   public CustomRole() { }
   public CustomRole(string name) { Name = name; }


  }
}

using Microsoft.AspNet.Identity.EntityFramework;

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
  public class CustomUserClaim : IdentityUserClaim<int>
  {

  }
}

using Microsoft.AspNet.Identity.EntityFramework;

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
  public class CustomUserLogin : IdentityUserLogin<int>
  {

  }
}

using Microsoft.AspNet.Identity.EntityFramework;

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
  public class CustomUserRole : IdentityUserRole<int>
  {

  }
}
```

در اینجا نحوه‌ی تغییر primary key از نوع string را به نوع int، مشاهده می‌کنید. این تغییر نیاز به اعمال به کلاس‌های کاربران و همچنین نقش‌های آن‌ها نیز دارد. به همین جهت صرفا تغییر کلاس ابتدایی ApplicationUser کافی نیست و باید کلاس‌های فوق را نیز اضافه کرد و تغییر داد.

بدیهی است در اینجا کلاس پایه کاربران را می‌توان سفارشی سازی کرد و خواص دیگری را نیز به آن افزود. برای مثال در اینجا یک کلاس جدید آدرس تعریف شده‌است که ارجاعی از آن در کلاس کاربران نیز قابل مشاهده است.

سایر کلاس‌های مدل‌های اصلی برنامه که جداول بانک اطلاعاتی را تشکیل خواهند داد نیز در آینده به همین اسمبلی DomainClasses اضافه می‌شوند.

ساختار پروژه‌ی AspNetIdentityDependencyInjectionSample.DataLayer جهت اعمال الگوی واحد کار

اگر به همان فایل Models\IdentityModels.cs ابتدایی پروژه که اکنون کلاس ApplicationUser آن‌را به پروژه‌ی DomainClasses منتقل کرده‌ایم، مجددا مراجعه کنید، کلاس DbContext مخصوص ASP.NET Identity نیز در آن تعریف شده‌است:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
```

این کلاس را به پروژه‌ی DataLayer منتقل می‌کنیم و از آن به عنوان DbContext اصلی برنامه استفاده خواهیم کرد. بنابراین دیگر نیازی نیست چندین DbContext در برنامه داشته باشیم. IdentityDbContext، در اصل از DbContext مشتق شده‌است.
اینترفیس IUnitOfWork برنامه، در پروژه‌ی DataLayer چنین شکلی را دارد که نمونه‌ای از آن‌را در مطلب آشنایی با نحوه‌ی پیاده سازی الگوی واحد کار در EF Code First ، پیشتر ملاحظه کرده‌اید.

```
using System.Collections.Generic;
using System.Data.Entity;

namespace AspNetIdentityDependencyInjectionSample.DataLayer.Context
{
  public interface IUnitOfWork
  {
   IDbSet<TEntity> Set<TEntity>() where TEntity : class;
   int SaveAllChanges();
   void MarkAsChanged<TEntity>(TEntity entity) where TEntity : class;
   IList<T> GetRows<T>(string sql, params object[] parameters) where T : class;
   IEnumerable<TEntity> AddThisRange<TEntity>(IEnumerable<TEntity> entities) where TEntity : class;
   void ForceDatabaseInitialize();
  }
}
```

اکنون کلاس ApplicationDbContext منتقل شده به DataLayer یک چنین امضایی را خواهد یافت:

```
public class ApplicationDbContext :
  IdentityDbContext<ApplicationUser, CustomRole, int, CustomUserLogin, CustomUserRole,
CustomUserClaim>,
  IUnitOfWork
{
  public DbSet<Category> Categories { set; get; }
  public DbSet<Product> Products { set; get; }
  public DbSet<Address> Addresses { set; get; }
```

تعریف آن باید جهت اعمال کلاس‌های سفارشی سازی شده‌ی کاربران و نقش‌های آن‌ها برای استفاده از primary key از نوع int به شکل فوق، تغییر یابد. همچنین در انتهای آن مانند قبل، IUnitOfWork نیز ذکر شده‌است. پیاده سازی کامل این کلاس را از پروژه‌ی پیوست انتهای بحث می‌توانید دریافت کنید.
کار کردن با این کلاس، هیچ تفاوتی با DbContextهای متداول EF Code First ندارد و تمام اصول آن‌ها یکی است.

در ادامه اگر به فایل App_Start\IdentityConfig.cs مراجعه کنید، کلاس ذیل در آن قابل مشاهده‌است:

```
public class ApplicationDbInitializer : DropCreateDatabaseIfModelChanges<ApplicationDbContext>
```

نیازی به این کلاس به این شکل نیست. آن‌را حذف کنید و در پروژه‌ی DataLayer، کلاس جدید ذیل را اضافه نمائید:

```
using System.Data.Entity.Migrations;

namespace AspNetIdentityDependencyInjectionSample.DataLayer.Context
{
  public class Configuration : DbMigrationsConfiguration<ApplicationDbContext>
  {
   public Configuration()
   {
    AutomaticMigrationsEnabled = true;
    AutomaticMigrationDataLossAllowed = true;
   }
  }
}
```

در این مثال، بحث migrations به حالت خودکار تنظیم شده‌است و تمام تغییرات در پروژه‌ی DomainClasses را به صورت خودکار به بانک اطلاعاتی اعمال می‌کند. تا همینجا کار تنظیم DataLayer به پایان می‌رسد.

ساختار پروژه‌ی AspNetIdentityDependencyInjectionSample.ServiceLayer

در ادامه مابقی کلاس‌های موجود در فایل App_Start\IdentityConfig.cs را به لایه سرویس برنامه منتقل خواهیم کرد. همچنین برای آن‌ها یک سری اینترفیس جدید نیز تعریف می‌کنیم، تا تزریق وابستگی‌ها به نحو صحیحی صورت گیرد. اگر به فایل‌های کنترلر این مثال پیش فرض مراجعه کنید (پیش از تغییرات بحث جاری)، هرچند به نظر در کنترلرها، کلاس‌های موجود در فایل App_Start\IdentityConfig.cs تزریق شده‌اند، اما به دلیل عدم استفاده از اینترفیس‌ها، وابستگی کاملی بین جزئیات پیاده سازی این کلاس‌ها و نمونه‌های تزریق شده به کنترلرها وجود دارد و عملا معکوس سازی واقعی وابستگی‌ها رخ نداده‌است. بنابراین نیاز است این مسایل را اصلاح کنیم.

الف) انتقال کلاس ApplicationUserManager به لایه سرویس برنامه

کلاس ApplicationUserManager فایل App_Start\IdentityConfig.c را به لایه سرویس منتقل می‌کنیم:

```
using System;
using System.Security.Claims;
using System.Threading.Tasks;
using AspNetIdentityDependencyInjectionSample.DomainClasses;
using AspNetIdentityDependencyInjectionSample.ServiceLayer.Contracts;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.DataProtection;

namespace AspNetIdentityDependencyInjectionSample.ServiceLayer
{
  public class ApplicationUserManager
   : UserManager<ApplicationUser, int>, IApplicationUserManager
  {
   private readonly IDataProtectionProvider _dataProtectionProvider;
   private readonly IIdentityMessageService _emailService;
   private readonly IApplicationRoleManager _roleManager;
   private readonly IIdentityMessageService _smsService;
   private readonly IUserStore<ApplicationUser, int> _store;

   public ApplicationUserManager(IUserStore<ApplicationUser, int> store,
    IApplicationRoleManager roleManager,
    IDataProtectionProvider dataProtectionProvider,
    IIdentityMessageService smsService,
    IIdentityMessageService emailService)
    : base(store)
   {
    _store = store;
    _roleManager = roleManager;
    _dataProtectionProvider = dataProtectionProvider;
    _smsService = smsService;
    _emailService = emailService;

    createApplicationUserManager();
   }


   public void SeedDatabase()
   {
   }

   private void createApplicationUserManager()
   {
    // Configure validation logic for usernames
    this.UserValidator = new UserValidator<ApplicationUser, int>(this)
    {
      AllowOnlyAlphanumericUserNames = false,
      RequireUniqueEmail = true
    };

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

```
    // Configure user lockout defaults
    this.UserLockoutEnabledByDefault = true;
    this.DefaultAccountLockoutTimeSpan = TimeSpan.FromMinutes(5);
    this.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.
    this.RegisterTwoFactorProvider("PhoneCode", new PhoneNumberTokenProvider<ApplicationUser, int>
    {
      MessageFormat = "Your security code is: {0}"
    });
    this.RegisterTwoFactorProvider("EmailCode", new EmailTokenProvider<ApplicationUser, int>
    {
      Subject = "SecurityCode",
      BodyFormat = "Your security code is {0}"
    });
    this.EmailService = _emailService;
    this.SmsService = _smsService;

    if (_dataProtectionProvider != null)
    {
      var dataProtector = _dataProtectionProvider.Create("ASP.NET Identity");
      this.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser, int>(dataProtector);
    }
   }
  }
}
```

تغییراتی که در اینجا اعمال شده‌اند، به شرح زیر می‌باشند:

- متد استاتیک Create این کلاس حذف و تعاریف آن به سازنده‌ی کلاس منتقل شده‌اند. به این ترتیب با هربار وهله سازی این کلاس توسط IoC Container به صورت خودکار این تنظیمات نیز به کلاس پایه UserManager اعمال می‌شوند.

- اگر به کلاس پایه UserManager دقت کنید، به آرگومان‌های جنریک آن یک int هم اضافه شده‌است. این مورد جهت استفاده از primary key از نوع int ضروری است.

- در کلاس پایه UserManager تعدادی متد وجود دارند. تعاریف آن‌ها را به اینترفیس IApplicationUserManager منتقل خواهیم کرد. نیازی هم به پیاده سازی این متدها در کلاس جدید ApplicationUserManager نیست؛ زیرا کلاس پایه UserManager پیشتر آن‌ها را پیاده سازی کرده‌است. به این ترتیب می‌توان به یک تزریق وابستگی واقعی و بدون وابستگی به پیاده سازی خاص UserManager رسید. کنترلری که با IApplicationUserManager بجای ApplicationUserManager کار می‌کند، قابلیت تعویض پیاده سازی آن‌را جهت آزمون‌های واحد خواهد یافت.

- در کلاس اصلی ApplicationDbInitializer پیش فرض این مثال، متد Seed هم قابل مشاهده‌است. این متد را از کلاس جدید Configuration اضافه شده به DataLayer حذف کرده‌ایم. از این جهت که در آن از متدهای کلاس ApplicationUserManager مستقیما استفاده شده‌است. متد Seed اکنون به کلاس جدید اضافه شده به لایه سرویس منتقل شده و در آغاز برنامه فراخوانی خواهد شد. DataLayer نباید وابستگی به لایه سرویس داشته باشد. لایه سرویس است که از امکانات DataLayer استفاده می‌کند.

- اگر به سازنده‌ی کلاس جدید ApplicationUserManager دقت کنید، چند اینترفیس دیگر نیز به آن تزریق شده‌اند. اینترفیس IApplicationRoleManager را ادامه تعریف خواهیم کرد. سایر اینترفیس‌های تزریق شده مانند IUserStore، IDataProtectionProvider و IIdentityMessageService جزو تعاریف اصلی ASP.NET Identity بوده و نیازی به تعریف مجدد آن‌ها نیست. فقط کلاس‌های EmailService و SmsService فایل App_Start\IdentityConfig.c را نیز به لایه سرویس منتقل کرده‌ایم. این کلاس‌ها بر اساس تنظیمات IoC Container مورد استفاده، در اینجا به صورت خودکار ترزیق خواهند شد. حالت پیش فرض آن، وهله سازی مستقیم است که مطابق کدهای فوق به حالت تزریق وابستگی‌ها بهبود یافته‌است.

ب) انتقال کلاس ApplicationSignInManager به لایه سرویس برنامه

کلاس ApplicationSignInManager فایل App_Start\IdentityConfig.c را نیز به لایه سرویس منتقل می‌کنیم.

```
using AspNetIdentityDependencyInjectionSample.DomainClasses;
using AspNetIdentityDependencyInjectionSample.ServiceLayer.Contracts;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;

namespace AspNetIdentityDependencyInjectionSample.ServiceLayer
{
  public class ApplicationSignInManager :
```

```
   SignInManager<ApplicationUser, int>, IApplicationSignInManager
  {
   private readonly ApplicationUserManager _userManager;
   private readonly IAuthenticationManager _authenticationManager;

   public ApplicationSignInManager(ApplicationUserManager userManager,
              IAuthenticationManager authenticationManager) :
    base(userManager, authenticationManager)
   {
    _userManager = userManager;
    _authenticationManager = authenticationManager;
   }
  }
}
```

در اینجا نیز اینترفیس جدید IApplicationSignInManager را برای مخفی سازی پیاده سازی کلاس پایه توکار SignInManager، اضافه کرده‌ایم. این اینترفیس دقیقا حاوی تعاریف متدهای کلاس پایه SignInManager است و نیازی به پیاده سازی مجدد در کلاس ApplicationSignInManager نخواهد داشت.

ج) انتقال کلاس ApplicationRoleManager به لایه سرویس برنامه

کلاس ApplicationRoleManager فایل App_Start\IdentityConfig.c را نیز به لایه سرویس منتقل خواهیم کرد:

```
using AspNetIdentityDependencyInjectionSample.DomainClasses;
using AspNetIdentityDependencyInjectionSample.ServiceLayer.Contracts;
using Microsoft.AspNet.Identity;

namespace AspNetIdentityDependencyInjectionSample.ServiceLayer
{
  public class ApplicationRoleManager : RoleManager<CustomRole, int>, IApplicationRoleManager
  {
   private readonly IRoleStore<CustomRole, int> _roleStore;
   public ApplicationRoleManager(IRoleStore<CustomRole, int> roleStore)
    : base(roleStore)
   {
    _roleStore = roleStore;
   }


   public CustomRole FindRoleByName(string roleName)
   {
    return this.FindByName(roleName); // RoleManagerExtensions
   }

   public IdentityResult CreateRole(CustomRole role)
   {
    return this.Create(role); // RoleManagerExtensions
   }
  }
}
```

روش کار نیز در اینجا همانند دو کلاس قبل است. اینترفیس جدید IApplicationRoleManager را که حاوی تعاریف متدهای کلاس پایه توکار RoleManager است، به لایه سرویس اضافه می‌کنیم. کنترلرهای برنامه با این اینترفیس بجای استفاده مستقیم از کلاس ApplicationRoleManager کار خواهند کرد.

تا اینجا کار تنظیمات لایه سرویس برنامه به پایان می‌رسد.

ساختار پروژه‌ی AspNetIdentityDependencyInjectionSample.IocConfig

پروژه‌ی IocConfig جایی است که تنظیمات StructureMap را به آن منتقل کرده‌ایم:

```
using System;
using System.Data.Entity;
using System.Threading;
using System.Web;
using AspNetIdentityDependencyInjectionSample.DataLayer.Context;
```

```
using AspNetIdentityDependencyInjectionSample.DomainClasses;
using AspNetIdentityDependencyInjectionSample.ServiceLayer;
using AspNetIdentityDependencyInjectionSample.ServiceLayer.Contracts;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Owin.Security;
using StructureMap;
using StructureMap.Web;

namespace AspNetIdentityDependencyInjectionSample.IocConfig
{
    public static class SmObjectFactory
    {
        private static readonly Lazy<Container> _containerBuilder =
            new Lazy<Container>(defaultContainer, LazyThreadSafetyMode.ExecutionAndPublication);

        public static IContainer Container
        {
            get { return _containerBuilder.Value; }
        }

        private static Container defaultContainer()
        {
            return new Container(ioc =>
            {
                ioc.For<IUnitOfWork>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use<ApplicationDbContext>();


ioc.For<ApplicationDbContext>().HybridHttpOrThreadLocalScoped().Use<ApplicationDbContext>();
                ioc.For<DbContext>().HybridHttpOrThreadLocalScoped().Use<ApplicationDbContext>();

                ioc.For<IUserStore<ApplicationUser, int>>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<UserStore<ApplicationUser, CustomRole, int, CustomUserLogin, CustomUserRole,
CustomUserClaim>>();

                ioc.For<IRoleStore<CustomRole, int>>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<RoleStore<CustomRole, int, CustomUserRole>>();

                ioc.For<IAuthenticationManager>()
                      .Use(() => HttpContext.Current.GetOwinContext().Authentication);

                ioc.For<IApplicationSignInManager>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use<ApplicationSignInManager>();

                ioc.For<IApplicationUserManager>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use<ApplicationUserManager>();

                ioc.For<IApplicationRoleManager>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use<ApplicationRoleManager>();

                ioc.For<IIdentityMessageService>().Use<SmsService>();
                ioc.For<IIdentityMessageService>().Use<EmailService>();
                ioc.For<ICustomRoleStore>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use<CustomRoleStore>();

                ioc.For<ICustomUserStore>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use<CustomUserStore>();

                //config.For<IDataProtectionProvider>().Use(()=> app.GetDataProtectionProvider()); //
In Startup class

                ioc.For<ICategoryService>().Use<EfCategoryService>();
                ioc.For<IProductService>().Use<EfProductService>();
            });
        }
    }
}
```

در اینجا نحوه‌ی اتصال اینترفیس‌های برنامه را به کلاس‌ها و یا نمونه‌هایی که آن‌ها را می‌توانند پیاده سازی کنند، مشاهده می‌کنید.

برای مثال IUnitOfWork به ApplicationDbContext مرتبط شده‌است و یا دوبار تعاریف متناظر با DbContext را مشاهده می‌کنید. از این تعاریف به صورت توکار توسط ASP.NET Identity زمانیکه قرار است UserStore و RoleStore را وهله سازی کند، استفاده می‌شوند و ذکر آن‌ها الزامی است.

در تعاریف فوق یک مورد را به فایل Startup.cs موکول کرده‌ایم. برای مشخص سازی نمونه‌ی پیاده سازی کننده‌ی IDataProtectionProvider نیاز است به IAppBuilder کلاس Startup برنامه دسترسی داشت. این کلاس آغازین Owin اکنون به نحو ذیل بازنویسی شده‌است و در آن، تنظیمات IDataProtectionProvider را به همراه وهله سازی CreatePerOwinContext مشاهده می‌کنید:

```
using System;
using AspNetIdentityDependencyInjectionSample.IocConfig;
using AspNetIdentityDependencyInjectionSample.ServiceLayer.Contracts;
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.DataProtection;
using Owin;
using StructureMap.Web;

namespace AspNetIdentityDependencyInjectionSample
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            configureAuth(app);
        }

        private static void configureAuth(IAppBuilder app)
        {
            SmObjectFactory.Container.Configure(config =>
            {
                config.For<IDataProtectionProvider>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use(()=> app.GetDataProtectionProvider());
            });
            SmObjectFactory.Container.GetInstance<IApplicationUserManager>().SeedDatabase();

            // Configure the db context, user manager and role manager to use a single instance per
request
            app.CreatePerOwinContext(() =>
SmObjectFactory.Container.GetInstance<IApplicationUserManager>());

            // 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 =
SmObjectFactory.Container.GetInstance<IApplicationUserManager>().OnValidateIdentity()
                }
            });
            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);
        }

    }
```

```
}
```

این تعاریف از فایل پیش فرض Startup.Auth.cs پوشه‌ی App_Start دریافت و جهت کار با IoC Container برنامه، بازنویسی شده‌اند.

تنظیمات برنامه‌ی اصلی ASP.NET MVC، جهت اعمال تزریق وابستگی‌ها

الف) ابتدا نیاز است فایل Global.asax.cs را به نحو ذیل بازنویسی کنیم:

```
using System;
using System.Data.Entity;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using AspNetIdentityDependencyInjectionSample.DataLayer.Context;
using AspNetIdentityDependencyInjectionSample.IocConfig;
using StructureMap.Web.Pipeline;

namespace AspNetIdentityDependencyInjectionSample
{
    public class MvcApplication : HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);


            setDbInitializer();
            //Set current Controller factory as StructureMapControllerFactory
            ControllerBuilder.Current.SetControllerFactory(new StructureMapControllerFactory());
        }

        protected void Application_EndRequest(object sender, EventArgs e)
        {
            HttpContextLifecycle.DisposeAndClearAll();
        }

        public class StructureMapControllerFactory : DefaultControllerFactory
        {
            protected override IController GetControllerInstance(RequestContext requestContext, Type
controllerType)
            {
                if (controllerType == null)
                    throw new InvalidOperationException(string.Format("Page not found: {0}",
requestContext.HttpContext.Request.RawUrl));
                return SmObjectFactory.Container.GetInstance(controllerType) as Controller;
            }
        }

        private static void setDbInitializer()
        {
            Database.SetInitializer(new MigrateDatabaseToLatestVersion<ApplicationDbContext,
Configuration>());
            SmObjectFactory.Container.GetInstance<IUnitOfWork>().ForceDatabaseInitialize();
        }
    }
}
```

در اینجا در متد setDbInitializer، نحوه‌ی استفاده و تعریف فایل Configuration لایه Data را ملاحظه می‌کنید؛ به همراه متد آغاز بانک اطلاعاتی و اعمال تغییرات لازم به آن در ابتدای کار برنامه. همچنین ControllerFactory برنامه نیز به StructureMapControllerFactory تنظیم شده‌است تا کار تزریق وابستگی‌ها به کنترلرهای برنامه به صورت خودکار میسر شود. در پایان کار هر درخواست نیز منابع Disposable رها می‌شوند.

ب) به پوشه‌ی Models برنامه مراجعه کنید. در اینجا در هر کلاسی که Id از نوع string وجود داشت، باید تبدیل به نوع int

شوند. چون primary key برنامه را به نوع int تغییر داده‌ایم. برای مثال کلاس‌های EditUserViewModel و RoleViewModel باید تغییر کنند.

ج) اصلاح کنترلرهای برنامه جهت اعمال تزریق وابستگی‌ها

اکنون اصلاح کنترلرها جهت اعمال تزریق وابستگی‌ها ساده‌است. در ادامه نحوه‌ی تغییر امضای سازنده‌های این کنترلرها را جهت استفاده از اینترفیس‌های جدید مشاهده می‌کنید:

```
[Authorize]
public class AccountController : Controller
{
    private readonly IAuthenticationManager _authenticationManager;
    private readonly IApplicationSignInManager _signInManager;
    private readonly IApplicationUserManager _userManager;
    public AccountController(IApplicationUserManager userManager,
                             IApplicationSignInManager signInManager,
                             IAuthenticationManager authenticationManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _authenticationManager = authenticationManager;
    }

    [Authorize]
public class ManageController : Controller
{
    // Used for XSRF protection when adding external logins
    private const string XsrfKey = "XsrfId";

    private readonly IAuthenticationManager _authenticationManager;
    private readonly IApplicationUserManager _userManager;
    public ManageController(IApplicationUserManager userManager, IAuthenticationManager
authenticationManager)
    {
        _userManager = userManager;
        _authenticationManager = authenticationManager;
    }

    [Authorize(Roles = "Admin")]
public class RolesAdminController : Controller
{
    private readonly IApplicationRoleManager _roleManager;
    private readonly IApplicationUserManager _userManager;
    public RolesAdminController(IApplicationUserManager userManager,
                                IApplicationRoleManager roleManager)
    {
        _userManager = userManager;
        _roleManager = roleManager;
    }


    [Authorize(Roles = "Admin")]
public class UsersAdminController : Controller
{
    private readonly IApplicationRoleManager _roleManager;
    private readonly IApplicationUserManager _userManager;
    public UsersAdminController(IApplicationUserManager userManager,
                                IApplicationRoleManager roleManager)
    {
        _userManager = userManager;
        _roleManager = roleManager;
    }
```

پس از این تغییرات، فقط کافی است بجای خواص برای مثال RoleManager سابق از فیلدهای تزریق شده در کلاس، مثلا _roleManager جدید استفاده کرد. امضای متدهای یکی است و تنها به یک search و replace نیاز دارد.
البته تعدادی اکشن متد نیز در اینجا وجود دارند که از string id استفاده می‌کنند. این‌ها را باید به int? Id تغییر داد تا با نوع primary key جدید مورد استفاده تطابق پیدا کنند.

کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید:

AspNetIdentityDependencyInjectionSample

نظرات خوانندگان

نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۳/۱۰/۰۵ ۱۶:۱۱

ممنون از مطلب شما؛
برای تغییر نام جداول تشکیل شده، درون متد OnModelCreating کد زیر را نوشتم اما با بروز رسانی دیتابیس تغییری در اسامی جداول حاصل نشد:

```
modelBuilder.Entity<ApplicationUser>().ToTable("User").Property(p => p.Id).HasColumnName("Id");
            modelBuilder.Entity<CustomUserRole>().ToTable("UserRole").HasKey(p => new { p.RoleId,
p.UserId });
            modelBuilder.Entity<CustomUserLogin>().ToTable("UserLogin").HasKey(p => new {
p.LoginProvider, p.ProviderKey, p.UserId });
            modelBuilder.Entity<CustomUserClaim>().ToTable("UserClaim").HasKey(p => p.Id).Property(p =>
p.Id).HasColumnName("UserClaimId");
            modelBuilder.Entity<CustomRole>().ToTable("Role").Property(p =>
p.Id).HasColumnName("RoleId");
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۰۵ ۱۸:۵۳

این روش پس از امتحان، جواب داد :

```
protected override void OnModelCreating(DbModelBuilder builder)
        {
            base.OnModelCreating(builder);

            builder.Entity<ApplicationUser>().ToTable("Users");
            builder.Entity<CustomRole>().ToTable("Roles");
            builder.Entity<CustomUserClaim>().ToTable("UserClaims");
            builder.Entity<CustomUserRole>().ToTable("UserRoles");
            builder.Entity<CustomUserLogin>().ToTable("UserLogins");
        }
```

اگر متد base.OnModelCreating(builder) را در انتهای کار قرار دهید، تنظیمات پیش فرض کلاس پایه IdentityDbContext (یعنی همان نام‌های قدیمی) اعمال می‌شوند و تنظیمات شما بازنویسی خواهند شد. این متد باید در ابتدای کار فراخوانی شود.

نویسنده: فخاری
تاریخ: ۱۳۹۳/۱۰/۰۸ ۱۹:۲۱

با سلام
ممنون از مطلب مفید شما. واقعا عالی بود ☻
ولی قسمتی از کد برام نامفهوم بود :

```
public ApplicationDbContext()
            : base("connectionString1")
        {
            //this.Database.Log = data => System.Diagnostics.Debug.WriteLine(data);
            //فقط تعریف شده تا یک برک پوینت در اینجا قرار داده شود برای آزمایش تعداد بار فراخوانی آن
        }

        protected override void Dispose(bool disposing)
        {
            base.Dispose(disposing);
            //فقط تعریف شده تا یک برک پوینت در اینجا قرار داده شود برای آزمایش فراخوانی آن
        }
```

مخصوصا بخش Dispose .

چون قبلا که از الگوی واحد کار استفاده می‌شد این بخش وجود نداشت !
آیا وجود این کدها که در بالا اومده الزامی است؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۰۸ ۱۹:۲۴

خیر. متد Dispose را حذف کنید (در کلاس پایه هست). وجود سازنده هم صرفا جهت ساده سازی تعریف و انتخاب رشته اتصالی تعریف شده در web.config است.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۱ ۱۳:۲۷

نکته‌ای مهم در مورد مدیریت استراکچرمپ در این مثال

اگر از **Sm** ObjectFactory مطلب فوق استفاده می‌کنید، Container آن با ObjectFactory یکی نیست یا به عبارتی ObjectFactory اطلاعی در مورد تنظیمات کلاس سفارشی **Sm** ObjectFactory ندارد. بنابراین دیگر نباید از ObjectFactory قدیمی استفاده کنید.
در این حالت هرجایی ObjectFactory قدیمی را داشتید، با SmObjectFactory.Container تعویض می‌شود.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۳/۱۰/۱۴ ۱۸:۴۶

نحوه دسترسی به کاربری که لاگین کرده چطوریه؟
من با دستور HttpContext.Current.User کار میکنم
بعضی اوقات نال بر میگردونه

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۴ ۱۹:۱۷

User.Identity را در مثال فوق جستجو کنید:

```
User.Identity.GetUserName()
User.Identity.GetUserId()
```

نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۳/۱۰/۱۴ ۱۹:۲۱

برای مشخص کردن نمونه پیاده‌سازی کننده IDataProtectionProvider در یک برنامه کنسول نیز باید از فایل Startup استفاده کرد؟ بیشتر هدفم Seed کردن دیتابیس است (مثلاً ایمپورت تعداد زیادی کاربر از طریق یک فایل و...). اینکار رو در متد SeedDatabase هم انجام دادم ولی هر بار استثنای UserId not found رو در:

```
result = this.SetLockoutEnabled(user.Id, false);
```

صادر میکنه، با گذاشتن Breakpoint متوجه شدم که برای Id صفر رو در نظر میگیره! از این جهت ترجیح دادم برای اینکار از طریق برنامه کنسول ویندوزی هم آن را تست کنم، مثل روشی که در اینجا برای ایجاد کاربر نوشته شده.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۳/۱۰/۱۵ ۱:۱۵

در قسمت تزریق وابستگی کد زیر وجود داره

```
ioc.For<IIdentityMessageService>().Use<SmsService>();
ioc.For<IIdentityMessageService>().Use<EmailService>();
```

در صورت استفاده از اینترفیس مربوطه کدوم سرویس نمونه سازی میشه؟ در صورتی که هر دو سرویس وجود داشته باشه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۵ ۱:۵۵

این مشکل اصلاح شد . باید از named instanceها در این حالت خاص استفاده شود.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۳/۱۰/۱۵ ۶:۶

با انجام اصلاحات فوق، زمانی که کاربر اقدام به تایید شماره تلفن همراه کنه با اینکه PhoneCode انتخاب میشه اما بازم قسمت ارسال ایمیل اجرا میشه.

نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۳/۱۰/۱۵ ۱۱:۴۸

از همان متد SeedDatabase استفاده کردم، مشکل این بود که در متد Create نوع پسورد از این لحاظ که حداقل باید 6 کاراکتر باشه و درست بودن ایمیل و... نیز بررسی میشه، اگر مقادیر معتبر نباشه مقدار user.Id برابر با صفر میشه.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۵ ۱۳:۲۲

نیاز به تنظیمات setter injection هم داشت که اضافه شد .

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۳/۱۰/۱۵ ۲۰:۲۳

با این تنظیمات

```
// map same interface to different concrete classes
                ioc.For<IIdentityMessageService>().Use<SmsService>().Named("smsService");
                ioc.For<IIdentityMessageService>().Use<EmailService>().Named("emailService");
                ioc.For<IApplicationUserManager>().HybridHttpOrThreadLocalScoped()
                    .Use<ApplicationUserManager>()
                    .Ctor<IIdentityMessageService>("smsService").Is<SmsService>()
                    .Ctor<IIdentityMessageService>("emailService").Is<EmailService>()
                    .Setter<IIdentityMessageService>(userManager =>
userManager.SmsService).Is<SmsService>()
                    .Setter<IIdentityMessageService>(userManager =>
userManager.EmailService).Is<EmailService>();
```

و این کد سازنده UserManager

```
public ApplicationUserManager(
            IApplicationUserStore store,
            IApplicationRoleManager roleManager,
            IDataProtectionProvider dataProtectionProvider,
            IIdentityMessageService smsService,
            IIdentityMessageService emailService)
            : base(store as IUserStore<User,int>)
        {
            this.roleManager = roleManager;
            this.dataProtectionProvider = dataProtectionProvider;
            EmailService = emailService;
            SmsService = smsService;
            Debug.WriteLine("Ticks = {0}",DateTime.Now.Ticks);
            Debug.WriteLine(emailService.ToString());
            Debug.WriteLine(smsService.ToString());
            CreateApplicationUserManager();
        }
```

در زمان دیباگ اینجور خروجی توی پنجره دیباگ گرفتم

```
Ticks = 635560859158196052
PWS.ServiceLayer.EF.Identity.EmailService
PWS.ServiceLayer.EF.Identity.SmsService
Ticks = 635560859158246098
PWS.ServiceLayer.EF.Identity.EmailService
PWS.ServiceLayer.EF.Identity.EmailService
```

نمی‌دونم چرا دوبار سازنده فراخوانی شده در بار اول تزریق وابستگی درست انجام شده اما در اخرین بار هم سرویس ایمیل به ایمیل و پیامک هر تزریق شده.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۶ ۲:۴۵

جهت مدیریت تک وهله‌ای ApplicationUserManger این تغییرات را اعمال کنید؛ یا از این فایل نهایی استفاده کنید.

نویسنده: نیکناز
تاریخ: ۱۳۹۳/۱۰/۲۳ ۱۲:۳۴

سلام؛ در روش برنامه نویسی لایه ای به این صورتی که شما آموزش دادید امکان دیدن جدول‌های بانک در بخش Server Explorer نیست؟ منظورم قبل از اینزرت اطلاعات در آنهاست؟ چون می‌خوام یکسری از جداول رو دستی مستقیم در بانک پر کنم.
و اینکه بعد از اینکه dbContext را در پوشه App_Data حذف کردم چه جوری اتصال کاملشو از بین ببرم چون برنامه ام ارور پیدا نکردن dbcontext را می‌ده و چه جوری می‌تونم دوباره ReCreate کنمش.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۲۳ ۱۳:۲

- در سازنده‌ی کلاس ApplicationDbContext ، به connectionString1 اشاره شده که تعریف آن در فایل web.config برنامه موجود است. به عبارتی در این مثال چنین رشته‌ی اتصالی تعریف شده‌است:

```
<add name="connectionString1"
connectionString="Data Source=(local);Initial Catalog=TestDbIdentity;Integrated Security = true"
providerName="System.Data.SqlClient" />
```

بنابراین بانک اطلاعاتی پیش فرض آن TestDbIdentity نام دارد (جهت اتصال به آن، برای مشاهده جداول و یا تغییر و ثبت اطلاعات). این رشته اتصالی هم مخصوص SQL Server تنظیم شده‌است که می‌توانید توسط management studio و یا سایر ابزارهای مشابه، همانند قبل به آن متصل شوید.
- «در پوشه App_Data حذف کردم» ... از مثال نهایی کامل شده استفاده کنید (^) و نیازی به تکرار این مراحل نیست تا خطای یافت نشدن dbcontext را دریافت نکنید.
- برای ReCreate فقط کافی هست که بانک اطلاعاتی TestDbIdentity را drop کنید. بعد برنامه را مجددا از نو اجرا کنید. چون مراحل migrations آن به حالت خودکار تنظیم شده‌است، بانک اطلاعاتی را به صورت خودکار ایجاد می‌کند یا تغییرات کلاس‌های دومین برنامه را به صورت خودکار به بانک اطلاعاتی اعمال خواهد کرد.
- برای آشنایی بیشتر با مباحث تعریف رشته اتصالی EF Code First و مباحث Migrations آن، سری EF Code First را در سایت یکبار مطالعه کنید.

نویسنده: اقبال
تاریخ: ۱۳۹۳/۱۰/۲۷ ۱۶:۱۰

ممنون از مطالب شما اگر بخواهیم از IApplicationRoleManager و IApplicationUserManager یا به طور کلی از ایترفیس‌ها در AuthorizeAttribute سفارشی (یا فیلتر هایی) که ایجاد کرده ایم استفاده کنیم ، تنظیمات strucutremap به چه صورت خواهد بود.

```
public class CustomAuthorizeAttribute : AuthorizeAttribute
    {


        public IApplicationRoleManager _roleManager { get; set; }

        public IApplicationUserManager _userManager { get; set; }


        public CustomAuthorizeAttribute(IApplicationUserManager userManager,
                                        IApplicationRoleManager roleManager)
        {
            _userManager = userManager;
            _roleManager = roleManager;
        }
        protected override bool AuthorizeCore(HttpContextBase httpContext)
        {
            // Put your custom logic here, returning true for success and false for failure,
            // or return base.AuthorizeCore(httpContext) to defer to the base implementation
            IPrincipal user = httpContext.User;
            IIdentity identity = user.Identity;

            if (!identity.IsAuthenticated)
            {
                return false;
            }

            bool isAuthorized = true;
            // TODO: perform custom authorization against the App
            var qry = _userManager.Users.Where(u => u.UserName == identity.Name).ToList();

            return isAuthorized;
        }
    }
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۲۷ ۱۷:۴۱

« تزریق وابستگی‌ها در فیلترهای ASP.NET MVC »

نویسنده: م سلیمانی
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۲:۳۳

با سلام؛ پروژه این مثال و دانلود کردم و سپس اجراش کردم - مشکلی نداره ولی بعد چند دقیقه ناگهان با این پیغام مواجه میشم.

لطفا راهنمایی بفرمائید. با تشکر.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۳:۲۳

- لطفا خطاها را تصویری ارسال نکنید. متن موجود در تصاویر، قابلیت جستجو و پیگیری ندارند.
- متن آن واضح است و به عمد تنظیم شده‌است. عنوان کرده‌است که آدرس خاصی را یافت نمی‌کند و این آدرس‌ها قابلیت تزریق وابستگی ندارند. آدرس آن (arterySignalR/ping) مرتبط است به browser link که به این صورت می‌توانید آن‌را غیرفعال کنید:

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.IgnoreRoute("{*browserlink}", new { browserlink = @".*/arterySignalR/ping" });
    //...
}
```

و یا

```
<appSettings>
    <add key="vs:EnableBrowserLink" value="false" />
</appSettings>
```

نویسنده: س محمدرضا برنتی
تاریخ: ۱۳۹۴/۰۱/۱۶ ۱۶:۳۸

تنظیمات جهت اعمال تزریق وابستگی‌ها در WEB API چگونه خواهد بود ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۱/۱۶ ۱۶:۵۷

تنظیمات اولیه‌ی IoC Container در این حالت، با نمونه‌ی موجود یکی است. فقط Web API نیاز به معرفی SmObjectFactory.Container را به GlobalConfiguration مخصوص خودش، به صورت جداگانه دارد. اطلاعات بیشتر

نویسنده: سروش شیرالی
تاریخ: ۱۳۹۴/۰۱/۲۸ ۱۸:۳۵

با سلام؛ خیلی معماری جالبی بود. اما یه سوالی ذهن مرا مشغول کرده . البته من ساختار structureMap را درست نمی‌دانم ولی اکثر IOC Containerها نمونه singlethon را از اینترفیس‌ها برمی گردانند . که اگر اینگونه باشد به معنی وجود یک Unit OF Work و DbContext بوده در کل appllication بوده که این برای یک وب اپلیکیشن به منزله یک فاجعه است. اگر ممکن است مقداری راهنمایی بفرمایید. سپاسگزارم.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۱/۲۸ ۱۹:۶

طول عمر پیش فرض اشیاء ایجاد شده‌ی توسط استراکچرمپ از نوع transient است. به این معنا که هر درخواستی از آن شیء، سبب ایجاد یک وهله‌ی جدید از آن خواهد شد. اطلاعات بیشتر

نویسنده: سروش شیرالی
تاریخ: ۱۳۹۴/۰۱/۳۰ ۱۷:۵۸

فرض کنید ما از این پروژه استفاده کنیم . اما از الگوی repository و unit of work بر روی آنها نیز استفاده کنیم. به این ترتیب IOC از یک طرف applicationDbContext را به کلاس store در identity پاس داده و از طرف دیگر به کلاس unit of work آیا به نظر شما مشکلی پیش نخواهد آمد.
با توجه به آنکه به فرموده شما IOC در هر HttpRequest فقط یک آبجکت از کلاس applicationDbContext خواهد ساخت اگر store و unit of work هر دو در همان HttpRequest اقدام به اعمال savechange کنند چه؟ اصلا چینین چیزی ممکن است؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۱/۳۰ ۱۸:۳۲

context و uow یکی هستند. قسمت‌های 11 و 12 سری EF Code first سایت را نیاز است یکبار مطالعه کنید؛ به همراه تمام نظرات آن‌ها.

نویسنده: محمد رحیم پورابراهیم
تاریخ: ۱۳۹۴/۰۲/۲۸ ۱۳:۳۰

من یک پروژه انجام شده دارم که Id از نوع string هست ولی در مثالی که شما زدید Id از نوع int هست چجوری میتونم این روشی که گفتید رو پیاده سازی کنم ولی نوع فیلد Id رو تغییر ندم، اگه بخوام تغییر بدم تمام اطلاعات دیتابیس من دستخوش تغییرات زیادی میشه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۲/۲۸ ۱۴:۱۰

- متن را یکبار مطالعه کردید؟ یکبار مثال اصلی را با مثال فوق انطباق دادید؟
- در متن ذکر شده؛ مثال اصلی آن نوع ID را از نوع string تعریف کرده‌است. در اینجا برای نزدیک شدن مثال به دنیای واقعی و آن چیزی که مرسوم است، نوع Id به int تغییر داده شد.
اگر مورد نظر شما نیست، هرجایی که int مشاهده می‌کنید، تغییرش دهید به string. مثلا:

```
ApplicationUser : IdentityUser<int, CustomUserLogin, CustomUserRole, CustomUserClaim>
```

این int را تغییر دهید به string و در هر جای دیگری که تعریف شده و تغییر کرده. این تغییرات را با انطباق مثال تغییر یافته با

نمونه‌ی اصلی که لینکش در متن هست می‌توانید پیدا کنید.

نویسنده: نرگس حقیقی
تاریخ: ۱۳۹۴/۰۳/۲۹ ۱۹:۲

چطور در پروژه DomainClass که از نوع classLibrary هست
using Microsoft.AspNet.Identity.EntityFramework را اضافه کردین؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۲۹ ۲۰:۰

- وابستگی‌های هر پروژه را با مراجعه به فایل packages.config آن می‌توانید بررسی کنید.
- بعد از مشخص شدن لیست وابستگی‌ها، فقط کافی است تک تک آن‌ها را با دستور زیر نصب کنید:

```
PM> install-package name_here
```

نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۳۱ ۱۰:۴۰

البته پنجره گرافیکی nuget هم این امکان رو میده که مجبور نباشیم تک تکشون رو نصب کنیم
به محض باز شدن پنجره ، خودش شناسایی میکنه چه بسته‌های تعریف شدند ولی موجود نیست و با کلیک بر دکمه مربوطه همه رو نصب میکنه

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۳۱ ۱۰:۵۱

بله. البته اگر فایل packages.config را به پروژه‌ی جدید مربوطه کپی کنند .

نویسنده: سروش شیرالی
تاریخ: ۱۳۹۴/۰۳/۳۱ ۱۴:۴۷

می‌خواستم بدونم اگر در یک custom authorize attribute بخواهم roleهای کاربر فعلی را بدانم باید چکار کنم؟
در سیستم membership خیلی راحت می‌نوشتم:

```
var currentUserRoles =System.Web.Security.Roles.GetRolesForUser().Select(u => u.ToLower()).ToList();
```

اما در این معماری ای که شما نوشته اید چکار باید کرد؟ آیا باید را به attribute پاس داد که فکر نمی‌کنم ممکن باشد. ممکن است قدری راهنمایی بفرمایید؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۳۱ ۱۵:۹

« تزریق وابستگی‌ها در فیلترهای ASP.NET MVC »

نویسنده: ایمان محمدی
تاریخ: ۱۳۹۴/۰۵/۱۵ ۰:۴۰

برای ایجاد دیتابیس در مثال گفته شده ، Database.Initialize به کانکشن استرینگ پاس داده شده توسط stracturemap کاری نداره و از فایل کانفیگ برای ساخت دیتابیس استفاده می‌کنه و اگه کانکشین استرینگ در فایل کانفیگ نباشه یه دیتابیس به نام connectionString1 می‌سازه.

برای بررسی، در مثال خودتون کانکشن استرینگ کانفیگ و stracturemap را با نام دیتابیس متفاوت بدید.
فکر کنم بعد از اجرای Database.Initialize یک وهله جدید از ApplicationDbContext با امضای اول اون میسازه و کاری به وهله تزریق شده نداره

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۵/۱۵ ۱۱:۵۲

قسمت « مدیریت migrations خودکار برنامه در حالت استفاده از چندین بانک اطلاعاتی » را مطالعه کنید.

نویسنده: ایمان محمدی
تاریخ: ۱۳۹۴/۰۵/۱۵ ۱۲:۵۷

```
Database.SetInitializer(new MigrateDatabaseToLatestVersion<ApplicationDbContext, Configuration>());
            SmObjectFactory.Container.GetInstance<IUnitOfWork>().ForceDatabaseInitialize();
```

به خط اول دقت نکرده بودم که برای migration کلاس ApplicationDbContext معرفی شده و اینجا باید امضای کلاس اصلاح بشه.

نویسنده: سام میرزاقراچه
تاریخ: ۱۳۹۴/۰۵/۱۵ ۱۴:۵

در لایه سرویس چگونه باید یک Edit انجام شود؟ یا با این دستور چطور میشه کار کرد : MarkAsChanged

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۵/۱۵ ۱۴:۱۴

مراجعه کنید به قسمت پروژه‌های سایت. پروژه‌های سورس باز « طراحی فریمورک برای کار با Asp.net MVC و EF به صورت NTier »، « فروشگاه اینترنتی شهر طلایی من » و « سیستم مدیریت محتوای IRIS » از همین روش الگوی واحد کار استفاده می‌کنند.

نویسنده: ایمان محمدی
تاریخ: ۱۳۹۴/۰۵/۱۸ ۱۶:۹

در مثال بالا هر لاگین کاربر حداقل 9 بار کاربر همراه با رل هاش از دیتابیس فراخوانی میشه اگر ورود دو مرحله فعال باشه حدودا 28 بار میشه ، روش پیش فرض که آی دی رو string پیاد سازی کرده رو بررسی کردم چنین مشکلی نداشت. در کلاس ApplicationUserManager متد FindByIdAsync رو بصورت زیر تغییر بدید می‌تونید تعداد فراخوانی را بررسی کنید:

```
public override Task<ApplicationUser> FindByIdAsync(int userId)
        {
                    return base.FindByIdAsync(userId);
        }

        public override Task<ApplicationUser> FindByNameAsync(string userName)
        {
            return base.FindByNameAsync(userName);
        }
```

در کدهای Identity به دفعات از متود FindByIdAsync استفاده شده و فرض بر این بوده که دفعات بعد از کش DbContext اسفاده می‌شود.

در سورس UserStore متود FindByIdAsync از یک متد واسطه به نام GetUserAggregateAsync استفاده کرده :

```
public virtual Task<TUser> FindByIdAsync(TKey userId)
        {
```

```
            ThrowIfDisposed();
            return GetUserAggregateAsync(u => u.Id.Equals(userId));
        }
     protected virtual async Task<TUser> GetUserAggregateAsync(Expression<Func<TUser, bool>> filter)
        {
            TKey id;
            TUser user;
            if (FindByIdFilterParser.TryMatchAndGetId(filter, out id))
            {
                user = await _userStore.GetByIdAsync(id).WithCurrentCulture();
            }
            else
            {
                user = await Users.FirstOrDefaultAsync(filter).WithCurrentCulture();
            }
            if (user != null)
            {
                await EnsureClaimsLoaded(user).WithCurrentCulture();
                await EnsureLoginsLoaded(user).WithCurrentCulture();
                await EnsureRolesLoaded(user).WithCurrentCulture();
            }
            return user;
        }
```

وقتی پیاده سازی آی دی براساس int باشه متد FindByIdFilterParser.TryMatchAndGetId درست عمل نمی‌کنه و به جای فراخوانی GetByIdAsync که پشت صحنه از FindAsync و قابلیت فراخوانی از کش رو داره از FirstOrDefaultAsync استفاده می‌کنه و باعث فراخوانی مجدد از دیتابیس می‌شود.

راه حلی که من استفاده کردم پیاده سازی UserStore و تحریف متد FindByIdAsync :

```
public class MyUserStore
        :UserStore<ApplicationUser, CustomRole, int, CustomUserLogin, CustomUserRole, CustomUserClaim>
    {

        private DbSet<ApplicationUser> _myUserStore;
        public MyUserStore(ApplicationDbContext context)
            : base(context)
        {
            _myUserStore = (DbSet<ApplicationUser>) context.Set<ApplicationUser>();

        }

        public override Task<ApplicationUser> FindByIdAsync(int userId)
        {
           return  _myUserStore.FindAsync(userId);
        }

    }
}
```

همچنین در کلاس SmObjectFactory کد زیر :

```
ioc.For<IUserStore<ApplicationUser, int>>()
                .HybridHttpOrThreadLocalScoped()
                .Use<UserStore<ApplicationUser, CustomRole, int, CustomUserLogin, CustomUserRole,
CustomUserClaim>>();
```

با این کد جایگزین شود :

```
ioc.For<IUserStore<ApplicationUser, int>>()
                    .HybridHttpOrThreadLocalScoped()
                 .Use<MyUserStore>();
```

نویسنده: غلامرضا ربال
تاریخ: ۱۳۹۴/۰۵/۱۹ ۱۴:۱۳

با توجه به اینکه در ورژن نهایی پروژه ای که ارائه داده اید قسمت زیر در فایل Startup.cs موجود نیست

```
appBuilder.CreatePerOwinContext(
               () =>ProjectObjectFactory.Container.GetInstance<ApplicationUserManager>());
```

و این موضوع باعث خواهد شد تا SecutiyStampValidator کار کرد صحیحی نداشته باشد.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۵/۲۲ ۱۱:۵۱

جهت اطلاع

به این پروژه پشتیبانی از Web API هم اضافه شد.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۵/۲۴ ۱۴:۲۵

جهت اطلاع

پشتیبانی از SignalR هم اضافه شد. با این تغییرات

نویسنده: م علیزاده
تاریخ: ۱۳۹۴/۰۷/۱۲ ۱۹:۵۱

باسلام، در صورت امکان در مورد این سطر توضیح بفرمایید:

```
ioc.For<IIdentity>().Use(() => (HttpContext.Current != null && HttpContext.Current.User != null) ?
HttpContext.Current.User.Identity : null);
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۷/۱۲ ۲۰:۱۹

برای توضیحات بیشتر در مورد IIdentity مراجعه کنید به مطلب « تزریق وابستگی‌های رایج ASP.NET MVC به برنامه »

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۴/۰۷/۱۴ ۴:۵۸

سلام با کد ذیل _userStore واسه من نال بر میگردونه

```
using System.Data.Entity;
using System.Threading.Tasks;
using Microsoft.AspNet.Identity.EntityFramework;
using SmartMarket.Core.Domain.Members;
using SmartMarket.Data;

namespace SmartMarket.Services.Members
{
    /// <summary>
    /// The ApplicationUserStore Class
    /// </summary>
    public class ApplicationUserStore : UserStore<User, Role, int, UserLogin, UserRole, UserClaim>,
IApplicationUserStore
    {
#region Fields (1)

        private readonly IDbSet<User> _userStore;

#endregion Fields

#region Constructors (2)

        /// <summary>
        /// Initializes a new instance of the <see cref="ApplicationUserStore" /> class.
        /// </summary>
        /// <param name="dbContext">The database context.</param>
```

```
        public ApplicationUserStore(DbContext dbContext) : base(dbContext) { }

        /// <summary>
        /// Initializes a new instance of the <see cref="ApplicationUserStore"/> class.
        /// </summary>
        /// <param name="context">The context.</param>
        public ApplicationUserStore(IdentityDbContext context)
            : base(context)
        {
            _userStore = context.Set<User>();

        }

#endregion Constructors

#region Methods (2)

// Public Methods (2)

        /// <summary>
        /// Adds to previous passwords asynchronous.
        /// </summary>
        /// <param name="user">The user.</param>
        /// <param name="password">The password.</param>
        /// <returns></returns>
        public Task AddToPreviousPasswordsAsync(User user, string password)
        {
            user.PreviousUserPasswords.Add(new PreviousPassword { UserId = user.Id, PasswordHash =
password });
            return UpdateAsync(user);
        }

        /// <summary>
        /// Finds the by identifier asynchronous.
        /// </summary>
        /// <param name="userId">The user identifier.</param>
        /// <returns></returns>
        public override Task<User> FindByIdAsync(int userId)
        {
            return Task.FromResult(_userStore.Find(userId));
        }

#endregion Methods

        /// <summary>
        /// Creates the asynchronous.
        /// </summary>
        /// <param name="user">The user.</param>
        /// <returns></returns>
        public override async Task CreateAsync(User user)
        {
            await base.CreateAsync(user);
            await AddToPreviousPasswordsAsync(user, user.PasswordHash);
        }
    }
}
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۷/۱۴ ۹:۵۳

- مثال‌های متفرقه را در این سایت مطرح نکنید. مثال بررسی و اصلاح شده در اینجا قرار دارد.
- زمانیکه یک شیء وهله سازی می‌شود، فقط بر اساس یکی از سازنده‌های آن وهله سازی خواهد شد و نه تمام آن‌ها. این‌ها اصول اولیه‌ی کار با سی شارپ هست.
- در مثال متفرقه‌ای که مطرح کردید، نیازی به تعریف همزمان DbContext و همچنین IdentityDbContext نیست و کار اضافی انجام دادید؛ چون در اصل یکی هستند و بر اساس نیاز کلاسی که از آن ارث بری شده باید IdentityDbContext متفرقه باشد و نه DbContext خام.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۷/۱۴ ۱۰:۱۸

این نکته‌ای را که عنوان کردید با استفاده از DNTProfiler بررسی کردم و اصلا چنین چیزی (28 بار فراخوانی) نیست. در پشت

صحنه از نسخه‌ی Async متد Find استفاده می‌شود (در stack trace موجود هست) و حذف آن با متد ساده‌ای که نوشته شده، یک سری از سازوکارهای داخلی ASP.NET Identity را حذف می‌کند و به صلاح نیست.

عنوان: **نحوه‌ی محاسبه‌ی هش کلمات عبور کاربران در ASP.NET Identity**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۲۸ ۱۴:۱۵
آدرس: www.dotnettips.info
گروه‌ها: Security, Cryptography, ASP.NET Identity

روش‌های زیادی برای ذخیره سازی کلمات عبور وجود دارند که اغلب آن‌ها نیز نادرست هستند. برای نمونه شاید ذخیره سازی کلمات عبور، به صورت رمزنگاری شده، ایده‌ی خوبی به نظر برسد؛ اما با دسترسی به این کلمات عبور، امکان رمزگشایی آن‌ها، توسط مهاجم وجود داشته و همین مساله می‌تواند امنیت افرادی را که در چندین سایت، از یک کلمه‌ی عبور استفاده می‌کنند، به خطر اندازد.

در این حالت هش کردن کلمات عبور ایده‌ی بهتر است. هش‌ها روش‌هایی یک طرفه هستند که با داشتن نتیجه‌ی نهایی آن‌ها، نمی‌توان به اصل کلمه‌ی عبور مورد استفاده دسترسی پیدا کرد. برای بهبود امنیت هش‌های تولیدی، می‌توان از مفهومی به نام Salt نیز استفاده نمود. Salt در اصل یک رشته‌ی تصادفی است که پیش از هش شدن نهایی کلمه‌ی عبور، به آن اضافه شده و سپس حاصل این جمع، هش خواهد شد. اهمیت این مساله در بالا بردن زمان یافتن کلمه‌ی عبور اصلی از روی هش نهایی است (توسط روش‌هایی مانند brute force یا امتحان کردن بازه‌ی وسیعی از عبارات قابل تصور).

اما واقعیت این است که حتی استفاده از یک Salt نیز نمی‌تواند امنیت بازیابی کلمات عبور هش شده را تضمین کند. برای مثال نرم افزارهایی موجود هستند که با استفاده از پرداش موازی قادرند بیش از 60 میلیارد هش را در یک ثانیه آزمایش کنند و البته این کارآیی، برای کار با هش‌های متداولی مانند MD5 و SHA1 بهینه سازی شده‌است.

روش هش کردن کلمات عبور در ASP.NET Identity

ASP.NET Identity 2.x که در حال حاضر آخرین نگارش تکامل یافته‌ی روش‌های امنیتی توصیه شده‌ی توسط مایکروسافت، برای برنامه‌های وب است، از استانداردی به نام RFC 2898 و الگوریتم PKDBF2 برای هش کردن کلمات عبور استفاده می‌کند. مهم‌ترین مزیت این روش خاص، کندتر شدن الگوریتم آن با بالا رفتن تعداد سعی‌های ممکن است؛ برخلاف الگوریتم‌هایی مانند MD5 یا SHA1 که اساسا برای رسیدن به نتیجه، در کمترین زمان ممکن طراحی شده‌اند.

PBKDF2 یا password-based key derivation function جزئی از استاندارد RSA نیز هست (PKCS #5 version 2.0). در این الگوریتم، تعداد بار تکرار، یک Salt و یک کلمه‌ی عبور تصادفی جهت بالا بردن انتروپی (بی‌نظمی) کلمه‌ی عبور اصلی، به آن اضافه می‌شوند. از تعداد بار تکرار برای تکرار الگوریتم هش کردن اطلاعات، به تعداد باری که مشخص شده‌است، استفاده می‌گردد. همین تکرار است که سبب کندشدن محاسبه‌ی هش می‌گردد. عدد معمولی که برای این حالت توصیه شده‌است، 50 هزار است.

این استاندارد در دات نت توسط کلاس Rfc2898DeriveBytes پیاده سازی شده‌است که در ذیل مثالی را در مورد نحوه‌ی استفاده‌ی عمومی از آن، مشاهده می‌کنید:

```
using System;
using System.Diagnostics;
using System.Security.Cryptography;
using System.Text;

namespace IdentityHash
{
    public static class PBKDF2
    {
        public static byte[] GenerateSalt()
        {
            using (var randomNumberGenerator = new RNGCryptoServiceProvider())
            {
                var randomNumber = new byte[32];
                randomNumberGenerator.GetBytes(randomNumber);
                return randomNumber;
            }
        }

        public static byte[] HashPassword(byte[] toBeHashed, byte[] salt, int numberOfRounds)
        {
            using (var rfc2898 = new Rfc2898DeriveBytes(toBeHashed, salt, numberOfRounds))
            {
                return rfc2898.GetBytes(32);

            }
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            var passwordToHash = "VeryComplexPassword";
            hashPassword(passwordToHash, 50000);
            Console.ReadLine();
        }

        private static void hashPassword(string passwordToHash, int numberOfRounds)
        {
            var sw = new Stopwatch();
            sw.Start();
            var hashedPassword = PBKDF2.HashPassword(
                                        Encoding.UTF8.GetBytes(passwordToHash),
                                        PBKDF2.GenerateSalt(),
                                        numberOfRounds);
            sw.Stop();
            Console.WriteLine();
            Console.WriteLine("Password to hash : {0}", passwordToHash);
            Console.WriteLine("Hashed Password : {0}", Convert.ToBase64String(hashedPassword));
            Console.WriteLine("Iterations <{0}> Elapsed Time : {1}ms", numberOfRounds,
sw.ElapsedMilliseconds);
        }
    }
}
```

شیء Rfc2898DeriveBytes برای تشکیل، نیاز به کلمه‌ی عبوری که قرار است هش شود به صورت آرایه‌ای از بایت‌ها، یک Salt و یک عدد اتفاقی دارد. این Salt توسط شیء RNGCryptoServiceProvider ایجاد شده‌است و همچنین نیازی نیست تا به صورت مخفی نگه‌داری شود. آن‌را می‌توان در فیلدی مجزا، در کنار کلمه‌ی عبور اصلی ذخیره سازی کرد. نتیجه‌ی نهایی، توسط متد rfc2898.GetBytes دریافت می‌گردد. پارامتر 32 آن به معنای 256 بیت بودن اندازه‌ی هش تولیدی است. 32 حداقل مقداری است که بهتر است انتخاب شود.

پیش فرض‌های پیاده سازی Rfc2898DeriveBytes استفاده از الگوریتم SHA1 با 1000 بار تکرار است؛ چیزی که دقیقا در ASP.NET Identity 2.x بکار رفته‌است.

تفاوت‌های الگوریتم‌های هش کردن اطلاعات در نگارش‌های مختلف ASP.NET Identity

اگر به سورس نگارش سوم ASP.NET Identity مراجعه کنیم، یک چنین کامنتی در ابتدای آن قابل مشاهده است:

```
/* ========================
* HASHED PASSWORD FORMATS
* ========================
*
* Version 2:
* PBKDF2 with HMAC-SHA1, 128-bit salt, 256-bit subkey, 1000 iterations.
* (See also: SDL crypto guidelines v5.1, Part III)
* Format: { 0x00, salt, subkey }
*
* Version 3:
* PBKDF2 with HMAC-SHA256, 128-bit salt, 256-bit subkey, 10000 iterations.
* Format: { 0x01, prf (UInt32), iter count (UInt32), salt length (UInt32), salt, subkey }
* (All UInt32s are stored big-endian.)
*/
```

در نگارش دوم آن از الگوریتم PBKDF2 با هزار بار تکرار و در نگارش سوم با 10 هزار بار تکرار، استفاده شده‌است. در این بین، الگوریتم پیش فرض HMAC-SHA1 نگارش‌های 2 نیز به HMAC-SHA256 در نگارش 3، تغییر کرده‌است.

در یک چنین حالتی بانک اطلاعاتی ASP.NET Identity 2.x شما با نگارش بعدی سازگار نخواهد بود و تمام کلمات عبور آن باید مجددا ریست شده و مطابق فرمت جدید هش شوند. بنابراین امکان انتخاب الگوریتم هش کردن را نیز پیش بینی کرده‌اند .

در نگارش دوم ASP.NET Identity، متد هش کردن یک کلمه‌ی عبور، چنین شکلی را دارد:

```
public static string HashPassword(string password, int numberOfRounds = 1000)
{
```

```
    if (password == null)
        throw new ArgumentNullException("password");

    byte[] saltBytes;
    byte[] hashedPasswordBytes;
    using (var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, 16, numberOfRounds))
    {
        saltBytes = rfc2898DeriveBytes.Salt;
        hashedPasswordBytes = rfc2898DeriveBytes.GetBytes(32);
    }
    var outArray = new byte[49];
    Buffer.BlockCopy(saltBytes, 0, outArray, 1, 16);
    Buffer.BlockCopy(hashedPasswordBytes, 0, outArray, 17, 32);
    return Convert.ToBase64String(outArray);
}
```

تفاوت این روش با مثال ابتدای بحث، مشخص کردن طول salt در متد Rfc2898DeriveBytes است؛ بجای محاسبه‌ی اولیه‌ی آن. در این حالت متد Rfc2898DeriveBytes مقدار salt را به صورت خودکار محاسبه می‌کند. این salt بجای ذخیره شدن در یک فیلد جداگانه، به ابتدای مقدار هش شده اضافه گردیده و به صورت یک رشته‌ی base64 ذخیره می‌شود. در نگارش سوم ، از کلاس ویژه‌ی RandomNumberGenerator برای محاسبه‌ی Salt استفاده شده‌است.

نظرات خوانندگان

نویسنده: امیر صیدی لو
تاریخ: ۱۳۹۴/۰۴/۱۵ ۶:۳۴

ممنون از مطلب خوبتون
ولی یه مشکلی که من موقع تست برخوردم این بود که زمان تبدیل آرایه تولید شده به وسیله تابع HashPassword به معادل رشته ای اون برای ذخیره در دیتابیس و بازیابی اون رشته به معادل آرایه اون برای چک کردن صحت کلمه عبور هر دو مقدار قبل از تبدیل و بعد از تبدیل با هم برابر بودن و مشکلی نداشتن ولی هنگام همین عمل تبدیل برای مقدار salt و بازیابیش از دیتا بیس مقدار قبل تبدیل و بعدش یکسان نبودن به همین خاطر مجبور شدم مقدار salt رو به صورت آرایه توی دیتابیس ذخیره کنم، خروجی حاصل از salt هم چک کردم نمی‌دونم چرا آرایه حاصل بیشتر از 32 خانه بود؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۴/۱۵ ۱۰:۴

در ASP.NET Identity جمع هش و salt با فرمت base64 در بانک اطلاعاتی به صورت رشته‌ای با طول max ذخیره می‌شوند (هر دو با هم در یک فیلد). همچنین در اینجا طول salt به صورت صریح به 16 بایت تنظیم شده‌است (متد آخر مطلب).

نویسنده: امیر صیدی لو
تاریخ: ۱۳۹۴/۰۴/۱۵ ۱۰:۳۵

تو این حالت (یکی کردن salt و hashPassword) چطوری می‌تونیم مقدار salt رو از دیتا بیس بخونیم و با کلمه عبور ورودی کاربر جمع بزنیم و با مقدار hashPassword اولیه مقایسه کنیم؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۴/۱۵ ۱۱:۹

از متدهای HashPassword و VerifyHashedPassword سورس ASP.NET Identity ایده بگیرید. مورد اول برای ذخیره سازی اطلاعات در بانک اطلاعاتی است. مورد دوم در حین لاگین، جهت تعیین اعتبار کلمه‌ی عبور کاربر استفاده می‌شود.

عنوان: **Claim Based Identity**
نویسنده: احمد نواصری
تاریخ: ۱۳۹۴/۰۴/۰۶ ۱۴:۲۵
آدرس: www.dotnettips.info
گروه‌ها: MVC, Security, ASP.NET Identity

بنده در حال توسعه‌ی یک CMS هستم و این کار را برای یادگیری MVC انجام میدم. الان هم تقریبا رسیدم به اواخر کار و انشالله اگه کار تمام شد، نرم افزار را به صورت سورس باز منتشر می‌کنم. الان رسیدم به قسمت مدیریت کاربران. همانطور که می‌دانید ASP.NET در نسخه‌های جدید خودش بر خلاف نسخه‌های قدیمی که از SQL Membership استفاده می‌کرد الان از سیستم Identity بهره می‌برد، که انشالله در نوشتارهای بعدی به موضوع Identity به تفصیل خواهیم پرداخت. در حقیقت سیستم Identity یک نوع Claim Based Identity هست. اما حالا ببینیم که این Claim چی هست؟ اما قبل از آن یک سری اصطلاحات را که به درک بهتر مفهوم کمک می‌کند، باید تعریف کنیم:

Relying Party (RP) = Application

Service Providers (SP) = Application

RP یا SP یک Application می‌باشد که از Claim استفاده می‌کند. واژه‌ی Relying Party بدین دلیل انتخاب شده که Application روی یک Issuer به منظور تامین اطلاعات در مورد یک Identity تکیه می‌کند.

Subject = User

Principal = User

واژه‌ی Subject یا Principal در حقیقت یک User می‌باشد. این واژه زمانی معنا پیدا می‌کند که شما به User به عنوان یک Subject برای کنترل دسترسی، شخصی سازی (Personalization) و ... بنگرید. در Net Framework. به جای Subject از Principal استفاده می‌شود.

Security Token Service (STC) = Issuer

از دید فنی، STC یک رابط درون یک Issuer می‌باشد که درخواست‌ها (Request) را تأیید و اقدام به ساخت Issues Security Token (توکن‌های مسائل امنیتی) می‌نماید. توکن‌های مسائل امنیتی شامل یک سری Claim می‌باشند.

Identity Provider (IDP) = Issuer

تامین کننده Identity یک Issuer یا Token Issuer می‌باشد که وظیفه‌ی تعیین اعتبار (Validation) کاربر مانند نام کاربری، رمز عبور و ... را بر عهده دارد.

Active Client = Smart or rich Client

Passive Client = Browser

یک Active Client می‌تواند از یک کتابخانه‌ی پیچیده مانند WCF به منظور پیاده سازی پروتکل‌هایی که بتوانند یک Security Token را درخواست و پاس دهند، استفاده نمایند. به منظور پشتیبانی از مرورگرهای مختلف، سناریوهای passive از پروتکل‌های ساده‌تری به منظور درخواست و پاس دادن یک Security Token که بر روی پروتکل Http تکیه دارد استفاده می‌کند (Http Post - Http Get).

و اما Claim چیست؟

در حقیقت Claim عبارتست از یک بیانیه یا شرح که یک Subject در مورد خودش یا Subject دیگری می‌سازد. این بیانیه می‌تواند در مورد یک نام، هویت، کلید، گروه، حق دسترسی و یا یک قابلیت باشد. Claimها بوسیله یک Provider صادر و سپس بسته بندی (Package) شده و بوسیله‌ی یک Issuer صادر می‌شوند که این Issuer عموما با نام Security Token Service شناخته می‌شود.

به منظور آشنایی با این مبحث میتوانید به اینجا و اینجا مراجعه نمایید.

این نوشتار مقدمه‌ای بود بر مباحث ASP.NET Identity. بنده در حال ترجمه‌ی سه فصل آخر کتاب Pro ASP.NET Mvc 5 Platform که اختصاص به مبحث Identity دارد هستم. فصل 13 تقریبا تمام شده و انشالله بزودی آن‌را منتشر میکنم.

عنوان: **Asp.Net Identity #1**
نویسنده: احمد نواصری
تاریخ: ۱۳۹۴/۰۴/۰۷ ۱۳:۴۵
آدرس: www.dotnettips.info
گروه‌ها: Security, ASP.NET Identity

Identity ، API جدید مایکروسافت جهت مدیریت کاربران در برنامه‌های ASP.NET می‌باشد. نقطه‌ی اتکای مایکروسافت در سالهای اخیر برای مدیریت کاربران سیستم ASP.NET Membership بود که از ضعف‌های طراحی رنج می‌برد. مهمترین محدودیت این سیستم این بود که داده‌های ذخیره شده توسط Schema، فقط قابلیت کار با SQL Server را دارا بود که توسعه‌ی آن بدون پیاده سازی دوباره‌ی کلاسهای تامین کننده (Provider Classes) بسیار مشکل بود. بعد از آن مایکروسافت جهت کاهش پیچیدگی سیستم ASEP.NET Membership، دو سیستم دیگر به نامهای Simple Membership و ASP.NET Universal Providers را عرضه نمود. این سیستم‌ها تا حدودی پیچیدگی را کم کردند، اما همچنان برمبنای SQL Server بودند. تا اینکه مایکروسافت سیستم ASP.NET Membership را با سیستم Identity جهت حل دو مشکل مطرح شده تعویض نمود. سیستم Identity هم پایدار است و هم قابل توسعه. به علاوه با استفاده از قابلیت Open Source بودن، قابل وفق دادن می‌باشد (یعنی برحسب نیاز پروژه می‌توان کل یا قسمتهایی را پیاده سازی نمود). برای بسیاری از توسعه دهندگان برنامه‌های ASP.NET، سیستم Identity اولین پرده برداری از OWIN خواهد بود. در حقیقت OWIN یک لایه‌ی Abstract می‌باشد که برنامه‌ی وب را از محیطی که آن‌را میزبانی کرده، ایزوله می‌کند. هدف از ایزوله کردن، ایجاد انعطاف پذیری بیشتر در محیط هایی که برنامه‌های ASP.Net را میزبانی می‌کنند و همچنین ایجاد یک زیرساخت سبک وزن جهت سرویس دهی می‌باشد. همان طور که عنوان شد OWIN یک استاندارد باز است و مایکروسافت پروژه KATANA را به منظور پیاده سازی استانداردی از OWIN طراحی کرد. این طراحی همراه با یک سری کامپوننت به منظور تامین ویژگی‌هایی (Functionalities) که یک برنامه وب به آنها نیاز دارد همراه بود. زیبایی کار مایکروسافت در این بود که OWIN/KATANA پشته‌ی تکنولوژی ASP.NET را از بقیه‌ی قسمتهای .NET Framework ایزوله کرده که اجازه نرخ بالاتری از تغییرات را می‌دهد. توسعه دهندگان OWIN به جای استفاده از کل پلتفرم (بر عکس چیزی که در حال حاضر در سرویسهای ASP.NET رخ می‌دهد) فقط سرویس‌هایی را که برنامه آنها نیاز دارد، انتخاب می‌کنند. این سرویسهای منفرد که در واژه نامه‌ی OWIN با نام Middleware شناخته می‌شوند، می‌توانند در نرخ‌های مختلفی توسعه داده شوند و توسعه دهندگان را قادر می‌سازند که بین تأمین کنندگان سرویسهای مختلف یکی را انتخاب کنند و فقط وابسته به سرویس‌های پیاده سازی شده توسط مایکروسافت نباشند. و در پایان باید گفت که پلتفرم ASP.NET و IIS کنار گذاشته نخواهند شد. مایکروسافت شفاف سازی کرده‌است که یکی از ابعاد زیبای OWIN این است که به توسعه دهندگان اجازه انعطاف پذیری بیشتر را می‌دهد که در آن کامپوننت‌های Middleware بوسیله IIS میزبانی می‌شوند و هم اکنون پروژه‌ی KATANA توسط فضاهای نام System.Web پشتیبانی می‌شود. خوب دوستان این مقدمه ای بود بر سیستم Identity . انشالله در نوشته‌های بعدی بیشتر سیستم Identity را تحلیل و بررسی خواهیم کرد.

نظرات خوانندگان

نویسنده: افشار محبی
تاریخ: ۱۳۹۴/۰۴/۱۰ ۱۱:۴۴

مایکروسافت با سرعت زیادی در حال ایجاد تغییرات در سکوی توسعه وب خود است تا بتواند فاصله‌اش را از رقبای دیگر کم کند. توسعه دهنده‌ها هم باید با همین سرعت تغییرات اخیر مایکروسافت را دنبال کنند و الا بد جور از قافله عقب خواهند افتاد.

نویسنده: محمد رعیت پیشه
تاریخ: ۱۳۹۴/۰۴/۱۳ ۲۰:۱۱

همچنین به نظرم فرصت بسیار خوبی است برای کسانی که فکر می‌کنند عقب افتاده اند تا خود را با قافله جدید همراه کنند.

عنوان: Asp.Net Identity #2
نویسنده: احمد نواصری
تاریخ: ۱۳۹۴/۰۴/۱۰ ۲۰:۱۵
آدرس: www.dotnettips.info
گروه‌ها: Security, ASP.NET Identity

پیشتر در اینجا در مورد تاریخچه‌ی سیستم Identity مطالبی را عنوان کردیم. در این مقاله می‌خواهیم نحوه‌ی برپایی سیستم Identity را بحث کنیم.

ASP.NET Identity مانند ASP.NET Membership به اسکیمای SQL Server وابسته نیست؛ اما Relational Storage همچنان واحد ذخیره سازی پیش فرض و آسانی می‌باشد. بدین جهت که تقریبا بین همه‌ی توسعه دهندگان جا افتاده است. ما در این نوشتار از LocalDB جهت ذخیره سازی جداول استفاده می‌کنیم. ذکر این نکته ضروری است که سیستم Identity از Entity Framework Code First جهت ساخت اسکیما استفاده می‌کند.

پیش از هر چیز، ابتدا یک پروژه‌ی وب را ایجاد کنید؛ مانند شکل زیر:

در مرحله‌ی بعد سه پکیج زیر را باید نصب کنیم :

- Microsoft.AspNet.Identity.EntityFramework

- Microsoft.AspNet.Identity.OWIN

- Microsoft.Owin.Host.SystemWeb

بعد از اینکه پکیج‌های بالا را نصب کردیم، باید فایل Web.config را بروز کنیم. اولین مورد، تعریف یک Connection String می‌باشد:

```
<connectionStrings>
 <add name="IdentityDb"
      providerName="System.Data.SqlClient"
      connectionString="Data Source=(localdb)\v11.0;Initial Catalog=IdentityDb;Integrated
Security=True;Connect
Timeout=15;Encrypt=False;TrustServerCertificate=False;MultipleActiveResultSets=True"/>
 </connectionStrings>
```

بعد از آن، تعریف یک کلید در قسمت AppSettings تحت عنوان Owin:AppStartup است:

```
<appSettings>
 <add key="webpages:Version" value="3.0.0.0" />
 <add key="webpages:Enabled" value="false" />
 <add key="ClientValidationEnabled" value="true" />
 <add key="UnobtrusiveJavaScriptEnabled" value="true" />
```

</ "add key="owin:AppStartup" value="Users.IdentityConfig>

```
</appSettings>
```

Owin مدل شروع برنامه (Application Startup Model) خودش را تعریف می‌کند که از کلاس کلی برنامه (منظور Global.asax) جداست. AppSetting تعریف شده با نام owin:Startup شناخته می‌شود و مشخص کننده کلاسی است که Owin وهله سازی خواهد کرد. وقتی برنامه شروع به کار می‌کند، تنظیمات خودش را از این کلاس دریافت خواهد کرد (در این نوشتار کلاس IdentityConfig می‌باشد).

ساخت کلاس User

مرحله‌ی بعد ساخت کلاس User می‌باشد. این کلاس بیانگر یک کاربر می‌باشد. کلاس User باید از کلاس IdentityUser ارث بری کند که این کلاس در فضای نام Microsoft.AspNet.Identity.EntityFramework قرار دارد. کلاس IdentityUser فراهم کننده‌ی یک کاربر عمومی و ابتدایی است. اگر بخواهیم اطلاعات اضافی مربوط به کاربر را ذخیره کنیم باید آنها در کلاسی که از کلاس IdentityUser ارث بری می‌کند قرار دهیم. کلاس ما در اینجا AppUser نام دارد.

```
using System;
using Microsoft.AspNet.Identity.EntityFramework;
namespace Users.Models
{
   public class AppUser : IdentityUser
  {
      // پروپرتی‌های اضافی در اینجا
  }
}
```

ساخت کلاس Database Context برنامه

مرحله‌ی بعد ساخت کلاس DbContext برنامه می‌باشد که بر روی کلاس AppUser عمل میکند. کلاس Context برنامه که ما در اینجا آن را AppIdentityDbContext تعریف کرده‌ایم، از کلاس IdentityDbContext<T> ارث بری می‌کند که T همان کلاس User می‌باشد (در مثال ما AppUser) .

```
using System.Data.Entity;
using Microsoft.AspNet.Identity.EntityFramework;
using Users.Models;
```

```
namespace Users.Infrastructure {
 public class AppIdentityDbContext : IdentityDbContext<AppUser>
{
   public AppIdentityDbContext()
              : base("IdentityDb") { }

  static AppIdentityDbContext()
 {
    Database.SetInitializer<AppIdentityDbContext>(new IdentityDbInit());
  }
 public static AppIdentityDbContext Create() {
 return new AppIdentityDbContext();
 }
 }
public class IdentityDbInit
 : DropCreateDatabaseIfModelChanges<AppIdentityDbContext> {
 protected override void Seed(AppIdentityDbContext context) {
 PerformInitialSetup(context);
 base.Seed(context);
 }
 public void PerformInitialSetup(AppIdentityDbContext context) {
 // initial configuration will go here
 }
 }
}
```

ساخت کلاس User Manager

یکی از مهمترین کلاسهای Identity کلاس User Manager (مدیر کاربر) می‌باشد که نمونه‌هایی از کلاس User را مدیریت می‌کند. کلاسی را که تعریف می‌کنیم، باید از کلاس UserManager<T> ارث بری کند که T همان کلاس User می‌باشد. کلاس UserManager خاص EF نمی‌باشد و ویژگی‌های عمومی بیشتری برای ساخت و انجام عملیات بر روی داده‌های کاربر را فراهم می‌نماید.

Name	Description
ChangePasswordAsync(id, old, new)	Changes the password for the specified user.
CreateAsync(user)	Creates a new user without a password. See Chapter 15 for an example.
CreateAsync(user, pass)	Creates a new user with the specified password. See the "Creating Users" section.
DeleteAsync(user)	Deletes the specified user. See the "Implementing the Delete Feature" section.
FindAsync(user, pass)	Finds the object that represents the user and authenticates their password. See Chapter 14 for details of authentication.
FindByIdAsync(id)	Finds the user object associated with the specified ID. See the "Implementing the Delete Feature" section.
FindByNameAsync(name)	Finds the user object associated with the specified name. I use this method in the "Seeding the Database" section of Chapter 14.
UpdateAsync(user)	Pushes changes to a user object back into the database. See the "Implementing the Edit Feature" section.
Users	Returns an enumeration of the users. See the "Enumerating User Accounts" section.

کلاس UserManager حاوی متدهای بالا است. اگر دقت کنید، می‌بینید که تمامی متدهای بالا به کلمه‌ی Async ختم می‌شوند. زیرا Asp.Net Identity تقریبا کل ویژگیهای برنامه نویسی Async را پیاده سازی کرده است و این بدین معنی است که عملیات میتوانند به صورت غیر همزمان اجرا شده و دیگر فعالیت‌ها را بلوکه نکنند.

```
using Microsoft.AspNet.Identity;
```

```
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Users.Models;

namespace Users.Infrastructure
{
 public class AppUserManager : UserManager<AppUser>
{

 public AppUserManager(IUserStore<AppUser> store)
 : base(store) {
 }
 public static AppUserManager Create( IdentityFactoryOptions<AppUserManager> options, IOwinContext
context)
{
 AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
 AppUserManager manager = new AppUserManager(new UserStore<AppUser>(db));
 return manager;
 }
 }
}
```

زمانی که Identity نیاز به وهله‌ای از کلاس AppUserManager داشته باشد، متد استاتیک Create را صدا خواهد زد. این عمل زمانی اتفاق می‌افتد که عملیاتی بر روی داده‌های کاربر انجام گیرد. برای ساخت وهله‌ای از کلاس AppUserManager نیاز به کلاس UserStor<AppUser> دارد. کلاس UserStore<T> یک پیاده سازی از رابط IUserStore<T> توسط EF میباشد که وظیفه‌ی آن فراهم کننده‌ی پیاده سازی Storage-Specific متدهای تعریف شده در کلاس User Manager (که در اینجا AppUserManager) می‌باشد. برای ساخت UserStore<T> نیاز به وهله‌ای از کلاس AppIdentityDbContext می‌باشد که از طریق Owin به صورت زیر قابل دریافت است:

```
AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
```

یک پیاده سازی از رابط IOwinContext، به عنوان پارامتر به متد Create پاس داده می‌شود. در این پیاده سازی، یک تابع جنریک به نام Get تعریف شده که اقدام به برگشت وهله ای از اشیای ثبت شده‌ی در کلاس شروع Owin می‌نماید.

ساخت کلاس شروع Owin

اگر خاطرتان باشد یک کلید در قسمت AppSettings فایل Web.config به صورت زیر تعریف کردیم:

```
<add key="owin:AppStartup" value="Users.IdentityConfig" />
```

قسمت Value کلید بالا از دو قسمت تشکیل شده است: Users بیانگر فضای نام برنامه‌ی شماست و IdentityConfig بیانگر کلاس شروع می‌باشد. (البته شما می‌توانید هر نام دلخواهی را برای کلاس شروع بگذارید. فقط دقت داشته باشید که نام کلاس شروع و مقدار، با کلیدی که تعریف می‌کنید یکی باشد)

Owin مستقل از ASP.NET اعلام شده است و قراردادهای خاص خودش را دارد. یکی از این قراردادها تعریف یک کلاس و وهله سازی آن به منظور بارگذاری و پیکربندی میان افزار و انجام دیگر کارهای پیکربندی که نیاز است، می‌باشد. به طور پیش فرض این کلاس Start نام دارد و در پوشه‌ی App_Start تعریف می‌شود. این کلاس حاوی یک متد به نام Configuration می‌باشد که بوسیله زیرساخت Owin فراخوانی می‌شود و یک پیاده سازی از رابط Owin.IAppBuilder به عنوان آرگومان به آن پاس داده می‌شود که کار پشتیبانی از Setup میان افزار مورد نیاز برنامه را برعهده دارد.

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;
using Users.Infrastructure;
namespace Users
{
 public class IdentityConfig
```

```
{
 public void Configuration(IAppBuilder app)
{
 app.CreatePerOwinContext<AppIdentityDbContext>(AppIdentityDbContext.Create);
 app.CreatePerOwinContext<AppUserManager>(AppUserManager.Create);
 app.UseCookieAuthentication(new CookieAuthenticationOptions {
 AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
 LoginPath = new PathString("/Account/Login"),
 });
 }
 }
}
```

رابط IAppBuilder بوسیله تعدادی متد الحاقی که در کلاسهایی که در فضای نام Owin تعریف شده‌اند، تکمیل شده است. متد CreatePerOwinContext کار ساخت وهله‌ای از کلاس AppUserManager و کلاس AppIdentityDbContext را برای هر درخواست بر عهده دارد. این مورد تضمین می‌کند که هر درخواست، به یک داده‌ی تمیز از Asp.Net Identity دسترسی خواهد داشت و نگران اعمال همزمانی و یا کش ضعیف داده نخواهد بود. متد UseCookieAuthentication به Asp.Net Identity می‌گوید که چگونه از کوکی‌ها برای تصدیق هویت کاربران استفاده کند که Optionهای آن از طریق کلاس CookieAuthenticationOptions مشخص می‌شود. مهمترین قسمت در اینجا پروپرتی LoginPath می‌باشد و مشخص می‌کند که کلاینت‌های تصدیق هویت نشده، هنگام دسترسی به یک منبع محافظت شده، به کدام URL هدایت شوند که توسط یک رشته به متد PathString پاس داده می‌شود.

خوب دوستان برپایی سیستم Identity به پایان رسید. انشالله در قسمت بعدی به چگونگی استفاده‌ی از این سیستم خواهیم پرداخت.

عنوان: **اهراز هویت با حساب‌های اجتماعی Google, Facebook و ...**
نویسنده: امیدنصری
تاریخ: ۱۳۹۴/۰۴/۱۶ ۱:۵۰
آدرس: www.dotnettips.info
گروه‌ها: MVC, Security, ASP.NET Identity, Google

در این مقاله نحوه‌ی ورود به یک سایت ASP.NET MVC را با حساب‌های کاربری سایت‌های اجتماعی، بررسی خواهیم کرد. در اینجا با ورود به سایت در وب فرم‌ها آشنا شدید. توضیحات مربوطه به OpenID هم در اینجا قرار دارد.

مقدمه: شروع را با نصب ویژوال استودیوی نسخه رایگان 2013 برای وب و یا نسخه‌ی 2013 آغاز می‌کنیم. برای راهنمایی استفاده از Yahoo, Twitter, twitch, Tripit, Stack Exchange, STEAM salesforce buffer, Instagram, Linkedin, GitHub, Dropbox و بیشتر اینجا کلیک کنید.

توجه: برای استفاده از Google OAuth 2 و دیباگ به صورت لوکال بدون اخطار SSL، شما می‌بایستی نسخه‌ی ویژوال استودیو 2013 آپدیت 3 و یا بالاتر را نصب کرده باشید.

ساخت اولین پروژه: ویژوال استودیو را اجرا نماید. در سمت چپ بر روی آیکن Web کلیک کنید تا آیتم ASP.NET Web Application در دات نت 4.5.1 نمایش داده شود. یک نام را برای پروژه انتخاب نموده و OK را انتخاب نماید.
در دیالوگ بعدی آیتم MVC را انتخاب و اطمینان داشته باشید **Individual User Accounts** که با انتخاب **Change Authentication** به صورت دیالوگ برای شما نمایش داده می‌شود، انتخاب گردیده و در نهایت بر روی OK کلیک کنید.

فعال نمودن حساب کاربری گوگل و اعمال تنظیمات اولیه: در این بخش در صورتیکه حساب کاربری گوگل ندارید، وارد سایت گوگل شده و یک حساب کاربری را ایجاد نماید. در غیر اینصورت اینجا کلیک کنید تا وارد بخش Google Developers Console شوید.

در بخش منو بر روی ایجاد پروژه کلیک کنید تا پروژه‌ای جدید ایجاد گردد.

در دیالوگ باز شده نام پروژه خودتان را وارد کنید و دکمه‌ی Create را زده تا عملیات ایجاد پروژه انجام شود. در صورتیکه با موفقیت پیش رفته باشید، این صفحه برای شما بارگزاری میگردد.

فعال سازی Google+API

در سمت چپ تصویر **بالا** آیتمی با نام APIs & auth خواهید دید که بعد از کلیک بر روی آن، زیر مجموعه‌ای برای این آیتم فعال میگردد که می‌بایستی بر روی APIs کلیک و در این قسمت به جستجوی آیتمی با نام Google+ API پرداخته و در نهایت این آیتم را برای پروژه فعال سازید.

ایجاد یک Client ID : در بخش Credentials بر روی دکمه‌ی Create new Client ID کلیک نماید.

در دیالوگ باز شده از شما درخواست می‌شود تا نوع اپلیکشن را انتخاب کنید که در اینجا می‌بایستی آیتم اول (Web application) را برای گام بعدی انتخاب کنید و با کلیک بر روی Configure consent screen به صفحه‌ی Consent screen هدایت خواهید شد. فیلدهای مربوطه را به درستی پر کنید (این بخش به عنوان توضیحات مجوز ورود بین سایت شما و گوگل است).

در نهایت بعد از کلیک بر روی Save به صفحه‌ی Client ID بازگشت داده خواهید شد که در این صفحه با این دیالوگ برخورد خواهید کرد.

پروژه‌ی MVC خودتان را اجرا و لینک و پورت مربوطه را کپی کنید (http://localhost:5063).

در Authorized JavaScript Origins لینک را کپی نماید و در بخش Authorized redirect URls لینک را مجدد کپی نماید. با این تفاوت که بعد از پورت signin-google را هم قرار دهید. (http://localhost:5063/signin-google)

حال بر روی دکمه‌ی Create Client ID کلیک کنید.

پیکربندی فایل Startup.Auth :

فایل web.config را که در ریشه‌ی پروژه قرار دارد باز کنید. در داخل تگ appSettings کد زیر را کپی کنید. توجه شود بجای دو مقدار value، مقداری را که گوگل برای شما ثبت کرده است، وارد کنید.

```
<appSettings>
    <!--Google-->
    <add key="GoogleClientId" value="555533955993-
fgk9d4a9999ehvfpqrukjl7r0a4r5tus.apps.googleusercontent.com" />
    <add key="GoogleClientSecretId" value="QGEF4zY4GEwQNXe8ETwnVHfz" />
  </appSettings>
```

فایل Startup.Auth را باز کنید و دو پراپرتی و یک سازنده‌ی بدون ورودی را تعریف نماید. توضیحات بیشتر به صورت کامنت در کد زیر قرار گرفته است.

```
//فضا نام‌های استفاده شده در این کلاس
using System;
using System.Configuration;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.Google;
using Owin;
using Login.Models;
//فضا نام جاری پروژه
namespace Login
{
    /// <summary>
    /// در ریشه سایت فایلی با نام استارت آپ که کلاسی هم نام این کلاس با یک تابع و یک ورودی از نوع
اینترفیس  تعریف شده است
    ///که این دو کلاس به صورت پارشال مهروموم شده اند
    /// </summary>
    public partial class Startup
    {
        /// <summary>
        /// این پراپرتی مقدار کلایت ای دی رو از وب دات کانفیگ در سازنده بدون ورودی در خودش ذخیره میکند
        /// </summary>
        public string GoogleClientId { get; set; }
        /// <summary>
        /// این پراپرتی مقدار کلایت  سیکرت ای دی رو از وب دات کانفیگ در سازنده بدون ورودی در خودش
```

```
ذخیره میکند
        /// </summary>
        public string GoogleClientSecretId { get; set; }

        /// <summary>
        سازنده بدون ورودی ///
        /// به ازای هر بار نمونه سازی از کلاس، سازنده‌های بدون ورودی کلاس هر بار اجرا خواهند شد، توجه
شود که می‌توان از
        /// سازنده‌های استاتیک هم استفاده کرد، این سازنده فقط یک بار، در صورتی که از کلاس نمونه سازی شود
ایجاد میگردد
        /// </summary>
        public Startup()
        {
            //Get Client ID from Web.Config
            GoogleClientId = ConfigurationManager.AppSettings["GoogleClientId"];
            //Get Client Secret ID from Web.Config
            GoogleClientSecretId = ConfigurationManager.AppSettings["GoogleClientSecretId"];
        }

        ///// <summary>
        ///// سازنده استاتیک کلاس
        ///// </summary>
        //static Startup()
        //{
              //در صورتی که از این سازنده استفاده شود می‌بایست پراپرتی‌های تعریف شده در سطح کلاس به
صورت استاتیک تعریف گردد تا
              //بتوان در این سازنده سطح دسترسی گرفت
        //    GoogleClientId = ConfigurationManager.AppSettings["GoogleClientId"];
        //    GoogleClientSecretId = ConfigurationManager.AppSettings["GoogleClientSecretId"];
        //}
        // For more information on configuring authentication, please visit
http://go.microsoft.com/fwlink/?LinkId=301864
        public void ConfigureAuth(IAppBuilder app)
        {
            // Configure the db context, user manager and signin manager to use a single instance per
request
            app.CreatePerOwinContext(ApplicationDbContext.Create);
            app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);
            app.CreatePerOwinContext<ApplicationSignInManager>(ApplicationSignInManager.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);

            //Initialize UseGoogleAuthentication
            app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
            {
                ClientId = GoogleClientId,
                ClientSecret = GoogleClientSecretId
            });
        }
```

```
    }
}
```

حال پروژه را اجرا کرده و به صفحه‌ی ورود کاربر رجوع نمائید. همانگونه که در تصویر زیر مشاهده می‌کنید، دکمه‌ای با مقدار نمایشی گوگل در سمت راست، در بخش Use another service to log in اضافه شده است که بعد از کلیک بر روی آن، به صفحه‌ی هویت سنجی گوگل ریداریکت می‌شوید.

در اینجا از کاربر سوال پرسیده میشود که آیا به سایت پذیرنده اجازه داده شود که اطلاعات و ایمیل شما ارسال گردند که بعد از انتخاب دکمه‌ی Accept، لاگین انجام گرفته و اطلاعات ارسال می‌گردد.

توجه: رمز عبور شما به هیچ عنوان برای سایت پذیرنده ارسال نمی‌گردد.

لاگین با موفقیت انجام شد.

در مطلب بعدی سایر سایت‌های اجتماعی قرار خواهند گرفت.

پروژه‌ی مطلب جاری را میتوانید از اینجا دانلود کنید.

عنوان: **Asp.Net Identity #3**
نویسنده: احمد نواصری
تاریخ: ۱۳۹۴/۰۴/۱۷ ۱۹:۵۰
آدرس: www.dotnettips.info
گروه‌ها: MVC, Security, ASP.NET Identity

در مقاله‌ی پیشین نگاهی داشتیم به نحوه‌ی برپایی سیستم Identity. در این مقاله به نحوه‌ی استفاده از این سیستم به منظور طراحی یک سیستم مدیریت کاربران خواهیم پرداخت و انشالله در مقاله‌های بعدی این سیستم را تکمیل خواهیم نمود. کار را با اضافه کردن یک کنترلر جدید به پروژه آغاز می‌کنیم.

```
using System.Web;
using System.Web.Mvc;
using Microsoft.AspNet.Identity.Owin;
using Users.Infrastructure;

namespace Users.Controllers
{
    public class HomeController : Controller
    {
        private AppUserManager UserManager
        {
            get { return HttpContext.GetOwinContext().GetUserManager<AppUserManager>(); }
        }
        // GET: Home
        public ActionResult Index()
        {
            return View(UserManager.Users);

        }

}
```

در خط 10 یک پروپرتی از نوع AppUserManager (کلاسی که مدیریت کاربران را برعهده دارد) ایجاد می‌کنیم. اسمبلی Microsoft.Owin.Host.SystemWeb یک سری متدهای الحاقی را به کلاس HttpContext اضافه می‌کند که یکی از آنها متد GetOwinContext می‌باشد. این متد یک شیء Per-Request Context را از طریق رابط IOwinContext به OwinApi ارسال می‌کند؛ با استفاده از متد الحاقی GetUserManager<T> که T همان کلاس AppUserManager می‌باشد. حال که نمونه‌ای از کلاس AppUserManager را بدست آوردیم، می‌توانیم درخواستهایی را به جداول کاربران بدهیم. مثلا در خط 17 با استفاده از پروپرتی Users میتوانیم لیست کاربران موجود را بدست آورده و آن را به ویو پاس دهیم.

```
@using Users.Models
@model IEnumerable<AppUser>
@{
    ViewBag.Title = "Index";
}
<div class="panel panel-primary">
    <div class="panel-heading">
        User Accounts
    </div>
    <table class="table table-striped">
        <tr><th>ID</th><th>Name</th><th>Email</th></tr>
        @if (!Model.Any())
        {
            <tr><td colspan="3" class="text-center">No User Accounts</td></tr>
        }
        else
        {
            foreach (AppUser user in Model)
            {
                <tr>
                    <td>@user.Id</td>
                    <td>@user.UserName</td>
                    <td>@user.Email</td>
                </tr>
            }
        }
    </table>
</div>
@Html.ActionLink("Create", "CreateUser", null, new { @class = "btn btn-primary" })
```

نحوه‌ی ساخت یک کاربر جدید

ابتدا در پوشه Models یک کلاس ایجاد کنید :

```
namespace Users.Models
    {
        public class CreateModel
        {
            [Required]
            public string Name { get; set; }
            [Required]
            public string Email { get; set; }
            [Required]
            public string Password { get; set; }
        }
    }
```

فقط دوستان توجه داشته باشید که در پروژه‌های حرفه‌ای و تجاری هرگز اطلاعات مهم مربوط به مدل‌ها را در پوشه‌ی Models قرار ندهید. ما در اینجا صرف آموزش و برای جلوگیری از پیچیدگی مثال این کار را انجام میدهیم. برای اطلاعات بیشتر به این مقاله مراجعه کنید.

حال در کنترلر برنامه کدهای زیر را اضافه می‌کنیم:

```
public ActionResult CreateUser()
        {
            return View();
        }

        [HttpPost]
        public async Task<ActionResult> CreateUser(CreateModel model)
        {
            if (!ModelState.IsValid)
                return View(model);

            var user = new AppUser { UserName = model.Name, Email = model.Email };
            var result = await UserManager.CreateAsync(user, model.Password);

            if (result.Succeeded)
            {
                return RedirectToAction("Index");
            }

            foreach (var error in result.Errors)
            {
                ModelState.AddModelError("", error);
            }
            return View(model);
        }
```

در اکشن CreateUser ابتدا یک شیء از کلاس AppUser ساخته و پروپرتی‌های مدل را به پروپرتی‌های کلاس AppUser انتساب می‌دهیم. در مرحله‌ی بعد یک شیء از کلاس IdentityResult به نام result ایجاد کرده و نتیجه‌ی متد CreateAsync را درون آن قرار می‌دهیم. متد CreateAsync از طریق پروپرتی از نوع AppUserManager قابل دسترسی است و دو پارامتر را دریافت می‌کند. پارامتر اول یک شیء از کلاس AppUser و پارامتر دوم یک رشته‌ی حاوی Password می‌باشد و خروجی متد یک شیء از کلاس IdentityResult است. در مرحله‌ی بعد چک می‌کنیم اگر Result، مقدار Succeeded را داشته باشد (یعنی نتیجه موفقیت آمیز بود) آن‌وقت ... در غیر اینصورت خطاهای موجود را به ModelState اضافه نموده و به View می‌فرستیم.

```
@model Users.ViewModels.CreateModel

@Html.ValidationSummary(false)

@using (Html.BeginForm())
{
    <div class="form-group">
        <label>Name</label>
        @Html.TextBoxFor(x => x.UserName, new { @class = "form-control" })
    </div>
    <div class="form-group">
        <label>Email</label>
        @Html.TextBoxFor(x => x.Email, new { @class = "form-control" })
    </div>
```

```
    <div class="form-group">
        <label>Password</label>
        @Html.PasswordFor(x => x.Password, new { @class = "form-control" })
    </div>
    <button type="submit" class="btn btn-primary">Create</button>
    @Html.ActionLink("Cancel", "Index", null, new { @class = "btn btn-default" })
}
```

اعتبار سنجی رمز

عمومی‌ترین و مهمترین نیازمندی برای هر برنامه‌ای، اجرای سیاست رمزگذاری می‌باشد؛ یعنی ایجاد یک سری محدودیتها برای ایجاد رمز است. مثلا رمز نمی‌تواند از 6 کاراکتر کمتر باشد و یا باید حاوی حروف بزرگ و کوچک باشد و ... . برای اجرای سیاست‌های رمزگذاری از کلاس PasswordValidator استفاده میشود. کلاس PasswordValidator برای اجرای سیاستهای رمزگذاری از پروپرتی‌های زیر استفاده می‌کند.

Name	Description
RequiredLength	Specifies the minimum length of a valid passwords.
RequireNonLetterOrDigit	When set to true, valid passwords must contain a character that is neither a letter nor a digit.
RequireDigit	When set to true, valid passwords must contain a digit.
RequireLowercase	When set to true, valid passwords must contain a lowercase character.
RequireUppercase	When set to true, valid passwords must contain a uppercase character.

```
var manager = new AppUserManager(new UserStore<AppUser>(db))
            {
                PasswordValidator = new PasswordValidator
                {
                    RequiredLength = 6,
                    RequireNonLetterOrDigit = false,
                    RequireDigit = false,
                    RequireLowercase = true,
                    RequireUppercase = true
                }
            };
```

فقط دوستان توجه داشته باشید که کد بالا را در متد Create از کلاس AppUserManager استفاده کنید.

اعتبار سنجی نام کاربری

برای اعبارسنجی نام کاربری از کلاس UserValidator به صورت زیر استفاده می‌کنیم:

```
manager.UserValidator = new UserValidator<AppUser>(manager)
            {
                AllowOnlyAlphanumericUserNames = true,
                RequireUniqueEmail = true
            };
```

کد بالا را نیز در متد Create از کلاس AppUserManager قرار می‌دهیم.

نظرات خوانندگان

نویسنده: محمد بیات
تاریخ: ۱۳۹۴/۰۴/۲۶ ۱۴:۵

با سلام و خسته نباشید، تشکر بابت مطالب خوبتون یک سوال برای من پیش اومده و اینکه اگر بخواهیم کلاسهای دیگه ای داشته باشیم مثلا یک کلاس برای جدول کالاها که اصلا ربطی به جدول Users ما نداشته باشه چطور باید به دیتابیس کانتکس وصلش کنیم یا به عبارتی چطور می‌تونیم توی Identity بهش بفهمونیم که همچین جدولی رو هم برای ما بسازه! و بتونیم از طریق صدا زدن Databasecontext اطلاعاتش رو بخونیم....
من تمامی مطالب شما رو انجام دادم تا الان درست کار کرده و میتونم لاگین و ریجستر رو برای کاربرهام داشته باشم ولی نمیدونم چطور باید با جداول دیگه ارتباط برقرار کنم و بسازمشون.
ممنون میشم توضیح بدید.
موفق و پیروز باشید.

نویسنده: محسن خان
تاریخ: ۱۳۹۴/۰۴/۲۶ ۱۴:۳۱

در قسمت دوم Asp.Net Identity #2 این کلاس جدید رو باید به AppIdentityDbContext اضافه کنید (مثل روش معمول EF Code first توسط DbSetها). بعدش هم باید با مباحث migrations آشنایی داشته باشید.

نویسنده: احمد نواصری
تاریخ: ۱۳۹۴/۰۴/۳۱ ۰:۲۸

سلام. کلاس AppIdentityDbContext تعریف شده در Asp.net Identity #2 هیچ تفاوتی با dbContext تعریف شده توسط EF ندارد. شما میتونید DbSetهای خودتون رو توی کلاس AppIdentityDbContext تعریف کنید.

```
ublic class AppIdentityDbContext : IdentityDbContext<AppUser>
{
   public AppIdentityDbContext()
              : base("IdentityDb") { }

    // DbSet Definition
   public DbSet<Product> Products { get; set; }

    static AppIdentityDbContext()
    {
        Database.SetInitializer<AppIdentityDbContext>(new IdentityDbInit());
    }
    public static AppIdentityDbContext Create()
    {
        return new AppIdentityDbContext();
    }
 }
```

البته همونطور که دوستمون محسن خان فرمودن بایستی با مباحث Ef Code First آشنایی داشته باشید که در اینجا یک دوره کامل تدارک دیده شده.

نویسنده: روناک تابعی
تاریخ: ۱۳۹۴/۰۵/۰۹ ۱۵:۳۵

با سلام و تشکر. آیا سیستم Identity هیچ تنظیماتی در web.config نداره؟
نیازی به تنظیمات قسمت authentication نیست دیگه؟

نویسنده: روناک تابعی

تاریخ: ۱۳۹۴/۰۵/۰۹ ۱۵:۳۹

آیا استفاده از یک DbContext برای Identity و دیگر مدلهای برنامه خوبه یا اینکه برای دیگر مدلهای برنامه از DbContext جدا استفاده کنیم؟

نویسنده: محسن خان
تاریخ: ۱۳۹۴/۰۵/۰۹ ۱۵:۵۹

می‌تونید استفاده کنید. فقط باید به نکات استفاده از چندین Context در EF 6 Code first دقت داشته باشید.

نویسنده: احمد نواصری
تاریخ: ۱۳۹۴/۰۵/۱۰ ۲۰:۴۱

سلام. مراجعه کنید به Asp.Net Identity #2 . فقط کافیه که یک Connection String تعریف کنید واسه ارتباط به پایگاه داده و یک کلید که معرف کلاس شروع Owin هست . نیاز به تنظیمات اضافه‌تری نداره.

عنوان: **سفارشی‌سازی PasswordValidator در ASP.NET Identity**
نویسنده: علی اکبر کورش فر
تاریخ: ۱۳۹۴/۰۴/۲۲ ۱۵:۰
آدرس: www.dotnettips.info
گروه‌ها: MVC, Security, ASP.NET Identity

همانطور که می‌دانید Identity، فریمورک نسبتا جدیدی هست که مایکروسافت برای مدیریت کاربران و احراز هویت آن‌ها معرفی کرده و پیشرفت چشمگیری داشته است. در قسمت IdentityConfig (قسمتی که برای کانفیگ‌کردن Identity استفاده می‌شود) بخشی قابل تنظیم برای کانفیگ‌کردن سیاست‌های تعیین پسورد وجود دارد. به‌طور مثال : تعیین حداقل تعداد حروف برای کلمه‌ی عبور، ضرورت کوچک و بزرگ بودن حروف، الزام وجود کاراکتر ویژه.

این نیاز وجود دارد که PasswordValidator موجود در Identity را برای پروژه‌های مختلف سفارشی‌سازی کرد و این امکان فراهم شود که بتوان سیاست‌های کاری شرکت و پروژه را در قالب PasswordValidator اعمال کرد. به عنوان مثال بررسی کنیم اگر پسورد کاربر عدد 12345 وارد شده است، خطا صادر کنیم و اجازه انتساب آن را برای کاربر ندهیم یا منطق‌های دیگری که نیاز داریم. پس در اینجا به وجود یک CustomPasswordValidator نیاز هست است.

```
public class CustomPasswordValidator : PasswordValidator
{
    public override async Task<IdentityResult> ValidateAsync(string pass)
    {
            IdentityResult result = await base.ValidateAsync(pass);
            if (pass.Contains("12345"))
           {
            var errors = result.Errors.ToList();
            errors.Add("Passwords cannot contain numeric sequences");
            result = new IdentityResult(errors);
           }
           return result;
    }
}
```

در قطعه کد بالا یک کلاس ایجاد شد با نام CustomPasswordValidator و از PasswordValidator موجود در فضای نام Microsoft.AspNet.Identity ارث‌بری شد تا ویژگی‌های اصلی این کلاس را به‌صورت ذاتی داشته باشیم و بتوانیم سفارشی‌سازی‌های خودمان را بر روی آن اعمال می‌کنیم. متد ValidateAsync موجود override شده و بر روی ورودی آن شرط‌های پروژه و سیاست‌ها بررسی شده‌اند و درصورت تخلف از قوانین خطا صادر شد.

نقطه‌ی پایان کار اینجاست که در داخل کلاس کانفیگ موجود برای Identity که درون فایل web.config مشخص شده است (در این مثال کلاس IdentityConfig) برای قسمت PasswordValidator ، حالا باید از کلاس CustomPasswordValidator، یک شیء جدید ساخته شود:

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

عنوان: **ذخیره تنظیمات متغیر مربوط به یک وب اپلیکیشن ASP.NET MVC با استفاده از EF**
نویسنده: غلامرضا ربال
تاریخ: ۱۳۹۴/۰۴/۲۷ ۹:۱۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, MVC, Reflection, ASP.NET Identity, Caching

طی این مقاله، **نحوه‌ی ذخیره سازی تنظیمات متغیر و پویای یک برنامه را به صورت Strongly Typed** ارائه خواهم داد. برای این منظور، یک API را که از Reflection ، Cache ، Lazy Loading و Entity Framework بهره میگیرد، خواهیم ساخت.
برنامه‌ی هدف ما که از این API استفاده می‌کند، یک اپلیکیشن Asp.net MVC است. قبل از شروع به ساخت API مورد نظر، یک دید کلی در مورد آنچه که قرار است در نهایت توسعه یابد، در زیر مشاهده میکنید:

```
public SettingsController(ISettings settings)
{
  // example of saving
  _settings.General.SiteName = "دات نت تیپس";
  _settings.Seo.HomeMetaTitle = ".Net Tips";
  _settings.Seo.HomeMetaKeywords = "Asp.net MVC,Entity Framework,Reflection";
  _settings.Seo.HomeMetaDescription = "ذخیره تنظیمات برنامه";
  _settings.Save();
}
```

همانطور که در کدهای بالا مشاهده میکنید، شی _setting ما دارای دو پراپرتی فقط خواندنی بنام‌های General و Seo است که شامل تنظیمات مورد نظر ما هستند و این دو کلاس از کلاس پایه‌ی SettingBase ارث بری کرده‌اند. دو دلیل برای انجام این کار وجود دارد:
تنظیمات به صورت گروه بندی شده در کنار هم قرار گرفته‌اند و یافتن تنظیمات برای زمانی که نیاز به دسترسی به آنها داریم، راحت‌تر و ساده‌تر خواهد بود.

به این شکل تنظیمات قابل دسترس در یک گروه، از دیتابیس بازیابی خواهند شد.

اصلا چرا باید این تنظیمات را در دیتابیس ذخیره کنیم؟

شاید فکر کنید چرا باید تنظیمات را در دیتابیس ذخیره کنیم در حالی که فایل web.config در درسترس است و می‌توان توسط کلاس ConfigurationManager به اطلاعات آن دسترسی داشت. **جواب:** دلیل این است که با تغییر فایل web.config، برنامه‌ی وب شما ری استارت خواهد شد (چه زمان‌هایی یک برنامه Asp.net ری استارت میشود).
برای جلوگیری از این مساله، راه حل مناسب برای ذخیره سازی اطلاعاتی که نیاز به تغییر در زمان اجرا دارند، استفاده از از دیتابیس می‌باشد. در این مقاله از Entity Framework و پایگاه داده Sql Sever استفاده می‌کنم.

مراحل ساخت Setting API مورد نظر به شرح زیر است:
ساخت یک Asp.net Web Application

ساخت مدل Setting و افزودن آن به کانتکست Entity Framework

ساخت کلاس SettingBase برای بازیابی و ذخیره سازی تنظیمات با رفلکشن

ساخت کلاس GenralSettins و SeoSettings که از کلاس SettingBase ارث بری کرده‌اند.

ساخت کلاس Settings به منظور مدیریت تمام انواع تنظیمات

یک برنامه‌ی Asp.Net Web Application را از نوع MVC ایجاد کنید. تا اینجا مرحله‌ی اول ما به پایان رسید؛ چرا که ویژوال استودیو کارهای مورد نیاز ما را انجام خواهد داد.

لازم است مدل خود را به ApplicationDbContext موجود در فایل IdentityModels.cs معرفی کنیم. به شکل زیر:

```
namespace DynamicSettingAPI.Models
{
    public interface IUnitOfWork
    {
        DbSet<Setting> Settings { get; set; }
        int SaveChanges();
    }
}

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>,IUnitOfWork
    {
        public DbSet<Setting> Settings { get; set; }
        public ApplicationDbContext()
            : base("DefaultConnection", throwIfV1Schema: false)
        {
        }

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


namespace DynamicSettingAPI.Models
{
    public class Setting
    {
        public string Name { get; set; }
        public string Type { get; set; }
        public string Value { get; set; }
    }
}
```

مدل تنظیمات ما خیلی ساده است و دارای سه پراپرتی به نام‌های Value ، Type ، Name هست که به ترتیب برای دریافت مقدار تنظیمات، نام کلاسی که از کلاس SettingBase ارث برده و نام تنظیمی که لازم داریم ذخیره کنیم، در نظر گرفته شده‌اند.

لازم است تا متد OnModelCreating مربوط به ApplicationDbContext را نیز تحریف کنیم تا کانفیگ مربوط به مدل خود را نیز اعمال نمائیم.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Setting>()
                    .HasKey(x => new { x.Name, x.Type });

            modelBuilder.Entity<Setting>()
                        .Property(x => x.Value)
                        .IsOptional();

            base.OnModelCreating(modelBuilder);
        }
```

ساختاری به شکل زیر مد نظر ماست:

Max Rows: 1000

Name	Type	Value
AdminEmail	GeneralSettings	NULL
HomeMetaDescription	SeoSettings	Welcome to Talk Sharp
HomeMetaTitle	SeoSettings	NULL
SiteName	GeneralSettings	Talk Sharp
NULL	NULL	NULL

کلاس SettingBase ما همچین ساختاری را خواهد داشت:

```
namespace DynamicSettingAPI.Service
{
    public abstract class SettingsBase
    {
        //1
        private readonly string _name;
        private readonly PropertyInfo[] _properties;

        protected SettingsBase()
        {
            //2
            var type = GetType();
            _name = type.Name;
            _properties = type.GetProperties();
        }

        public virtual void Load(IUnitOfWork unitOfWork)
        {
            //3 get setting for this type name
            var settings = unitOfWork.Settings.Where(w => w.Type == _name).ToList();

            foreach (var propertyInfo in _properties)
            {
                //get the setting from setting list
                var setting = settings.SingleOrDefault(s => s.Name == propertyInfo.Name);
                if (setting != null)
                {
                    //4 set
                    propertyInfo.SetValue(this, Convert.ChangeType(setting.Value,
propertyInfo.PropertyType));
                }
            }
        }
        public virtual void Save(IUnitOfWork unitOfWork)
        {
            //5 get all setting for this type name
            var settings = unitOfWork.Settings.Where(w => w.Type == _name).ToList();

            foreach (var propertyInfo in _properties)
            {
                var propertyValue = propertyInfo.GetValue(this, null);
                var value = (propertyValue == null) ? null : propertyValue.ToString();

                var setting = settings.SingleOrDefault(s => s.Name == propertyInfo.Name);
                if (setting != null)
                {
                    // 6 update existing value
                    setting.Value = value;
                }
                else
                {
```

```
                    // 7 create new setting
                    var newSetting = new Setting()
                    {
                        Name = propertyInfo.Name,
                        Type = _name,
                        Value = value,
                    };
                    unitOfWork.Settings.Add(newSetting);
                }
            }
        }
    }
}
```

این کلاس قرار است توسط کلاس‌های تنظیمات ما به ارث برده شود و در واقع کارهای مربوط به رفلکشن را در این کلاس کپسوله کرده‌ایم. همانطور که مشخص است ما دو فیلد را به نام‌های _name و _properties به صورت فقط خواندنی در نظر گرفته ایم که نام کلاس مورد نظر ما که از این کلاس به ارث خواهد برد، به همراه پراپرتی‌های آن، در این ظرف‌ها قرار خواهند گرفت.
متد Load وظیفه‌ی واکشی تمام تنظیمات مربوط به Type و ست کردن مقادیر به دست آمده را به خصوصیات کلاس ما، برعهده دارد. کد زیر مقدار دریافتی از دیتابیس را به نوع داده پراپرتی مورد نظر تبدیل کرده و نتیجه را به عنوان Value پراپرتی ست میکند.

```
propertyInfo.SetValue(this, Convert.ChangeType(setting.Value, propertyInfo.PropertyType));
```

متد Save نیز وظیفه‌ی ذخیره سازی مقادیر موجود در خصوصیات کلاس تنظیماتی را که از کلاس SettingBase ما به ارث برده است، به عهده دارد.
این متد دیتاهای موجود دردیتابیس را که متعلق به کلاس ارث برده مورد نظر ما هستند، واکشی میکند و در یک حلقه، اگر خصوصیتی در دیتابیس موجود بود، آن را ویرایش کرده وگرنه یک رکورد جدید را ثبت میکند.

کلاس‌های تنظیمات شخصی سازی شده خود را به شکل زیر تعریف میکنیم :

```
public class GeneralSettings : SettingsBase
    {
        public string SiteName { get; set; }
        public string AdminEmail { get; set; }
        public bool RegisterUsersEnabled { get; set; }
    }

 public class GeneralSettings : SettingsBase
    {
        public string SiteName { get; set; }
        public string AdminEmail { get; set; }
    }
```

نیازی به توضیح ندارد.
برای اینکه تنظیمات را به صورت یکجا داشته باشیم و Abstraction ای را برای استفاده از این API ارائه دهیم، یک اینترفیس و یک کلاس که اینترفیس مذکور را پیاده کرده است در نظر میگیریم:

```
public interface ISettings
{
    GeneralSettings General { get; }
    SeoSettings Seo { get; }
    void Save();
}

public class Settings : ISettings
{
    // 1
    private readonly Lazy<GeneralSettings> _generalSettings;
    // 2
    public GeneralSettings General { get { return _generalSettings.Value; } }

    private readonly Lazy<SeoSettings> _seoSettings;
    public SeoSettings Seo { get { return _seoSettings.Value; } }
```

```
    private readonly IUnitOfWork _unitOfWork;
    public Settings(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
        // 3
        _generalSettings = new Lazy<GeneralSettings>(CreateSettings<GeneralSettings>);
        _seoSettings = new Lazy<SeoSettings>(CreateSettings<SeoSettings>);
    }

    public void Save()
    {
        // only save changes to settings that have been loaded
        if (_generalSettings.IsValueCreated)
            _generalSettings.Value.Save(_unitOfWork);

        if (_seoSettings.IsValueCreated)
            _seoSettings.Value.Save(_unitOfWork);

        _unitOfWork.SaveChanges();
    }
    // 4
    private T CreateSettings<T>() where T : SettingsBase, new()
    {
        var settings = new T();
        settings.Load(_unitOfWork);
        return settings;
    }
}
```

این اینترفیس مشخص می‌کند که ما به چه نوع تنظیماتی، دسترسی داریم و متد Save آن برای آپدیت کردن تنظیمات، در نظر گرفته شده است. هر کلاسی که از کلاس SettingBase ارث بری کرده را به صورت فیلد فقط خواندنی و با استفاده از کلاس Lazy درون آن ذکر میکنیم و به این صورت کلاس تنظیمات ما زمانی ساخته خواهد شد که برای اولین بار به آن دسترسی داشته باشیم.

متد CreateSetting وظیفه‌ی لود دیتا را از دیتابیس، بر عهده دارد که برای این منظور، متد لود Type مورد نظر را فراخوانی میکند. این متد وقتی به کلاس تنظیمات مورد نظر برای اولین بار دسترسی پیدا کنیم، فراخوانی خواهد شد.

حتما امکان این وجود دارد که شما از امکان Caching هم بهره ببرید برای مثال همچین متد و سازنده‌ای را در کلاس Settings در نظر بگیرید:

```
private readonly ICache _cache;
public Settings(IUnitOfWork unitOfWork, ICache cache)
{
    // ARGUMENT CHECKING SKIPPED FOR BREVITY
    _unitOfWork = unitOfWork;
    _cache = cache;
    _generalSettings = new Lazy<GeneralSettings>(CreateSettingsWithCache<GeneralSettings>);
    _seoSettings = new Lazy<SeoSettings>(CreateSettingsWithCache<SeoSettings>);
}

private T CreateSettingsWithCache<T>() where T : SettingsBase, new()
{
    // this is where you would implement loading from ICache
    throw new NotImplementedException();
}
```

در آخر هم به شکل زیر میتوان (به عنوان دمو فقط) از این API استفاده کرد.

```
public ActionResult Index()
        {
            using (var uow = new ApplicationDbContext())
            {
                var _settings = new Settings(uow);
                _settings.General.SiteName = "دات نت تیپس";
                _settings.General.AdminEmail = "admin@gmail.com";
                _settings.General.RegisterUsersEnabled = true;
                _settings.Seo.HomeMetaTitle = ".Net Tips";
                _settings.Seo.MetaKeywords = "Asp.net MVC,Entity Framework,Reflection";
                _settings.Seo.HomeMetaDescription = "ذخیره تنظیمات برنامه";

                var settings2 = new Settings(uow);
```

```
                var output = string.Format("SiteName: {0} HomeMetaDescription: {1}  MetaKeywords:  {2}
MetaTitle:  {3}  RegisterEnable:  {4}",
                    settings2.General.SiteName,
                    settings2.Seo.HomeMetaDescription,
                    settings2.Seo.MetaKeywords,
                    settings2.Seo.HomeMetaTitle,
                    settings2.General.RegisterUsersEnabled.ToString()
                    );
                return Content(output);
            }

        }
```

خروجی :

نکته: در پروژه ای که جدیدا در سایت ارائه داده‌ام و در حال تکمیل آن هستم، از بهبود یافته‌ی این مقاله استفاده می‌شود. حتی برای اسلاید شوهای سایت هم میشود از این روش استفاده کرد و از فرمت json بهره برد برای این منظور. حتما در پروژه‌ی مذکور همچین امکانی را هم در نظر خواهم گرفتم.

پیشنها میکنم سورس SmartStore را بررسی کنید. آن هم به شکل مشابهی ولی پیشرفته‌تر از این مقاله، همچین امکانی را دارد.

DynamicSettingAPI.zip

عنوان: **نرمال سازی اطلاعات کاربران در حین ثبت نام**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۵/۰۷ ۱۴:۲۰
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, MVC, Security, ASP.NET Identity

شرایط دنیای واقعی، بسیار متفاوت است از طراحی‌های ساده‌ی اولیه‌ی ثبت نام. در طراحی‌های ساده، ایمیل، نام کاربری و بسیاری از اطلاعات دیگر باید منحصربفرد باشند. ایندکس منحصربفرد تعریف می‌کنید. قیود و اعتبار سنجی سمت سرور و سمت کاربر را اضافه می‌کنید. چقدر عالی! اما ... دنیای واقعی شکل دیگری را دارد!
یک روز با ایمیل username@gmail.com ثبت نام می‌کنند. فردا با ایمیل user.name@gmail.com ثبت نام خواهند کرد. پس فردا با ایمیل us.er.name@gmail.com و به همین ترتیب! امروز با نام «کاربر یک» ثبت نام می‌کنند. فردا با نام «کاربر یک»! امروز با نام «مجید» ثبت نام می‌کنند، فردا با نام «مـجـــیـــد»! همچنین علاقه‌ی شدیدی هم به استفاده از ایمیل‌های fake دارند (راه حل).
بنابراین نیاز است اطلاعات کاربران را پیش از ثبت نام نرمال سازی کرد. برای مثال نقطه‌های ایمیل‌های جیمیل را حذف کرد؛ یا اگر اجازه داده‌اید که در بین کلمات نام کاربری، فاصله‌ای را وارد کنند، فقط یک فاصله مجاز باشد و یا اگر نامی را ثبت می‌کنید، به فکر حالت‌های کش آمده‌ی آن مانند «مـجـــــــیـــــــــد» هم باید بود و آن‌را تبدیل به حالت اصلی‌اش کرد.

نرمال سازی ایمیل‌های gmail

تا جایی که اطلاع دارم، حداقل فیس بوک و جی‌میل، بکارگیری نقاط را در ایمیل‌ها مجاز می‌دانند. برای مثال ترکیب‌های زیر از دید gmail تنها یک ایمیل محسوب می‌شوند:

johndoe@gmail.com

john....doe@gmail.com

johndoe+spamsite@gmail.com

راه حل پیشنهادی:

```
public static string FixGmailDots(string email)
{
    if (string.IsNullOrWhiteSpace(email))
        return string.Empty;

    email = email.ToLowerInvariant().Trim();
    var emailParts = email.Split('@');
    var name = emailParts[0].Replace(".", string.Empty).Replace("+", string.Empty);
    var emailDomain = emailParts[1];

    string[] domainsAllowedDots =
    {
        "gmail.com",
        "facebook.com"
    };

    var isFromDomainsAllowedDots = domainsAllowedDots.Any(domain => emailDomain.Equals(domain));
    return !isFromDomainsAllowedDots ? email : string.Format("{0}@{1}", name, emailDomain);
}
```

در اینجا بررسی می‌شود که آیا دومین ایمیل دریافتی از سمت جیمیل یا فیس بوک است؟ اگر بله، آنگاه نقطه‌ها و +های آن‌ها حذف می‌شوند.
این بررسی باید در حین ثبت نام و همچنین ویرایش اطلاعات کاربری جهت نرمال سازی اطلاعات اعمال شود.
اگر سایت‌های دیگری هم هستند که بکارگیری نقاط را مجاز می‌دانند، آرایه‌ی domainsAllowedDots را تکمیل کنید.

نرمال سازی ورود حروف ویژه

نرمال سازی ابتدایی ثبت نام کاربران در سایت جاری به صورت ذیل است:

```
friendlyName =
```

```
friendlyName.ApplyCorrectYeKe().RemoveDiacritics().CleanUnderLines().RemovePunctuation();
var trimmedFriendlyName = friendlyName.Trim().Replace(" ", "");
```

با ApplyCorrectYeKe آشنایی دارید؛ همان یک دست کردن ی و ک فارسی و عربی است.

RemoveDiacritics همان حذف اعراب از کلمات است است.

متد پاکسازی underlineهای ویژه یا همان نام‌های کش آمده، به صورت زیر است:

```
public static string CleanUnderLines(string text)
{
    if (string.IsNullOrWhiteSpace(text))
        return string.Empty;

    const char chr1600 = (char)1600; //1600=ـ
    const char chr8204 = (char)8204; //=8204

    return text.Replace(chr1600.ToString(CultureInfo.InvariantCulture), "")
               .Replace(chr8204.ToString(CultureInfo.InvariantCulture), "");
}
```

و متد RemovePunctuation :

```
public static string RemovePunctuation(string text)
{
 return string.IsNullOrWhiteSpace(text) ? string.Empty : new string(text.Where(c =>
!char.IsPunctuation(c)).ToArray());
}
```

این موارد، «حداقل»هایی هستند که باید جهت نرمال سازی اطلاعات، در حین ثبت نام اعمال شوند.

نظرات خوانندگان

نویسنده: مجتبی آزاد
تاریخ: ۱۳۹۴/۰۵/۰۷ ۱۸:۴۳

سلام
طبق دستوری که برای حذف کردن علامت + نوشته شده، بعد از اجرای کد
ایمیل johndoe+spamsite@gmail.com به johndoespamsite@gmail.com تبدیل می‌شود. آیا بهتر یا صحیح‌تر نیست که مقدار بعد از علامت + حذف شود و ایمیل به این شکل در بیایید: johndoe@gmail.com ؟

عنوان: تزریق وابستگی‌های رایج ASP.NET MVC به برنامه
نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۴/۰۵/۱۷ ۲۳:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, MVC, Refactoring, Dependency Injection, ASP.NET Identity

در پروژه خود می‌توانیم StructureMap را به گونه‌ایی تنظیم کنیم که کار تزریق لایه‌های انتزاعی ASP.NET را نیز انجام دهد؛ مثلاً CurrentHttpContext و یا داده‌های مربوط به مسیریابی و... به عنوان مثال در برنامه شما ممکن است کدهای زیر چندین و چند بار تکرار شده باشند:

```
var userId= User.Identity.GetUserId();
var user = _context.Users.Find(userId);

var user = int.Parse(User.Identity.GetUserId());
```

کدهای فوق به این معنی است که پروژه‌ی شما به صورت کامل به سیستم ASP.NET Identity گره خورده است. خوب، این حالت زمانی پیچیده‌تر خواهد شد که در آینده بخواهید به یک سیستم Identity جدیدتر مهاجرت کنید.
در ادامه نحوه‌ی تزریق وابستگی‌های رایج ASP.NET را بررسی خواهیم کرد. ابتدا یک کلاس رجستری را به صورت زیر ایجاد خواهیم کرد:

```
public class CommonASPNETRegistry : StructureMap.Configuration.DSL.Registry
{
        public CommonASPNETRegistry()
        {
            For<IIdentity>().Use(() => HttpContext.Current.User.Identity);
            // Other dependencies
        }
}
```

در کد فوق همانطور که مشخص است، یک کلاس ریجستری ایجاد کرده‌ایم (Registry در واقع یکی از مفاهیم مربوط به استراکچرمپ می‌باشد که امکان ماژولار کردن تنظیمات را درون کلاس‌هایی مجزا، در اختیارمان قرار می‌دهد). درون سازنده‌ی این کلاس گفته‌ایم: زمانیکه درخواستی برای اینترفیس IIdentity داده شد، یک وهله از HttpContext.Current.User.Identity را در اختیار درخواست کننده قرار بده.
لازم به ذکر است می‌توانستیم از وابستگی‌های عنوان شده نیز بدون تزریق کردن آنها درون کنترلرها نیز استفاده کنیم. اما ریجستر کردن آنها این امکان را در اختیارمان قرار می‌دهد تا در هر جایی از برنامه‌مان بتوانیم به آنها دسترسی پیدا کنیم. در ادامه خواهید دید که دسترسی آسان به آنها می‌تواند خیلی مفید واقع شود؛ همچنین امکان تست کردن نیز آسانتر خواهد شد.
قدم بعدی افزودن Registry ایجاد شده به تنظیمات IoC Containerمان است:

```
public static class SmObjectFactory
{
        private static readonly Lazy<Container> _containerBuilder =
            new Lazy<Container>(defaultContainer, LazyThreadSafetyMode.ExecutionAndPublication);

        public static IContainer Container
        {
            get { return _containerBuilder.Value; }
        }

        private static Container defaultContainer()
        {
            return new Container(ioc =>
            {
                // Other settings
                ioc.AddRegistry(new CommonASPNETRegistry());

            });
        }
}
```

اکنون به سادگی می‌توانیم از وابستگی‌های عنوان شده در برنامه‌مان استفاده کنیم. برای استفاده‌ی از آن، مثال اول را در نظر

بگیرید " **یافتن کاربر فعلی** ". همانطور که عنوان شد، استفاده از کدهایی شبیه به حالت زیر جهت یافتن کاربر جاری در برنامه ممکن است چندین بار تکرار شده باشد:

```
var user = int.Parse(User.Identity.GetUserId());
```

خوب، برای حل این مشکل اینترفیس زیر را اضافه می‌کنیم:

```
public interface ICurrentUser
{
        ApplicationUser User { get; }
}
```

پیاده‌سازی آن نیز به این صورت خواهد بود:

```
public class CurrentUser : ICurrentUser
{
        private readonly IIdentity _identity;
        private readonly IApplicationUserManager _userManager;
        private ApplicationUser _user;
        public CurrentUser(IIdentity identity, IApplicationUserManager userManager)
        {
            _identity = identity;
            _userManager = userManager;
        }
        public ApplicationUser User
        {
            get { return _user ?? (_user = _userManager.FindById(int.Parse(_identity.GetUserId()))); }
        }
}
```

درون کلاس فوق به اینترفیس IIdentity جهت ارائه آی‌دی کاربر جاری و اینترفیس IApplicationUserManager جهت یافتن اطلاعات کاربر نیاز خواهیم داشت. همانطور که مشاهده می‌کنید فیلد user_ در صورتیکه از قبل موجود باشد، برگردانده خواهد شد؛ در غیر اینصورت آن را از کانتکست مربوطه واکشی خواهد کرد.

اکنون با استفاده از روش فوق نه تنها درون کنترلرهایمان بلکه در هر جایی از برنامه‌مان می‌توانیم به کاربر جاری دسترسی داشته باشیم. همچنین در آینده نیز به راحتی می‌توانیم از سیستم ASP.NET Identity به هر سیستم دیگری سوئیچ کنیم.

برای استفاده از اینترفیس فوق نیز به این صورت عمل خواهیم کرد:

```
public class HomeController : BaseController
{
    private readonly ICurrentUser _currentUser;
    public HomeController(ICurrentUser user)
    {
        _user = user;
    }
    public ActionResult Index()
    {
        // user
        var user = _currentUser.User;
        // user id
        var userId = _currentUser.User.Id;
    }
}
```

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۵/۲۲ ۱۳:۱۸

با تشکر. مثال AspNetIdentityDependencyInjectionSample جهت تزریق IIdentity به روز شد. با این تغییرات .

نویسنده: غلامرضا ربال
تاریخ: ۱۳۹۴/۰۶/۲۰ ۱۸:۱۱

با تشکر.
با تنظیمات زیر

```
ioc.For<IIdentity>()
                    .Use(
                        () =>
                            (HttpContext.Current != null && HttpContext.Current.User != null)
                                ? HttpContext.Current.User.Identity
                                : null);
```

همیشه مقدار تزریق شده در کلاس سرویس کاربر ، null میباشد.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۲۰ ۲۰:۴۲

HttpContext در یک سری روال‌های آغازین برنامه هنوز مقدار دهی نشده و نال هست. ولی پس از آن خیر:

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۴/۰۶/۲۷ ۱۶:۵۶

سلام مهندس تشکر بابت مطلب خوب شما
یک قسمت کد ظاهرا سهوا اشتباهی نوشته شده لطفا اصلاحش کنید

```
private readonly ICurrentUser _currentUser;
    public HomeController(ICurrentUser user)
```

```
    {
        _user = user;
    }
```

که به جای آن در سازنده باید قسمت زیر نوشته باشه

```
private readonly ICurrentUser _currentUser;
    public HomeController(ICurrentUser user)
    {
        _currentUser = user;
    }
```

عنوان: IdentityServer قسمت اول
نویسنده: ناصرنیازی
تاریخ: ۱۳۹۴/۰۵/۲۸ ۱۸:۰
آدرس: www.dotnettips.info
گروه‌ها: Security, OpenID, ASP.NET Identity, IdentityServer, OAuth2

استفاده از سرویس‌های متنوع گوگل همگی با یک آکانت واحد، ایده‌ی جالبی است که پایه‌ی ایجاد پروژه‌ای به نام IdentityServer بوده است. IdentityServer یک پروژه‌ی متن باز است که قرار بود و شاید هنوز هم هست که بخشی از ویژوال استودیو باشد.

این پروژه یک سرور واحد برای مدیریت هویت ایجاد می‌کند که تمام کلاینت‌ها از این سرور اهراز هویت شده و سپس از سرویس‌ها استفاده می‌کنند. یعنی بخش مدیریت هویت تمام کاربران در پروژه برعهده‌ی IdentityServer گذاشته می‌شود و همه‌ی برنامه‌ها هویت کاربران را از IdentityServer می پرسند.
تصویری که توسعه دهندگان این پروژه برای معماری پروژه خود ارائه داده‌اند:

برای استفاده از آی‌دنتیتی‌سرور، ابتدا آن را از مخزن گیت‌هاب، بارگذاری می‌کنیم و سپس برای پروژه، یک Application جدید در IIS ایجاد می‌کنیم.
دقت داشته باشید که IdentityServer از SSL و بستر امن استفاده می‌کند که تنظیمات ساخت Certificate را می‌توانید از اینجا فرا بگیرید.
با توجه به پشتیبانی گسترده‌ی این پروژه از OpenID و OAuth2 مطالعه‌ای مختصر در این موارد به درک فرایند اهراز هویت توسط آی دنتیتی سرور بسیار کمک خواهد کرد.
پس از راه اندازی SSL و تنظیمات Certificate مربوط به آن می‌توانید شروع به راه اندازی سرور خود کنید. راه اندازی اولیه سرور تنظیمات مربوط به بانک اطلاعاتی، نکات ریزی دارد که بخش کلی از آن اینجا آمده است.
برای شروع به استفاده از سرور و درک چگونگی عملکرد آن نیاز دارید تا مدیر سرور را نصب کنید؛ تا هم بتوانید کاربر تعریف کنید و هم نقش‌ها (Roles) و دسترسی‌ها را مدیریت کنید. نگارش مدیر آی‌دنتیتی‌سرور نیز از اینجا قابل دسترسی می‌باشد.
پس از نصب آی‌دنتیتی‌سرور باید تنظیمات مربوط به ذخیره سازی اطلاعات آن را انجام دهید که آی‌دنتیتی‌سرور پیاده سازی خوبی برای Entity Framework دارد که می‌توانید با نصب آن همه‌ی کارهای ذخیره سازی در بانک اطلاعاتی را به EF بسپارید. البته برای

ذخیره‌ی یوزر می‌توان از حالت InMemory هم استفاده کرد که در نسخه‌ی مثال یک کاربر با نام bob و رمز bob در داخل کد نویسی تعریف شده بود، که می‌توان برای پروژه‌های کوچک همان را توسعه داد.

در سطح بانک هم آی‌دنتیتی‌سرور از دو بانک اطلاعاتی، استفاده می‌کند؛ یکی برای ذخیره‌ی تنظیمات سرور و دیگری برای ذخیره‌ی اطلاعات هویتی.

مزیت بزرگ آی‌دنتیتی‌سرور در اعتبار سنجی جدای از پیاده سازی‌های فراوان انجام شده برای محیط‌های مختلف و در قابلیت اعتبار سنجی دو طرفه‌ی آن می‌باشد.

یعنی هم سمت سرور و هم سمت کلاینت و هم سرویس‌هایی که از آی‌دنتیتی‌سرور استفاده می‌کنند، باید اعتبار سنجی شوند و همه چیز به یک رمز و نام کاربری ساده خلاصه نمی‌شود.

در زمان نگارش این متن، نسخه‌ی 2 نسخه‌ی پایدار ارائه شده است و نسخه‌ی سه در مرحله تست می‌باشد. البته پیاده سازی‌هایی هم از نسخه‌ی 3 در محیط‌های مختلف مثل MVC و WEB API ارائه شده است؛ ولی هنوز به پایداری لازم نرسیده است.

نظرات خوانندگان

نویسنده: مسعود دانش پور
تاریخ: ۱۳۹۴/۰۵/۳۰ ۲۲:۳۳

من سه سالی هست دارم سعی می‌کنم این پروژه رو نهایی کنم.ولی هنوز به دلیل مشغله زیاد موفق نشدم.

عنوان: **IdentityServer قسمت دوم**
نویسنده: ناصرنیازی
تاریخ: ۱۳۹۴/۰۶/۰۳ ۱:۰
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, Security, ASP.NET Identity, IdentityServer

پس از تلاش‌های اولیه برای راه اندازی که نیاز به گوگل کردن موارد مختلف دارد از جمله راه اندازی ssl و certification در لوکال هاست و تنظیم IIS برای استفاده از آن، می‌توان به راه اندازی اولیه آی‌دن‌تی‌تی سرور رسید .
پیش فرض این آموزش این نسخه از آی دنتیتی سرور است :
https://github.com/IdentityServer/IdentityServer2

نگاهی اجمالی به سورس: IdentityServer2

```
Sampel----
```

```
AdfsIntegrationFullSample --------
```

```
AdfsIntegrationSampleClient --------
```

```
InMemoryHost --------
```

```
(MVC and WCF RP (SAML --------
```

```
(MVC and Web API (JWT --------
```

```
MembershipRebootUserRepository --------
```

```
OIDC --------
```

```
SelfHostConsoleHost --------
```

```
ServiceBus Integration --------
```

```
src----
```

```
Libraries --------
```

```
WebSite --------
```

```
Tests --------
```

از نام مثال‌ها کاملاً مشخص است که هر کدام چه بخشی را پوشش می‌دهند.
مثلاً پوشه‌ی MembershipRebootUserRepository، مربوط به مدیریت کاربران است و توابع غنی بسیار خوبی در آن استفاده شده‌است. خود MembershipReboot یک پروژه‌ی دیگر است برای مدیریت کاربران که توسط BrockAllen توسعه داده شده و مدیریت کاربران را بسیار آسان کرده‌است؛ برای مثال به راحتی می‌توان

```
BrockAllen.MembershipReboot
BrockAllen.MembershipReboot.Ef
BrockAllen.MembershipReboot  .Repository
```

از این آدرس را به سورس اصلی آی دنتیتی سرور اضافه کرد و از توابع مدیریت کاربران این پروژه استفاده کرد. چون کار را بسیار آسان کرده است.
برای مثال من برای ایجاد کاربر نیاز داشتم که علاوه بر درج کاربر در سرور آی دنتیتی، در بانک اطلاعاتی خودم هم کاربر درج شود. برای همین برای آی دنتیتی سرور صفحه‌ی ثبت نام نوشتم. برای اینکار فقط با یک شیء از membership به راحتی این تابع پیاده سازی شد:

```
Membership.CreateUser(userName, password, email);
          Roles.AddUserToRoles(userName, "IdentityServerUsers");
```

نمونه استفاده

این افزودن کاربر را می‌توان از هسته‌ی اصلی خود آی دنتیتی سرور استفاده کرد. ولی استفاده از آن به علت راحت بودن و توابع خیلی زیاد BrockAllen.MembershipReboot است و استفاده از آن بسیار کاربردی می‌تواند باشد.

برای استفاده از خود آی دنتیتی سرور برای ساخت یوزر، وارد قسمت سورس شوید و به کنترلر احراز هویت، یک تابع به نام رجیستر اضافه کنید؛ به این فایل:

https://github.com/IdentityServer/IdentityServer2/blob/master/src/OnPremise/WebSite/Controller/AccountController.cs

شکل تابع :

```
[HttpPost]
public ActionResult Register(RegisterModel model)
 {
         UserRepository.CreateUser(model.userName,model.password,model.email);
         Roles.AddUserToRoles(userName, "IdentityServerUsers");
         return  View(model);
}
```

خود کلاس UserRepository یک کلاس پیاده سازی شده از اینترفیس IUserManagementRepository است که می‌توان خود آن‌را نیز با تزریق وابستگی‌ها تغییر داد و از Membership و بازنویسی توابع قدرتمند MembershipReboot آن استفاده کرد. لیست توابع این اینترفیس که می‌توانید استفاده کنید:

```
void CreateUser(string userName, string password, string email = null);
        void DeleteUser(string userName);

        IEnumerable<string> GetUsers(int start, int count, out int totalCount);
        IEnumerable<string> GetUsers(string filter, int start, int count, out int totalCount);

        void SetPassword(string userName, string password);

        void SetRolesForUser(string userName, IEnumerable<string> roles);
        IEnumerable<string> GetRolesForUser(string userName);

        IEnumerable<string> GetRoles();
        void CreateRole(string roleName);
        void DeleteRole(string roleName);
```

آدرس اینترفیس

عنوان: اعتبارسنجی درخواست های $http با استفاده از یک Interceptor
نویسنده: محمود راستین
تاریخ: ۱۳۹۴/۰۶/۱۱ ۱۰:۰
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, MVC, Security, AngularJS, ASP.NET Identity

پیش نیاز :

هدایت خودکار کاربر به صفحه لاگین در حین اعمال Ajax ایی
Angular Interceptors

ابتدا مشکل و هدف را بیان می‌کنیم:

مشکل : کاربر در صفحه‌ای حضور دارد که نیاز به اعتبارسنجی داشته و مدت اعتبار کاربر نیز تمام شده است، ولی هنوز در صفحه‌ای که نباید حضور داشته باشد، حضور دارد و بدتر از آن این است که می‌تواند درخواست‌های بی نتیجه‌ای را نیز ارسال کند.
هدف : کاربر را سریعا به صفحه‌ای که به آن تعلق دارد هدایت کنیم (یعنی صفحه‌ی ورود به سیستم).

و حالا از ابتدا پروسه را دنبال می‌کنیم. یک Controller سمت سرور داریم به این صورت :

```
[Authorize(Roles = AuthorizeRole.SuperAdministrator)]
    public partial class HomeController : Controller
    {
        [HttpPost]
        [AngularValidateAntiForgeryToken]
        public virtual JsonResult GetUserInfo()
        {
            var userInfoViewModel = _applicationUserManager.GetUserInfoById(User.Identity.GetUserId());
            return Json(userInfoViewModel);
        }
    }
```

همانطور که مشاهده می‌کنید از Authorize پیش فرض استفاده کرده‌ایم. حالا سمت کلاینت با استفاده از HTTP می‌خواهیم درخواستی را ارسال کنیم.

یعنی با کدی همانند کد زیر:

```
$scope.getUserInfo = function() {
            $http({
                    method: 'POST',
                    url: 'Home/GetUserInfo',
                    headers: $scope.getHeaders()
                }).
                success(function(data, status, headers, config) {
                    $scope.userInfo = data;
                }).
                error(function(data, status, headers, config) {

                }).then(function(res) {

                });
        }
```

و نتیجه‌ی کدهای بالا به صورت زیر درخواهد آمد :

```
success(function(data, status, headers, config) {  data = Object {UserName: "dotnettips.info"}, status = 200,
    $scope.userInfo = data;
}).
```

همانطور که می‌بینید داده‌های اولیه کاربر پس از ورود به سیستم، بدون هیچ مشکلی دریافت می‌شوند.

نکته : زمانیکه status برابر با 200 هست، یعنی درخواست OK می‌باشد. (در پیوست ، لیست تمامی کدها قرار داده شده است)

حالا فرض کنید کاربر در صفحه حضور دارد و به هر دلیلی اعتبار حضور کاربر منقضی شده است و حالا پس از مدتی کاربر درخواستی را به سرور ارسال می‌کند و می‌خواهد اطلاعات خودش را مشاهده کند.

درخواست کاربر با همان کدهای اولیه ارسال می‌شود و خروجی اینبار به صورت زیر در خواهد آمد :

```
success(function(data, status, headers, config) {  data = "", status = 200,
    $scope.userInfo = data;
}).
```

همانطور که می‌بینید وضعیت اینبار نیز OK می‌باشد، ولی هیچ داده‌ای از سرور دریافت نشده است. کاربر قطعا در اینجا دچار سردرگمی می‌شود. چون هیچ چیزی را مشاهده نمی‌کند و به هیچ مسیر دیگری نیز هدایت نمی‌شود و هرکار دیگری نیز انجام دهد، پاسخی مشاهده نمی‌کند.

راه حل چیست ؟

ابتدا باید برای درخواست‌های Ajax ایی اعتبارسنجی را اعمال کنیم. برای این کار باید یک Attribute جدید بسازیم. یعنی به این صورت :

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, Inherited = true, AllowMultiple =
true)]
    public class MyCustomAuthorize : AuthorizeAttribute
    {
        protected override void HandleUnauthorizedRequest(AuthorizationContext filterContext)
        {
            if (filterContext.HttpContext.Request.IsAjaxRequest())
            {
                // یعنی اعتبارسنجی نشده است
                filterContext.HttpContext.Response.StatusCode = (int)HttpStatusCode.Unauthorized;
                filterContext.HttpContext.Response.End();
            }

            base.HandleUnauthorizedRequest(filterContext);
        }
    }
```

با استفاده از Attribute بالا ما درخواست‌های ای‌جکسی را نیز اعتبارسنجی می‌کنیم.

توجه : کد کامل‌تر همراه با توضیحات در بخش پیش نیازها آمده است.

حالا در سمت کلاینت نیز به این صورت عمل می‌کنیم :

```
$http({
                    method: 'POST',
                    url: 'Home/GetUserInfo',
                    headers: $scope.getHeaders()
```

```
                }).
                success(function(data, status, headers, config) {
                    $scope.userInfo = data;
                }).
                error(function(data, status, headers, config) {
                    if (status === 401) {
                        // you are not authorized
                    }
                }).then(function(res) {

                });
```

حالا دیگر متوجه خواهیم شد که کاربر اعتبارسنجی نشده است، یا اعتبار آن منقضی شده است و می‌توانیم کاربر را به مسیر **"ورود به سیستم"** هدایت کنیم.

در console مرورگر نیز خطای زیر رخ می‌دهد :

```
POST http://localhost:000000/Administrator/Home/GetUserInfo 401 (Unauthorized)
```

اکنون به هدف خودمان رسیده‌ایم. اما برای هر درخواست باید این دستوارت را تکرار کنیم ؟! قطعا خیر.

حالا باید از یک Interceptor استفاده و درخواست‌های HTTP خودمان را مدیریت کنیم. برای این منظور ما یک Interceptor جدید را همانند کدهای زیر می‌نویسیم:

```
factory('AuthorizedInterceptor', function ($q) {
        return {
            response: function(response) {
                return response || $q.when(response);
            },
            responseError: function(rejection) {
                if (rejection.status === 401) {
                    // you are not authorized
                }
                return $q.reject(rejection);
            }
        };
    })
```

همانطور که مشاهده می‌کنید کدهای خطای درخواست $http را در Interceptor قرار دادیم. حالا کاربر درخواستی را ارسال می‌کند و ما با توجه به این Interceptor متوجه خواهیم شد که آیا اعتبار آن منقضی شده است یا خیر و در صورت منقضی شدن، کاربر را به صفحه‌ی ورود هدایت خواهیم کرد. یعنی بدین صورت :

```
}).factory('AuthorizedInterceptor', function ($q) {
    return {
        response: function(response) {
            return response || $q.when(response);
        },
        responseError: function(rejection) {  rejection = Object {data: "", status: 401, config: Object, statusText: "Unauthorized"}
            if (rejection.status === 401) {
                // you are not authorized
            }
            return $q.reject(rejection);
        }
    };
});
```

در خروجی بالا می‌بینید که کاربر، اعتبارسنجی نشده است و آن را به مسیر ورود هدایت خواهیم کرد.

با Interceptor بالا دیگر نیازی نیست برای هر درخواستی این وضعیت را چک کنیم و به صورت خودکار این چک کردن رخ خواهد داد.

پیوست : http_status_codes.rar

نظرات خوانندگان

نویسنده: یاسر مرادی
تاریخ: ۱۳۹۴/۰۶/۱۳ ۱۳:۱

با سلام و احترام

ممنون بابت مقاله خوبتون

ولی این شرط

```
filterContext.HttpContext.Request.IsAjaxRequest
```

معنی

یعنی اعتبارسنجی نشده است

را نمی‌دهد.

شما در متد HandleUnauthorizedRequest هستید و بالطبع اعتبار درخواست رد شده است، کاری که می‌خواهید انجام دهید، اعتبارسنجی نیست، بلکه اصلاح Response درخواست‌های Ajax ای است که Status Code آنها ۲۰۰ است که نباید باشد.

این نوع دقت داشتن و شفافیت لازمه ایجاد یک کد صحیح است، چون الآن و در مرحله ای بالاتر، می‌توانیم بگوییم که هر Response ای که وارد متد HandleUnauthorizedRequest شده است، اگر Status Code 401 ندارد، Status Code آن برابر ۴۰۱ قرار داده شود. فارغ از Ajax بودن یا نبودن.

و باز در مرحله ای بالاتر اگر در پروژه از Owin استفاده شده باشد، یک Owin Middleware و اگر نه یک ASP.NET Module جای مناسب‌تری برای نوشتن این چنین کدی است.

از این که IsAjaxRequest همیشه درست کار نمی‌کند و فقط وقتی درخواست Ajax ای بگوید که Ajax ای است، کار خواهد کرد نیز بگذریم.

موفق و پایدار باشید.

نویسنده: محسن خان
تاریخ: ۱۳۹۴/۰۶/۱۳ ۱۳:۲۷

نویسنده‌ی گرامی، خواهشمند است جهت رفاه حال خوانندگان گرانقدر، جمله‌ی مبهم «یعنی اعتبارسنجی نشده است» را به جمله‌ی بهتر «یعنی درخواست ای‌جکسی در این مرحله از کار اعتبارسنجی‌اش رد شده‌است و اکنون می‌خواهیم وضعیت خروجی آن‌را اصلاح کنیم» تغییر دهید. با تشکر فراوان!

نویسنده: محمود راستین
تاریخ: ۱۳۹۴/۰۶/۱۴ ۲:۵۶

با سلام
عبارت " یعنی اعتبارسنجی نشده است " برای تعریف خط زیرین خودش هستش که داره پاسخ رو اصلاح می‌کنه و برای شرط

استفاده نشده است که این پاسخ ما با توجه به کد وضعیت ،یعنی اعتبار سنجی نشده است . به بیان ساده‌تر معنی تغییر کد پاسخ می‌باشد.
من شخصا از مطلب جناب نصیری در اینجا استفاده می‌کنم برای این اعتبارسنجی. پیشنهاد میکنم این مطلب و نظرات اون رو ببینید. همچنین این مطلب در پیش نیازها درج شد تا در آینده خوانندگان دچار مشکل نشوند و ابتدا این مطلب را بخوانند.
و جناب مرادی در خط آخر مطلبی رو بیان کردید ، خوشحال میشیم جزئیات بیشتری رو بیان کنید. یعنی چی همیشه درست کار نمی‌کند ؟ چه شرایطی ممکن است رخ دهد که اشتباه کار کند ؟

با تشکر

نویسنده: محسن خان
تاریخ: ۱۳۹۴/۰۶/۱۴ ۱۰:۱۹

چه شرایطی ممکن است رخ دهد که اشتباه کار کند ؟

اگر این درخواست دستی صادر بشه (یک ربات باشه) یا اگر فایروال خاصی در این میان هدرهای Ajax رو حذف کنه . مورد اول مهم نیست، چون نیازی نیست تا ربات را به صفحه‌ی لاگین هدایت کرد (مرورگر باز شده‌ای نداره). مورد دوم هم خیلی خیلی بعید هست. یعنی من ندیدم که در عمل فایروال‌های شرکت‌ها هدرهای HTTP رو تا این حد دستکاری کنند.

عنوان: **جلوگیری از ورود همزمان کاربران با نام کاربری و رمز عبور یکسان**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۴/۰۶/۱۲ ۱۶:۱۰
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, MVC, Security, ASP.NET Identity

در اکثر برنامه‌های وب، کاربر قادر است با یک نام کاربری و رمز عبور در چند Session همزمان لاگین کند. ممکن است سیاست برخی مدیران محصول این باشد که جلوی این مورد را بگیرند تا به عنوان مثال کاربران را به جای استفاده‌ی همزمان از یک نام کاربری و رمز عبور، مجبور به خرید مجوزهای بیشتری کنند. ASP.NET Identity به صورت پیش فرض این مورد را پشتیبانی نمی‌کند؛ اما به کمک استفاده از امکانات درونی آن می‌توان این پشتیبانی را اضافه کرد.
یکی از فیلدهای جدول AspNetUsers فیلد SecurityStamp می‌باشد. SecurityStamp یک مقدار تصادفی است:

Security Stamp باید با هربار تغییر اطلاعات احراز هویت (مانند رمز عبور) و اختیارات کاربر(Role) تغییر کند. به عنوان مثال کاربری در چند مرورگر لاگین کرده و گزینه‌ی مرا به خاطر داشته باش را انتخاب کرده است. اگر این کاربر رمز عبورش از هر جایی عوض شود، باید لاگین او در همه‌ی Sessionها غیر معتبر شود. این مورد با تغییر کردن SecurityStamp بعد از تغییر رمز عبور صورت می‌گیرد. ASP.NET مقدار SecurityStamp را در کوکی کاربر نگه می‌دارد و در بازه‌های زمانی، این مقدار را با مقدار درون دیتابیس مقایسه می‌کند و در صورت عدم برابری، کاربر را احراز هویت نمی‌کند. بازه‌ی زمانی این بررسی در متد ConfigureAuth قابل تنظیم است که در ادامه شرح داده خواهد شد.
صورت مساله یافتن راه حلی جهت جلوگیری از ورود همزمان چند کاربر با یک نام کاربری و رمز عبور به سیستم می‌باشد. یکی از راه‌حل هایی که در ابتدا به ذهن می‌آید استفاده از Session و نگهداری کاربران لاگین کرده در حافظه می‌باشد. پیاده سازی این راه حل می‌تواند به کمک یک کلاس Static صورت پذیرد، اما قسمت چالشی این موضوع این است که چه زمانی باید کاربر از لیست حذف گردد؟ اگر اتصال کاربر قطع شود چه عملی باید صورت گیرد؟
راه حل دیگر استفاده از SecurityStamp هست؛ به این صورت که با هربار لاگین کاربر این مقدار تصادفی به‌روز گردد و ASP.NET Identity به گونه‌ای تنظیم شود که با هر درخواست HTTP، صحت SecurityStamp بررسی گردد. مقدار پیش فرض بازه‌ی زمانی بررسی، هر 30 دقیقه یک بار است.
در مثال‌های رسمی ASP.NET Identity لاگین به صورت ذیل پیاده سازی شده است:

```
[HttpPost]
```

```
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
        {
            if (!ModelState.IsValid)
            {
                return View(model);
            }
            // This doesn't count login failures towards account lockout
            // To enable password failures to trigger account lockout, change to shouldLockout: true
            var result = await SignInManager.PasswordSignInAsync(model.Email, model.Password,
model.RememberMe, shouldLockout: false);
            switch (result)
            {
                case SignInStatus.Success:
                    return RedirectToLocal(returnUrl);
                case SignInStatus.LockedOut:
                    return View("Lockout");
                case SignInStatus.RequiresVerification:
                    return RedirectToAction("SendCode", new { ReturnUrl = returnUrl, RememberMe =
model.RememberMe });
                case SignInStatus.Failure:
                default:
                    ModelState.AddModelError("", "Invalid login attempt.");
                    return View(model);
            }
        }
```

این کد را باید به گونه‌ای تغییر داد که اگر نام کاربری و رمز عبور معتبر بودند، مقدار SeucrityStamp به‌روز گردد. به همین منظور قبل از فراخوانی PasswordSignInAsync کد ذیل اضافه می‌گردد:

```
var loggedinUser = await UserManager.FindAsync(model.Email, model.Password);
            if (loggedinUser != null)
            {
                await UserManager.UpdateSecurityStampAsync(loggedinUser.Id);
            }
```

همانطور که مشاهده می‌شود، جهت بروز رسانی SecurityStamp از سازوکار درونی ASP.NET Identity، در واقع متد UpdateSecurityStampAsync بهره گرفته شده است.

اکنون باید تنظیمات پیش‌فرض بازه‌ی زمانی بررسی صحت SecurityStamp تغییر داده شود. این تنظیمات در فایل Startup.Auth.cs در پوشه‌ی App_Start قرار دارند:

```
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))
                }
            });
```

در کد بالا OnValidateIdentity باید مقدار ذیل را بگیرد:

```
OnValidateIdentity =
                        SecurityStampValidator.OnValidateIdentity<ApplicationUserManager,
ApplicationUser>(
                            TimeSpan.FromMinutes(0), // <-- Note the timer is set for zero
                            (manager, user) => user.GenerateUserIdentityAsync(manager))
```

اکنون Framework موظف است در هر درخواست HTTP، صحت SeucirtyStamp را بررسی کند. بنابراین اگر کاربری در سیستم

لاگین باشد و کاربر دیگری با همان نام کاربری و رمز عبور لاگین کند، کاربر اول از سیستم لاگ اوت می‌شود؛ چرا که مقدار SecurityStamp او دیگر معتبر نیست. باید در نظر گرفته شود این عمل در سیستم‌های با تعداد کاربر زیاد باعث افزایش درخواست‌های دیتابیس می‌شود.

اکنون جهت تست، اگر با مرورگر اول در سیستم لاگین صورت گیرد، سپس با همان اطلاعات در مرورگری دیگر، لاگین صورت گیرد، کاربر اول پس از درخواست بعدی، از سیستم لاگ اوت می‌شود. در مثال انتهای مطلب، صفحه‌ی About به صورت غیر عمومی درآمده که می‌توان بررسی راه حل جاری را در آن صفحه صورت داد.

اگر بخواهیم لاگ اوت شدن کاربر را آنی کنیم، می‌توان در فواصل زمانی مشخصی، یک درخواست Ajax از سمت کلاینت به سرور ارسال کرد و تصدیق هویت کاربر را بررسی کرد:

```
window.setInterval(function() {
            $.ajax({
                url:,
                type: "Post",
                dataType: "json",
                success: function(data) {
                    if (!data) {
                        alert("Someone has logged in using your username and password!");
                        location.reload();
                    }
                }
            });
        }, 60000);
```

در کد بالا به کمک window.setInterval، هر یک دقیقه یک بار لاگین بودن کاربر بررسی می‌گردد و در صورت لاگین نبودن، پیغام لازم به کاربر نمایش داده می‌شود. در نظر داشته باشید، این کد تنها باید در صفحات غیر عمومی قرار داده شود.

کد اکشن بررسی لاگین بودن به سادگی ذیل است:

```
[AllowAnonymous]
        public virtual ActionResult Authenticated()
        {
            return Json(User.Identity.IsAuthenticated);
        }
```

نکته‌ی مهم این است که خصیصه‌ی AllowAnonymous بالای اکشن قرار گرفته باشد تا در صورتیکه Controller به صورت عمومی در دسترس نیست، اکشن همیشه و حتی وقتی کاربر لاگ اوت شده در دسترس باشد. در مثال انتهای مطلب صفحه‌ی About تنها در اختیار کاربران احراز هویت شده قرار گرفته است، بنابراین اگر دو کاربر با اطلاعات یکسانی به سیستم لاگین کنند، کاربر اول پیغام خطای ذیل را گرفته و به صفحه‌ی لاگین می‌رود. این کد در صفحه‌ی About در مثال انتهای مطلب قرار گرفته است:

Guildfor
Js OOP In JavaScript: ...
ct
The page at localhost:62378 says:
Someone has logged in using your username and password!
OK
e.

سورس کد

نظرات خوانندگان

نویسنده: غلامرضا ربال
تاریخ: ۱۳۹۴/۰۶/۱۳ ۱۵:۵۷

با تشکر بابت مقاله مفیدی که منتشر کردید.
خودم هم قصد داشتم مطلبی درباره موضوع مرتبط به این مقاله (SecurityStamp) منتشر کنم که الان با توضیحات کامل شما دیگه لزومی نمی‌بینم این کار را انجام دهم.
فقط باید توجه داشت که مقدار دهی Interval با

```
TimeSpan.FromMinutes(0)
```

به معنای کوئری زدن در هر درخواست از دیتابیس برای رفرش کردن نقش‌های کاربر نیز میباشد. حال اگر سیستم بزرگ بوده و علاوه بر گروه‌های کاربری ، دارای سیستم دسترسی‌ها داینامیک هم باشد امکان دارد زمان گزارش گیری کمی افزایش یابد و با تعداد زیاد کاربران این عمل به صرفه نخواهد بود.
کاری که خودم در پروژه << طراحی فریمورکی برای کار با Asp.net MVC و EF >> انجام دادم به این صورت است که یک فیلد به نام IsChangedPermissions در کلاس کاربر قرار دادم تا هر وقت دسترسی‌ها او تغییر کند این فیلد را با مقدار true مقدار دهی کنم و با این صورت لازم نیست در هر درخواست دسترسی‌های کاربر از دیتابیس واکشی شوند . و اگر لازم بود اکانت کاربر را به صورت آنی غیر فعال کنیم کافیست فیلد SecurityStamp او را با متد یاد شده در مطلب تغییر دهیم که این امر با توجه به مقدار دهی interval با مقدار 0 ، سبب خروج کاربر مورد نظر از حساب خود خواهد شد.
البته لازم است بعد از چک کردن فیلد IsChangedPermissions ، اگر مقدار true را در برداشت آن را false مقدار دهی کنیم تا برای درخواست‌های بعدی مشکلی پیش نیاید.
برای این منظور یک SecurityStampValidator شخصی سازی شده در نظر گرفتم که قسمت مد نظر برای تغییر به صورت زیر است:

```
if (validate)
                {
                    var manager = context.OwinContext.GetUserManager<ApplicationUserManager>();
                    var userId = getUserIdCallback(context.Identity);
                    if (manager != null)
                    {
                        var user = await manager.FindByIdAsync(userId).WithCurrentCulture();
                        var reject = true;
                        // Refresh the identity if the stamp matches, otherwise reject
                        if (user != null && manager.SupportsUserSecurityStamp)
                        {
                            var securityStamp =

context.Identity.FindFirstValue(Constants.DefaultSecurityStampClaimType);
                            if (securityStamp == await
manager.GetSecurityStampAsync(userId).WithCurrentCulture())
                            {
                                reject = false;
                                // Regenerate fresh claims if possible and resign in
                                if (user.IsChangedPermissions && regenerateIdentityCallback != null)
                                {
                                    var identity = await regenerateIdentityCallback.Invoke(manager,
user).WithCurrentCulture();
```

عنوان: **ایجاد سرویس Account Manager با تکنولوژی های Identity 2.1 و Web API 2.2**
نویسنده: مهرداد کاهه
تاریخ: ۱۳۹۴/۰۶/۲۲ ۱۷:۵۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Security, ASP.NET Web API, AngularJS, ASP.NET Identity, JSON Web Token

ASP.NET Identity 2.1 جدیدترین فریم ورک عضویت و مدیریت کاربر است که چندی پیش توسط شرکت مایکروسافت منتشر شد. این سیستم عضویت می‌تواند به تمامی فریم‌ورک‌های دات نتی مانند Web API، MVC و ... متصل گردد.
در این دوره چند قسمتی به همراه یک پروژه‌ی نمونه، نحوه‌ی ارتباط Identity و Web API را نمایش خواهیم داد. در قسمت front-end این پروژه‌ی SPA، ما از AngularJs استفاده خواهیم نمود. قسمت front-end که توسط AngularJs توسعه داده می‌شود از bearet token based authentication استفاده می‌کند. این متد از JSON Web Token برای فرایند Authorization بهره می‌گیرد که به اختصار آن را JWT نیز می‌نامند. این روش سیستم اعتبار سنجی role based بوده و تمامی آنچه را که در یک سیستم membership داریم، پوشش می‌دهد. توجه داشته باشید که برای فهم بهتر تمامی مراحل، ما پروژه را بدون هیچ قالب از پیش تعریف شده‌ای در VS2013 آغاز می‌کنیم.
به دلیل اینکه پروژه در طی چند قسمت انجام می‌پذیرد آن را به بخش‌های زیر تقسیم می‌کنیم.

تنظیمات اولیه ASP.NET Identity 2.1 با Web API
ایجاد Account Confirmation به وسیله Identity به همراه تنظیمات policy برای user name و password
توسعه OAuth Json Web Token Authentication به کمک Web API و Identity
ایجاد یک سیستم Role Based و تایید صلاحیت‌های مربوط به آن
توسعه Web API Claims Authorization در Identity 2.1
توسعه بخش front-end با AngularJs

تنظیمات اولیه ASP.NET Identity 2.1 با Web API

1. تنظیمات ASP.Net Identity 2.1

1-1. ساخت یک پروژه Web API

در ابتدا ما یک empty solution را با نام "AspNetIdentity" همانند شکل مقابل می‌سازیم.

یک ASP.NET Web Application با نام "AspNetIdentity.WebApi" را به این solution اضافه می‌نماییم. در بخش select template ما empty template را انتخاب می‌کنیم. همچنین در قسمت add folders and core references for: نیز هیچیک از گزینه‌ها را انتخاب نمی‌کنیم.

1-2. نصب packageهای مورد نیاز از Nuget

در زیر packageهای مورد نیاز برای ASP.NET Web API و Owin را مشاهده میکنید. همچنین packageهای مربوط به ASP.Net Identity 2.1 نیز در زیر قرار داده شده‌اند.

```
Install-Package Microsoft.AspNet.Identity.Owin -Version 2.1.0
Install-Package Microsoft.AspNet.Identity.EntityFramework -Version 2.1.0
Install-Package Microsoft.Owin.Host.SystemWeb -Version 3.0.0
Install-Package Microsoft.AspNet.WebApi.Owin -Version 5.2.2
Install-Package Microsoft.Owin.Security.OAuth -Version 3.0.0
Install-Package Microsoft.Owin.Cors -Version 3.0.0
```

1-3. اضافه کردن user class و database context

حال که تمامی پکیج‌های مورد نیاز را به پروژه خود اضافه نمودیم، قصد داریم تا اولین کلاس EF را با نام "ApplicationUser" به پروژه اضافه کنیم. این کلاس، کاربری را که قصد ثبت نام در membership system، دارد را نمایش می‌دهد. برای این کار ما یک کلاس جدید را به نام "ApplicationUser" می‌سازیم و کلاس "Microsoft.AspNet.Identity.EntityFramework.IdentityUser" را در آن به ارث می‌بریم.

برای این کار ما یک پوشه‌ی جدید را در برنامه با نام "Infrastructure" می‌سازیم و درون آن کلاس ذکر شده را اضافه می‌کنیم:

```
public class ApplicationUser : IdentityUser
    {
        [Required]
        [MaxLength(100)]
        public string FirstName { get; set; }
```

```
        [Required]
        [MaxLength(100)]
        public string LastName { get; set; }

        [Required]
        public byte Level { get; set; }

        [Required]
        public DateTime JoinDate { get; set; }

    }
```

پس از افزودن کلاس User، نوبت به اضافه نمودن Db Context است. این کلاس وظیفه‌ی ارتباط با پایگاه داده را بر عهده دارد. ما یک کلاس جدید را با نام ApplicationDbContext، به پوشه‌ی Infrastructure اضافه می‌نماییم. کد مربوط به این کلاس به صورت زیر است:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext()
            : base("DefaultConnection", throwIfV1Schema: false)
        {
            Configuration.ProxyCreationEnabled = false;
            Configuration.LazyLoadingEnabled = false;
        }

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

    }
```

همانطور که ملاحظه می‌کنید این کلاس از IdentityDbContext ارث بری نموده است. این کلاس یک نسخه‌ی جدیدتر از DbContext است که تمامی نگاشت‌های Entity Framework Code First را انجام می‌دهد. در ادامه ما یک Connection String را با نام DefaultConnection در فایل web.config اضافه می‌نماییم.

همچنین متد static ایی را با نام Create که در تکه کد فوق از سوی Owin Startup class فراخوانی می‌گردد که در ادامه به شرح آن نیز خواهیم پرداخت.

ConnectionString قرار داده شده در فایل web.config در قسمت زیر قرار داده شده است:

```
<connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=.\sqlexpress;Initial
Catalog=AspNetIdentity;Integrated Security=SSPI;" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

قدم 4: ساخت پایگاه داده و فعال سازی DB Migration

حال ما باید EF CodeFirst Migration را برای آپدیت کردن دیتابیس، بجای دوباره ساختن آن به ازای هر تغییری، فعال نماییم. برای این کار بایستی در قسمت NuGet Package Manager Console عبارت زیر را وارد نماییم:

```
enable-migrations
add-migration InitialCreate
```

اگر تا به اینجای کار تمامی مراحل به درستی صورت گرفته باشند، پوشه‌ی Migration با موفقیت ساخته می‌شود. همچنین پایگاه داده متشکل از جداول مورد نیاز برای سیستم Identity نیز ایجاد می‌شود. برای اطلاعات بیشتر می‌توانید مقالات مرتبط با Code First را مطالعه نمایید. ساختار جداول ما باید به صورت زیر باشد:

AspNetIdentity
Database Diagrams
Tables
System Tables
FileTables
dbo.__MigrationHistory
dbo.AspNetRoles
dbo.AspNetUserClaims
dbo.AspNetUserLogins
dbo.AspNetUserRoles
dbo.AspNetUsers
Columns
Id (PK, nvarchar(128), not null)
FirstName (nvarchar(100), not null)
LastName (nvarchar(100), not null)
Level (tinyint, not null)
JoinDate (datetime, not null)
Email (nvarchar(256), null)
EmailConfirmed (bit, not null)
PasswordHash (nvarchar(max), null)
SecurityStamp (nvarchar(max), null)
PhoneNumber (nvarchar(max), null)
PhoneNumberConfirmed (bit, not null)
TwoFactorEnabled (bit, not null)
LockoutEndDateUtc (datetime, null)
LockoutEnabled (bit, not null)
AccessFailedCount (int, not null)
UserName (nvarchar(256), not null)
Keys

در بخش بعدی، کلاس‌های مربوط به UserManager را ایجاد خواهیم کرد و پس از آن فایل Startup Owin را برای مدیریت کاربران، تشریح می‌کنیم.

عنوان: راهنمای تغییر بخش احراز هویت و اعتبارسنجی کاربران سیستم مدیریت محتوای IRIS به ASP.NET Identity – بخش اول
نویسنده: مهدی سعیدی فر
تاریخ: ۱۳۹۴/۰۷/۱۷ ۱۷:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, MVC, Security, ASP.NET Identity, IrisCMS

سیستم مدیریت محتوای IRIS از سیستم‌های اعتبار سنجی و مدیریت کاربران رایج نظیر ASP.NET Membership و یا ASP.NET Simple Membership استفاده نمی‌کند و از یک سیستم احراز هویت سفارشی شده مبتنی بر FormsAuthentication بهره می‌برد. زمانیکه در حال نوشتن پروژه‌ی IRIS بودم هنوز ASP.NET Identity معرفی نشده بود و به دلیل مشکلاتی که سیستم‌های قدیمی ذکر شده داشت، یک سیستم اعتبار سنجی کاربران سفارشی شده را در پروژه پیاده سازی کردم.
برای اینکه با معایب سیستم‌های مدیریت کاربران پیشین و مزایای ASP.NET Identity آشنا شوید، مقاله زیر می‌تواند شروع خیلی خوبی باشد:
معرفی ASP.NET Identity

به این نکته نیز اشاره کنم که هنوز هم می‌توان از FormsAuthentication مبتنی بر OWIN استفاده کرد؛ ولی چیزی که برای من بیشتر اهمیت دارد خود سیستم Identity هست، نه نحوه‌ی ورود و خروج به سایت و تولید کوکی اعتبارسنجی.
باید اعتراف کرد که سیستم جدید مدیریت کاربران مایکروسافت خیلی خوب طراحی شده است و اشکالات سیستم‌های پیشین خود را ندارد. به راحتی می‌توان آن را توسعه داد و یا قسمتی از آن را تغییر داد و به جرات می‌توان گفت که پایه و اساس هر سیستم اعتبار سنجی و مدیریت کاربری را که در نظر داشته باشید، به خوبی پیاده سازی کرده است.
در ادامه قصد دارم، چگونگی مهاجرت از سیستم فعلی به سیستم Identity را بدون از دست دادن اطلاعات فعلی شرح دهم.

رفع باگ ثبت کاربرهای تکراری نسخه‌ی کنونی

قبل از این که سراغ پیاده سازی Identity برویم، ابتدا باید یک باگ مهم را در نسخه‌ی قبلی، برطرف نماییم. نسخه‌ی کنونی مدیریت کاربران اجازه‌ی ثبت کاربر با ایمیل و یا نام کاربری تکراری را نمی‌دهد. جلوگیری از ثبت نام کاربر جدید با ایمیل یا نام کاربری تکراری از طریق کدهای زیر صورت گرفته است؛ اما در عمل، همیشه هم درست کار نمی‌کند.

```
public AddUserStatus Add(User user)
        {
            if (ExistsByEmail(user.Email))
                return AddUserStatus.EmailExist;
            if (ExistsByUserName(user.UserName))
                return AddUserStatus.UserNameExist;

            _users.Add(user);
            return AddUserStatus.AddingUserSuccessfully;
        }
```

شاید الان با خود بگویید که چرا برای فیلدهای UserName و Email ایندکس منحصر به فرد تعریف نشده است؟ دلیلش این بوده که در زمان نگارش پروژه، Entity Framework پشتیبانی پیش فرضی از تعریف ایندکس نداشت و نوشتن همین شرطها کافی به نظر می‌رسید. باز هم ممکن است بگویید که مسائل همزمانی چگونه مدیریت شده است و اگر دو کاربر مختلف در یک لحظه، نام کاربری یکسانی را انتخاب کنند، سیستم چگونه از ثبت دو کاربر مختلف با نام کاربری یکسان ممانعت می‌کند؟
جواب این است که ممانعتی نمی‌کند و دو کاربر با نام‌های کاربری یکسان ثبت می‌شوند؛ اما من برای وبسایت خودم که تعداد کاربرانش محدود بود این سناریو را محتمل نمی‌دانستم و کد خاصی برای جلوگیری از این اتفاق پیاده سازی نکرده بودم.
با این حال، در حال حاضر نزدیک به 20 کاربر تکراری در دیتابیسی که این سیستم استفاده می‌کند ثبت شده است. اما واقعا آیا دو کاربر مختلف اطلاعات یکسانی وارد کرده‌اند؟
دلیل رخ دادن این اتفاق این است که کاربری که در حال ثبت نام در سایت است، وقتی که بر روی دکمه‌ی ثبت نام کلیک می‌کند و اطلاعات به سرور ارسال می‌شوند، در سمت سرور بعد از رد شدن از شرطهای تکراری نبودن UserName و Email، قبل از رسیدن به متد SaveChanges برای ذخیره شدن اطلاعات کاربر جدید در دیتابیس، وقفه ای در ترد این درخواست به وجود می‌آید. کاربر که احساس می‌کند اتفاقی رخ نداده است، دوباره بر روی دکمه‌ی ثبت نام کلیک می‌کند و همان اطلاعات قبلی به سرور ارسال می‌شود و این درخواست نیز دوباره شرطهای تکراری نبودن اطلاعات را با موفقیت رد می‌کند(چون هنوز SaveChanges درخواست اول

فراخوانی نشده است) و این بار SaveChanges درخواست دوم با موفقیت فراخوانی می‌شود و کاربر ثبت می‌شود. در نهایت هم ترد درخواست اول به ادامه‌ی کار خود می‌پردازد و SaveChanges درخواست اول نیز فراخوانی می‌شود و خیلی راحت دو کاربر با اطلاعات یکسان ثبت می‌شود. این سناریو را در ویژوال استادیو با قرار دادن یک break point قبل از فراخوانی متد SaveChanges می‌توانید شبیه سازی کنید.

احتمالا این سناریو با مباحث همزمانی در سیستم عامل و context switchهای بین تردها مرتبط است و این context switchها بین درخواست‌ها و atomic نبودن روند چک کردن اطلاعات و ثبت آن ها، سبب بروز چنین مشکلی میشود.

برای رفع این مشکل می‌توان از غیر فعال کردن یک دکمه در حین انجام پردازش‌های سمت سرور استفاده کرد تا کاربر بی حوصله، نتواند چندین بار بر روی یک دکمه کلیک کند و یا راه حل اصولی‌تر این است که ایندکس منحصر به فرد برای فیلدهای مورد نظر تعریف کنیم.

به طور پیش فرض در ASP.NET Identity برای فیلدهای UserName و Email ایندکس منحصر به فرد تعریف شده است. اما مشکل این است که به دلیل وجود کاربرانی با Email و UserName تکراری در دیتابیس کنونی، امکان تعریف Index منحصر به فرد وجود ندارد و پیش از انجام هر کاری باید این ناهنجاری را در دیتابیس برطرف نماییم.

به شخصه معمولا برای انجام کارهایی از این دست، یک کنترلر در برنامه خود تعریف می‌کنم و در آنجا کارهای لازم را انجام می‌دهم.

در اینجا من برای حذف کاربران با اطلاعات تکراری، یک کنترلر به نام Migration و اکشن متدی به نام RemoveDuplicateUsers تدارک دیدم.

```
using System.Linq;
using System.Web.Mvc;
using Iris.Datalayer.Context;

namespace Iris.Web.Controllers
{
    public class MigrationController : Controller
    {
        public ActionResult RemoveDuplicateUsers()
        {
            var db = new IrisDbContext();

            var lstDuplicateUserGroup = db.Users
                                            .GroupBy(u => u.UserName)
                                            .Where(g => g.Count() > 1)
                                            .ToList();

            foreach (var duplicateUserGroup in lstDuplicateUserGroup)
            {
                foreach (var user in duplicateUserGroup.Skip(1).Where(user => user.UserMetaData != 
null))
                {
                    db.UserMetaDatas.Remove(user.UserMetaData);
                }

                db.Users.RemoveRange(duplicateUserGroup.Skip(1));
            }

            db.SaveChanges();

            return new EmptyResult();
        }
    }
}
```

در اینجا کاربران بر اساس نام کاربری گروه بندی می‌شوند و گروه‌هایی که بیش از یک عضو داشته باشند، یعنی کاربران آن گروه دارای نام کاربری یکسان هستند و به غیر از کاربر اول گروه، بقیه باید حذف شوند. البته این را متذکر شوم که منطق وبسایت من به این شکل بوده است و اگر منطق کدهای شما فرق می‌کند، مطابق با منطق خودتان این کدها را تغییر دهید.

تذکر: اینجا شاید بگویید که چرا cascade delete برای UserMetaData فعال نیست و بخواهید که آن را اصلاح کنید. پیشنهاد من این است که اکنون از هدف اصلی منحرف نشوید و تمام تمرکز خود را بر روی انتقال به ASP Identity با حداقل تغییرات بگذارید. این گونه نباشد که در اواسط کار با خود بگویید که بد نیست حالا فلان کتابخانه را آپدیت کنم یا این تغییر را بدهم بهتر می‌شود. سعی کنید با حداقل تغییرات رو به جلو حرکت کنید؛ آپدیت کردن کتابخانه‌ها را هم بعدا می‌شود انجام داد.

مقایسه ساختار جداول دیتابیس کاربران IRIS با ASP.NET Identity

ساختار جداول ASP.NET Identity به شکل زیر است:

ساختار جداول سیستم کنونی هم بدین شکل است:

Users
Id
UserName
Password
Email
IP
IsBaned
CreatedDate
BanedDate
LastLoginDate
LastPasswordChange
Role_Id
LastActivity
Roles
Id
Name
Description

همان طور که مشخص است در هر دو سیستم، بین ساختار جداول و رابطه‌ی بین آن‌ها شباهت‌ها و تفاوت هایی وجود دارد. سیستم Identity دو جدول بیشتر از IRIS دارد و برای جداولی که در سیستم کنونی وجود ندارند نیاز به انجام کاری نیست و به هنگام پیاده سازی Identity، این جداول به صورت خودکار به دیتابیس اضافه خواهند شد.

دو جدول مشترک در این دو سیستم، جداول Users و Roles هستندکه نحوه‌ی ارتباطشان با یکدیگر متفاوت است. در Iris بین User و Role رابطه‌ی یک به چند وجود دارد ولی در Identity، رابطه‌ی بین این دو جدول چند به چند است و جدول واسط بین آن‌ها نیز UserRoles نام دارد.

از آن جایی که من قصد دارم در سیستم جدید هم رابطه‌ی بین کاربر و نقش چند به چند باشد، به پیش فرض‌های Identity کاری ندارم. به رابطه‌ی کنونی یک به چند کاربر و نقشش نیز دست نمی‌گذارم تا در انتها با یک کوئری از دیتابیس، اطلاعات نقش‌های کاربران را به جدول جدیدش منتقل کنم.

جدولی که در هر دو سیستم مشترک است و هسته‌ی آن‌ها را تشکیل می‌دهد، جدول Users است. اگر دقت کنید می‌بینید که این جدول در هر دو سیستم، دارای یک سری فیلد مشترک است که دقیقا هم نام هستند مثل Id، UserName و Email؛ پس این فیلدها از نظر کاربرد در هر دو سیستم یکسان هستند و مشکلی ایجاد نمی‌کنند.

یک سری فیلد هم در جدول User در سیستم IRIS هست که در Identity نیست و بلعکس. با این فیلدها نیز کاری نداریم چون در هر دو سیستم کار مخصوص به خود را انجام می‌دهند و تداخلی در کار یکدیگر ایجاد نمی‌کنند.

اما فیلدی که برای ذخیره سازی پسورد در هر دو سیستم استفاده می‌شود دارای نام‌های متفاوتی است. در Iris این فیلد Password نام دارد و در Identity نامش PasswordHash است.

برای اینکه در سیستم کنونی، نام فیلد Password جدول User را به PasswordHash تغییر دهیم قدم‌های زیر را بر می‌داریم:

وارد پروژه‌ی DomainClasses شده و کلاس User را باز کنید. سپس نام خاصیت Password را به PasswordHash تغییر دهید. پس از این تغییر بلافاصله یک گزینه زیر آن نمایان می‌شود که می‌خواهد در تمام جاهایی که از این نام استفاده شده است را به نام جدید تغییر دهد؛ آن را انتخاب کرده تا همه جا Password به PasswordHash تغییر کند.

برای این که این تغییر نام بر روی دیتابیس نیز اعمال شود باید از Migration استفاده کرد. در اینجا من از مهاجرت دستی که بر اساس کد هست استفاده می‌کنم تا هم بتوانم کدهای مهاجرت را پیش از اعمال بررسی و هم تاریخچه‌ای از تغییرات را ثبت کنم.

برای این کار، Package Manager Console را باز کرده و از نوار بالایی آن، پروژه پیش فرض را بر روی DataLayer قرار دهید. سپس در کنسول، دستور زیر را وارد کنید:

```
Add-Migration Rename_PasswordToPasswordHash_User
```

اگر وارد پوشه Migrations پروژه DataLayer خود شوید، باید کلاسی با نامی شبیه به 201510090808056_Rename_PasswordToPasswordHash_User ببینید. اگر آن را باز کنید کدهای زیر را خواهید دید:

```
public partial class Rename_PasswordToPasswordHash_User : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Users", "PasswordHash", c => c.String(nullable: false, maxLength: 200));
            DropColumn("dbo.Users", "Password");
        }

        public override void Down()
```

```
        {
            AddColumn("dbo.Users", "Password", c => c.String(nullable: false, maxLength: 200));
            DropColumn("dbo.Users", "PasswordHash");
        }
    }
```

بدیهی هست که این کدها عمل حذف ستون Password را انجام میدهند که سبب از دست رفتن اطلاعات می‌شود. کدهای فوق را به شکل زیر ویرایش کنید تا تنها سبب تغییر نام ستون Password به PasswordHash شود.

```
public partial class Rename_PasswordToPasswordHash_User : DbMigration
    {
        public override void Up()
        {
            RenameColumn("dbo.Users", "Password", "PasswordHash");
        }

        public override void Down()
        {
            RenameColumn("dbo.Users", "PasswordHash", "Password");
        }
    }
```

سپس باز در کنسول دستور Update-Database را وارد کنید تا تغییرات بر روی دیتابیس اعمال شود.

دلیل اینکه این قسمت را مفصل بیان کردم این بود که می‌خواستم در مهاجرت از سیستم اعتبارسنجی خودتان به ASP.NET Identity دید بهتری داشته باشید.

تا به این جای کار فقط پایگاه داده سیستم کنونی را برای مهاجرت آماده کردیم و هنوز ASP.NET Identity را وارد پروژه نکردیم. در بخش‌های بعدی Identity را نصب کرده و تغییرات لازم را هم انجام می‌دهیم.

عنوان: راهنمای تغییر بخش احراز هویت و اعتبارسنجی کاربران سیستم مدیریت محتوای IRIS به ASP.NET Identity – بخش دوم
نویسنده: مهدی سعیدی فر
تاریخ: ۱۳۹۴/۰۷/۲۷ ۱۰:۲۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, MVC, Security, ASP.NET Identity, IrisCMS

در بخش اول ، کارهایی که انجام دادیم به طور خلاصه عبارت بودند از:
1- حذف کاربرانی که نام کاربری و ایمیل تکراری داشتند
2- تغییر نام فیلد Password به PasswordHash در جدول User
سیستم مدیریت محتوای IRIS، برای استفاده از Entity Framework، از الگوی واحد کار (Unit Of Work) و تزریق وابستگی استفاده کرده است و اگر با نحوه‌ی پیاده سازی این الگوها آشنا نیستید، خواندن مقاله EF Code First #12 را به شما توصیه می‌کنم.
برای استفاده از ASP.NET Identity نیز باید از الگوی واحد کار استفاده کرد و برای این کار، ما از مقاله اعمال تزریق وابستگی‌ها به مثال رسمی ASP.NET Identity استفاده خواهیم کرد. **نکته مهم:** در ادامه اساس کار ما بر پایه‌ی مقاله اعمال تزریق وابستگی‌ها به مثال رسمی ASP.NET Identity است و چیزی که بیشتر برای ما اهمیت دارد کدهای نهایی آن هست؛ پس حتما به مخزن کد آن مراجعه کرده و کدهای آن را دریافت کنید. **تغییر نام کلاس User به ApplicationUser**
اگر به کدهای مثال رسمی ASP.NET Identity نگاهی بیندازید، می‌بینید که کلاس مربوط به جدول کاربران ApplicationUser نام دارد، ولی در سیستم IRIS نام آن User است. بهتر است که ما هم نام کلاس خود را از User به ApplicationUser تغییر دهیم چرا که مزایای زیر را به دنبال دارد:
1- به راحتی می‌توان کدهای مورد نیاز را از مثال Identity کپی کرد.
2- در سیستم Iris، بین کلاس User متعلق به پروژه خودمان و User مربوط به HttpContext تداخل رخ می‌داد که با تغییر نام کلاس User دیگر این مشکل را نخواهیم داشت.
برای این کار وارد پروژه Iris.DomainClasses شده و نام کلاس User را به ApplicationUser تغییر دهید. دقت کنید که این تغییر نام را از طریق Solution Explorer انجام دهید و نه از طریق کدهای آن. پس از این تغییر ویژوال استودیو می‌پرسد که آیا نام این کلاس را هم در کل پروژه تغییر دهد که شما آن را تایید کنید.
برای آن که نام جدول Users در دیتابیس تغییری نکند، وارد پوشه‌ی Entity Configuration شده و کلاس UserConfig را گشوده و در سازنده‌ی آن کد زیر را اضافه کنید:

```
ToTable("Users");
```

نصب ASP.NET Identity

برای نصب ASP.NET Identity دستور زیر را در کنسول Nuget وارد کنید:

```
Get-Project Iris.DomainClasses, Iris.Datalayer, Iris.Servicelayer, Iris.Web | Install-Package
Microsoft.AspNet.Identity.EntityFramework
```

از پروژه AspNetIdentityDependencyInjectionSample.DomainClasses کلاس‌های CustomUserRole، CustomUserLogin، CustomUserClaim و CustomRole را به پروژه Iris.DomainClasses منتقل کنید. تنها تغییری که در این کلاس‌ها باید انجام دهید، اصلاح namespace آنهاست.
همچنین بهتر است که به کلاس CustomRole، یک property به نام Description اضافه کنید تا توضیحات فارسی نقش مورد نظر را هم بتوان ذخیره کرد:

```
public class CustomRole : IdentityRole<int, CustomUserRole>
    {
        public CustomRole() { }
        public CustomRole(string name) { Name = name; }

        public string Description { get; set; }

    }
```

نکته: پیشنهاد می‌کنم که اگر می‌خواهید مثلا نام CustomRole را به IrisRole تغییر دهید، این کار را از طریق find and replace انجام ندهید. با همین نام‌های پیش فرض کار را تکمیل کنید و سپس از طریق خود ویژوال استودیو نام کلاس را تغییر دهید تا ویژوال استودیو به نحو بهتری این نام‌ها را در سرتاسر پروژه تغییر دهد.

سپس کلاس ApplicationUser پروژه IRIS را باز کرده و تعریف آن را به شکل زیر تغییر دهید:

```
public class ApplicationUser : IdentityUser<int, CustomUserLogin, CustomUserRole, CustomUserClaim>
```

اکنون می‌توانید propertyهای Id، UserName، PasswordHash و Email را حذف کنید؛ چرا که در کلاس پایه IdentityUser تعریف شده اند.

تغییرات DataLayer

وارد Iris.DataLayer شده و کلاس IrisDbContext را به شکل زیر ویرایش کنید:

```
public class IrisDbContext : IdentityDbContext<ApplicationUser, CustomRole, int, CustomUserLogin,
CustomUserRole, CustomUserClaim>,
        IUnitOfWork
```

اکنون می‌توانید property زیر را نیز حذف کنید چرا که در کلاس پایه تعریف شده است:

```
public DbSet<ApplicationUser> Users { get; set; }
```

نکته مهم: حتما برای کلاس IrisDbContext سازنده ای تعریف کنید که صراحتا نام رشته اتصالی را ذکر کرده باشد، اگر این کار را انجام ندهید با خطاهای عجیب غریبی روبرو می‌شوید.

```
public IrisDbContext()
            : base("IrisDbContext")
        {
        }
```

همچنین درون متد OnModelCreating کدهای زیر را پس از فراخوانی متد base.OnModelCreating(modelBuilder) جهت تعیین نام جداول دیتابیس بنویسید:

```
modelBuilder.Entity<CustomRole>().ToTable("AspRoles");
            modelBuilder.Entity<CustomUserClaim>().ToTable("UserClaims");
            modelBuilder.Entity<CustomUserRole>().ToTable("UserRoles");
            modelBuilder.Entity<CustomUserLogin>().ToTable("UserLogins");
```

از این جهت نام جدول CustomRole را در دیتابیس AspRoles انتخاب کردم تا با نام جدول Roles نقش‌های کنونی سیستم Iris تداخلی پیش نیاید. اکنون دستور زیر را در کنسول Nuget وارد کنید تا کدهای مورد نیاز برای مهاجرت تولید شوند:

```
Add-Migration UpdateDatabaseToAspIdentity
```

```
public partial class UpdateDatabaseToAspIdentity : DbMigration
{
        public override void Up()
        {
            CreateTable(
                "dbo.UserClaims",
                c => new
                    {
                        Id = c.Int(nullable: false, identity: true),
                        UserId = c.Int(nullable: false),
                        ClaimType = c.String(),
                        ClaimValue = c.String(),
                        ApplicationUser_Id = c.Int(),
                    })
                .PrimaryKey(t => t.Id)
                .ForeignKey("dbo.Users", t => t.ApplicationUser_Id)
```

```
                .Index(t => t.ApplicationUser_Id);

            CreateTable(
                "dbo.UserLogins",
                c => new
                    {
                        LoginProvider = c.String(nullable: false, maxLength: 128),
                        ProviderKey = c.String(nullable: false, maxLength: 128),
                        UserId = c.Int(nullable: false),
                        ApplicationUser_Id = c.Int(),
                    })
                .PrimaryKey(t => new { t.LoginProvider, t.ProviderKey, t.UserId })
                .ForeignKey("dbo.Users", t => t.ApplicationUser_Id)
                .Index(t => t.ApplicationUser_Id);

            CreateTable(
                "dbo.UserRoles",
                c => new
                    {
                        UserId = c.Int(nullable: false),
                        RoleId = c.Int(nullable: false),
                        ApplicationUser_Id = c.Int(),
                    })
                .PrimaryKey(t => new { t.UserId, t.RoleId })
                .ForeignKey("dbo.Users", t => t.ApplicationUser_Id)
                .ForeignKey("dbo.AspRoles", t => t.RoleId, cascadeDelete: true)
                .Index(t => t.RoleId)
                .Index(t => t.ApplicationUser_Id);

            CreateTable(
                "dbo.AspRoles",
                c => new
                    {
                        Id = c.Int(nullable: false, identity: true),
                        Description = c.String(),
                        Name = c.String(nullable: false, maxLength: 256),
                    })
                .PrimaryKey(t => t.Id)
                .Index(t => t.Name, unique: true, name: "RoleNameIndex");

            AddColumn("dbo.Users", "EmailConfirmed", c => c.Boolean(nullable: false));
            AddColumn("dbo.Users", "SecurityStamp", c => c.String());
            AddColumn("dbo.Users", "PhoneNumber", c => c.String());
            AddColumn("dbo.Users", "PhoneNumberConfirmed", c => c.Boolean(nullable: false));
            AddColumn("dbo.Users", "TwoFactorEnabled", c => c.Boolean(nullable: false));
            AddColumn("dbo.Users", "LockoutEndDateUtc", c => c.DateTime());
            AddColumn("dbo.Users", "LockoutEnabled", c => c.Boolean(nullable: false));
            AddColumn("dbo.Users", "AccessFailedCount", c => c.Int(nullable: false));
        }

        public override void Down()
        {
            DropForeignKey("dbo.UserRoles", "RoleId", "dbo.AspRoles");
            DropForeignKey("dbo.UserRoles", "ApplicationUser_Id", "dbo.Users");
            DropForeignKey("dbo.UserLogins", "ApplicationUser_Id", "dbo.Users");
            DropForeignKey("dbo.UserClaims", "ApplicationUser_Id", "dbo.Users");
            DropIndex("dbo.AspRoles", "RoleNameIndex");
            DropIndex("dbo.UserRoles", new[] { "ApplicationUser_Id" });
            DropIndex("dbo.UserRoles", new[] { "RoleId" });
            DropIndex("dbo.UserLogins", new[] { "ApplicationUser_Id" });
            DropIndex("dbo.UserClaims", new[] { "ApplicationUser_Id" });
            DropColumn("dbo.Users", "AccessFailedCount");
            DropColumn("dbo.Users", "LockoutEnabled");
            DropColumn("dbo.Users", "LockoutEndDateUtc");
            DropColumn("dbo.Users", "TwoFactorEnabled");
            DropColumn("dbo.Users", "PhoneNumberConfirmed");
            DropColumn("dbo.Users", "PhoneNumber");
            DropColumn("dbo.Users", "SecurityStamp");
            DropColumn("dbo.Users", "EmailConfirmed");
            DropTable("dbo.AspRoles");
            DropTable("dbo.UserRoles");
            DropTable("dbo.UserLogins");
            DropTable("dbo.UserClaims");
        }
}
```

بهتر است که در کدهای تولیدی فوق، اندکی متد Up را با کد زیر تغییر دهید:

```
AddColumn("dbo.Users", "EmailConfirmed", c => c.Boolean(nullable: false, defaultValue:true));
```

چون در سیستم جدید احتیاج به تایید ایمیل به هنگام ثبت نام است، بهتر است که ایمیل‌های قبلی موجود در سیستم نیز به طور پیش فرض تایید شده باشند.
در نهایت برای اعمال تغییرات بر روی دیتابیس دستور زیر را در کنسول Nuget وارد کنید:

```
Update-Database
```

تغییرات ServiceLayer

ابتدا دستور زیر را در کنسول Nuget وارد کنید:

```
Get-Project Iris.Servicelayer, Iris.Web | Install-Package Microsoft.AspNet.Identity.Owin
```

سپس از فولدر Contracts پروژه AspNetIdentityDependencyInjectionSample.ServiceLayer فایل‌های IApplicationRoleManager، IApplicationSignInManager، IApplicationUserManager، ICustomRoleStore و ICustomUserStore را در فولدر Interfaces پروژه Iris.ServiceLayer کپی کنید. تنها کاری هم که نیاز هست انجام بدهید اصلاح namespace هاست.
باز از پروژه AspNetIdentityDependencyInjectionSample.ServiceLayer کلاس‌های ApplicationRoleManager، ApplicationSignInManager، ApplicationUserManager، CustomRoleStore، CustomUserStore، EmailService و SmsService را به پوشه EFServcies پروژه‌ی Iris.ServiceLayer کپی کنید.
نکته: پیشنهاد می‌کنم که EmailService را به IdentityEmailService تغییر نام دهید چرا که در حال حاضر سیستم Iris دارای کلاسی به نامی EmailService هست.

تنظیمات StructureMap برای تزریق وابستگی ها

پروژه Iris.Web را باز کرده، به فولدر DependencyResolution بروید و به کلاس IoC کدهای زیر را اضافه کنید:

```
x.For<IIdentity>().Use(() => (HttpContext.Current != null && HttpContext.Current.User != null) ?
HttpContext.Current.User.Identity : null);

                x.For<IUnitOfWork>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<IrisDbContext>();

                x.For<IrisDbContext>().HybridHttpOrThreadLocalScoped()
                   .Use(context => (IrisDbContext)context.GetInstance<IUnitOfWork>());
                x.For<DbContext>().HybridHttpOrThreadLocalScoped()
                   .Use(context => (IrisDbContext)context.GetInstance<IUnitOfWork>());

                x.For<IUserStore<ApplicationUser, int>>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<CustomUserStore>();

                x.For<IRoleStore<CustomRole, int>>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<RoleStore<CustomRole, int, CustomUserRole>>();

                x.For<IAuthenticationManager>()
                      .Use(() => HttpContext.Current.GetOwinContext().Authentication);

                x.For<IApplicationSignInManager>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use<ApplicationSignInManager>();

                x.For<IApplicationRoleManager>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use<ApplicationRoleManager>();

                // map same interface to different concrete classes
                x.For<IIdentityMessageService>().Use<SmsService>();
                x.For<IIdentityMessageService>().Use<IdentityEmailService>();

                x.For<IApplicationUserManager>().HybridHttpOrThreadLocalScoped()
                   .Use<ApplicationUserManager>()
                   .Ctor<IIdentityMessageService>("smsService").Is<SmsService>()
```

```
                    .Ctor<IIdentityMessageService>("emailService").Is<IdentityEmailService>()
                    .Setter<IIdentityMessageService>(userManager =>
userManager.SmsService).Is<SmsService>()
                    .Setter<IIdentityMessageService>(userManager =>
userManager.EmailService).Is<IdentityEmailService>();

                x.For<ApplicationUserManager>().HybridHttpOrThreadLocalScoped()
                    .Use(context =>
(ApplicationUserManager)context.GetInstance<IApplicationUserManager>());

                x.For<ICustomRoleStore>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use<CustomRoleStore>();

                x.For<ICustomUserStore>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use<CustomUserStore>();
```

اگر HttpContext.Current.GetOwinContext() شناسایی نمی‌شود دلیلش این است که متد GetOwinContext یک متد الحاقی است که برای استفاده از آن باید پکیج نیوگت زیر را نصب کنید:

```
Install-Package Microsoft.Owin.Host.SystemWeb
```

تغییرات Iris.Web

در ریشه پروژه‌ی Iris.Web یک کلاس به نام Startup بسازید و کدهای زیر را در آن بنویسید:

```
using System;
using Iris.Servicelayer.Interfaces;
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.DataProtection;
using Owin;
using StructureMap;

namespace Iris.Web
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            configureAuth(app);
        }

        private static void configureAuth(IAppBuilder app)
        {
            ObjectFactory.Container.Configure(config =>
            {
                config.For<IDataProtectionProvider>()
                      .HybridHttpOrThreadLocalScoped()
                      .Use(() => app.GetDataProtectionProvider());
            });

            //ObjectFactory.Container.GetInstance<IApplicationUserManager>().SeedDatabase();

            // 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 =
ObjectFactory.Container.GetInstance<IApplicationUserManager>().OnValidateIdentity()
                }
            });
            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);

            app.CreatePerOwinContext(
               () => ObjectFactory.Container.GetInstance<IApplicationUserManager>());

            // 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(
            //    clientId: "",
            //    clientSecret: "");

        }
    }
}
```

تا به این جای کار اگر پروژه را اجرا کنید نباید هیچ مشکلی مشاهده کنید. در بخش بعدی کدهای مربوط به کنترلرهای ورود، ثبت نام، فراموشی کلمه عبور و ... را با سیستم Identity پیاده سازی می‌کنیم.