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.zip

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 SmObjectFactory مطلب فوق استفاده می‌کنید، Container آن با ObjectFactory یکی نیست یا به عبارتی ObjectFactory اطلاعی در مورد تنظیمات کلاس سفارشی SmObjectFactory 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.