 nozerowu merge md file    56a7562 28 minutes ago

1 contributor

## 1.1 ABP总体介绍 - 入门介绍

ABP是“ASP.NET Boilerplate Project (ASP.NET样板项目)”的简称。

ASP.NET Boilerplate是一个用最佳实践和流行技术开发现代WEB应用程序的新起点，它旨在成为一个通用的WEB应用程序基础框架和项目模板。

ASP.NET Boilerplate 基于DDD的经典分层架构思想，实现了众多DDD的概念（但没有实现所有DDD的概念）。

- ABP的官方网站：<http://www.aspnetboilerplate.com>
- ABP在Github上的开源项目：<https://github.com/aspnetboilerplate>

ABP框架于2014年5月4日首次在Github开源，截止到2015年5月25日，总共进行了1271次代码提交，49次版本发布，现在的版本号是0.6.1.1。

### 1.1.1 ABP采用了以下技术

- 服务器端：
- ASP.NET MVC 5、Web API 2、C# 5.0
  - DDD领域驱动设计（Entities、Repositories、Domain Services、Domain Events、Application Services、DTOs等）
  - Castle windsor（依赖注入容器）
  - Entity Framework 6 \ NHibernate，数据迁移
  - Log4Net（日志记录）
  - AutoMapper（实现Dto类与实体类的双向自动转换）

- 客户端：
- Bootstrap
  - Less
  - AngularJs
  - jQuery
  - Modernizr
  - 其他JS库: jQuery.validate、jQuery.form、jQuery.blockUI、json2

### 1.1.2 ABP框架已实现了以下特性

- 多语言/本地化支持
- 多租户支持（每个租户的数据自动隔离，业务模块开发者不需要在保存和查询数时写相应代码）
- 软删除支持（继承相应的基类或实现相应接口，会自动实现软删除）
- 统一的异常处理（应用层几乎不需要处理自己写异常处理代码）
- 数据有效性验证（Asp.NET MVC只能做到Action方法的参数验证，ABP实现了Application层方法的参数有效性验证）
- 日志记录（自动记录程序异常）
- 模块化开发（每个模块有独立的EF DbContext，可单独指定数据库）

- Repository仓储模式（已实现了Entity Framework、NHibernate、MongoDB、内存数据库）
- Unit Of Work工作单元模式（为应用层和仓储层的方法自动实现数据库事务）
- EventBus实现领域事件(Domain Events)
- DLL嵌入资源管理
- 通过Application Services自动创建Web Api层（不需要写ApiController层了）
- 自动创建Javascript 的代理层来更方便使用Web Api
- 封装一些Javascript 函数，更方便地使用ajax、消息框、通知组件、忙状态的遮罩层

“Zero”的模块，实现了以下功能：

- 身份验证与授权管理（通过ASP.NET Identity实现的）
- 用户&角色管理
- 系统设置存取管理（系统级、租户级、用户级，作用范围自动管理）
- 审计日志（自动记录每一次接口的调用者和参数）

在其他项目中的很多优秀设计，在ABP项目中也已存在，而且可能实现得更好。ABP框架的代码，都通过xUnit进行了单元测试。作者一直在用ABP框架开发他们的实际项目，从Github和他官方论坛上的信息可以看到，有很多国外的开发者在将ABP用作生产项目的基础框架。如果需要直接使用ABP组件，可以通过Nuget安装（在VS的Nuget包管理界面搜索ABP）。

为了更好地将ABP适用于自己的项目，我对ABP的源码做了一些修改后使用的，没有直接使用ABP组件。

我希望更多国内的架构师能关注ABP这个项目，也许这其中有帮助到你的地方，也许有你的参与，这个项目可以发展得更好。

今天只是作了一个大概介绍，希望有更多的朋友能去阅读源代码，然后参与讨论。

### 1.1.3 ABP适用的场景

中小规模WEB应用开发，可直接使用ABP框架。较大型项目可以在ABP框架的源码基础上进行扩展，以实现分布式架构。

注：处理高并发并不是ABP的强项。需要非常高并发的DDD框架，建议去研究netfocus的ENode。

## 1.2 ABP总体介绍 - 层架构体系

### 1.2.1 前言

为了减少复杂性和提高代码的可重用性，采用分层架构是一种被广泛接受的技术。为了实现分层的体系结构，ABP遵循DDD（领域驱动设计）的原则，将工程分为四个层：

- 展现层（**Presentation**）：提供一个用户界面，实现用户交互操作。
- 应用层（**Application**）：进行展现层与领域层之间的协调，协调业务对象来执行特定的应用程序的任务。它不包含业务逻辑。
- 领域层（**Domain**）：包括业务对象和业务规则，这是应用程序的核心层。
- 基础设施层（**Infrastructure**）：提供通用技术来支持更高的层。例如基础设施层的仓储(Repository)可通过ORM来实现数据库交互。

根据实际需要，可能会有额外添加的层。例如：

- 分布式服务层（**Distributed Service**）：用于公开应用程序接口供远程客户端调用。比如通过ASP.NET Web API或WCF来实现。这些都是常见的以领域为中心的分层体系结构。不同的项目在实现上可能会有细微的差别。

### 1.2.2 ABP的体系结构

Layers

Presentation	View Models (Javascript)	Views (HTML/CSS)	Localization				
Web	Web API Controllers	MVC Controllers					
Application	Application Services	DTOs	DTO Mappers				
Domain	Entities	Value Objects	Repositories	Business Classes	Domain Services	Unit of Work	Domain Events
Infrastructure	ORM	DB Migrations					

Others

Server Side	Dependency Injection	Logging
-------------	----------------------	---------

一个简单的解决方案，大致包含5个项目：

```
解决方案'SimpleTaskSystem' (5 个项目)
└─ SimpleTaskSystem.Application
└─ SimpleTaskSystem.Core
└─ SimpleTaskSystem.EntityFramework
└─ SimpleTaskSystem.Web
└─ SimpleTaskSystem.WebApi
```

每一层可以用一个或多个程序集来实现。

1.2.3 领域层

领域层就是业务层，是一个项目的核心，所有业务规则都应该在领域层实现。

实体（Entity）： 实体代表业务领域的数据和操作，在实践中，通过用来映射成数据库表。

仓储（Repository）： 仓储用来操作数据库进行数据存取。仓储接口在领域层定义，而仓储的实现类应该写在基础设施层。

领域服务（Domain service）： 当处理的业务规则跨越两个（及以上）实体时，应该写在领域服务方法里面。

领域事件（Domain Event）： 在领域层有些特定情况发生时可以触发领域事件，并且在相应地方捕获并处理它们。

工作单元（Unit of Work）： 工作单元是一种设计模式，用于维护一个由已经被修改(如增加、删除和更新等)的业务对象组成的列表。它负责协调这些业务对象的持久化工作及并发问题。

1.2.4 应用层

应用层提供一些应用服务（Application Services）方法供展现层调用。一个应用服务方法接收一个DTO(数据传输对象)作为输入参数，使用这个输入参数执行特定的领域层操作，并可根据需要可返回另一个DTO。在展现层到领域层之间，不应该接收或返回实体(Entity)对象，应该进行DTO映射。一个应用服务方法通常被认为是一个工作单元（Unit of Work）。用户输入参数的验证工作也应该在应用层实现。ABP提供了一个基础架构让我们很容易地实现输入参数有效性验证。建议使用一种像AutoMapper这样的工具来进行实体与DTO之间的映射。

1.2.5 基础设施层

当在领域层中为定义了仓储接口，应该在基础设施层中实现这些接口。可以使用ORM工具，例如EntityFramework或NHibernate。ABP的基类已经提供了对这两种ORM工具的支持。数据库迁移也被用于这一层。

1.2.6 WEB与展现层

Web层使用ASP.NET MVC和Web API来实现。可分别用于多页面应用程序(MPA)和单页面应用程序(SPA)。

在SPA中，所有资源被一次加载到客户端浏览器中（或者先只加载核心资源，其他资源懒加载），然后通过AJAX调用服务端WebApi接口获取数据，再根据数据生成HTML代码。不会整个页面刷新。现在已经有很多SPA的JS框架，例如： AngularJs、DurandalJs、BackboneJs、EmberJs。 ABP可以使用任何类似的前端框架，但是ABP提供了一些帮助类，让我们更方便地使用AngularJs和DurandalJs。

在经典的多页面应用（MPA）中，客户端向服务器端发出请求，服务器端代码（ASP.NET MVC控制器）从数据库获得数据，并且使用Razor视图生成HTML。这些被生成后的HTML页面被发送回客户端显示。每显示一个新的页面都会整页刷新。

SPA和MPA涉及到完全不同的体系结构，也有不同的应用场景。一个管理后台适合用SPA，博客就更适合用MPA，因为它更利于被搜索引擎抓取。

SignalR是一种从服务器到客户端发送推送通知的完美工具。它能给用户丰富的实时的体验。已经有很多客户端的Javascript框架或库，JQuery是其中最流行的，并且它有成千上万免费的插件。使用Bootstrap可以让我们更轻松地完成写Html和CSS的工作。

ABP也实现了根据Web API接口自动创建 Javascript的代码函数，来简化JS对Web Api的调用。还有把服务器端的菜单、语言、设置等生成到JS端。（但是在我自己的项目中，我是把这些自动生成功能关闭的，因为必要性不是很大，而这些又会比较影响性能）。

ABP会自动处理服务器端返回的异常，并以友好的界面提示用户。

### 1.2.7 其它

ABP使用Castle Windsor为整个程序框架提供依赖注入的功能。使用Log4Net日志记录组件，提供给其他各层调用以进行日志记录。

## 1.3 ABP总体介绍 - 模块系统

### 1.3.1 ABP模块系统简介

ABP框架提供了创建和组装模块的基础，一个模块能够依赖于另一个模块。在通常情况下，一个程序集就可以看成是一个模块。在ABP框架中，一个模块通过一个类来定义，而这个类要继承自AbpModule。

译者注：如果学习过Orchard的朋友，应该知道module模块的强大。模块的本质就是可重用性，你可以在任意的地方去调用，而且通过实现模块，你写的模块也可以给别人用。.net可以通过反射获取一个程序集中的类以及方法。

Assembly程序集：Assembly是一个用来包含程序的名称，版本号，自我描述，文件关联关系和文件位置等信息的一个集合。最简单的理解就是：一个你自己写的类库生成的dll就可以看做是一个程序集，这个程序集可以包括很多类，类又包括很多方法等。

下面的例子，我们开发一个可以在多个不同应用中被调用MyBlogApplication模块，代码如下：

```
public class MyBlogApplicationModule : AbpModule //定义
{
    public override void Initialize() //初始化
    {
        IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
        //这行代码的写法基本上是不变的。它的作用是把当前程序集的特定类或接口注册到依赖注入容器中。
    }
}
```

ABP框架会扫描所有的程序集，并且发现AbpModule类中所有已经导入的类，如果你已经创建了包含多个程序集的应用，对于ABP，我们的建议是为每一个程序集创建一个Module（模块）。

### 1.3.2 生命期事件

在一个应用中，ABP框架调用了Module模块的一些指定的方法来进行启动和关闭模块的操作。我们可以重载这些方法来完成我们自己的任务。

ABP框架通过依赖关系的顺序来调用这些方法，假如：模块A依赖于模块B,那么模块B要在模块A之前初始化，模块启动的方法顺序如下：

1. PreInitialize-B
2. PreInitialize-A
3. Initialize-B
4. Initialize-A
5. PostInitialize-B
6. PostInitialize-A

下面是具体方法的说明：

## PreInitialize

预初始化：当应用启动后，第一次运行会先调用这个方法。在依赖注入注册之前，你可以在这个方法中指定你需要注入的自定义启动类。举个例子吧：假如你创建了一个自定义的 `TestAbpSession : IAbpSession`，那么请在这个方法中对 `TestAbpSession` 进行注册。

## Initialize

初始化：在这个方法中一般是来进行依赖注入的注册，一般我们通过 `IocManager.RegisterAssemblyByConvention` 这个方法来实现。如果你想实现自定义的依赖注入，那么请参考依赖注入的相关文档。

## PostInitialize

提交初始化：最后一个方法，这个方法用来解析依赖关系。

## Shutdown

关闭：当应用关闭以后，这个方法被调用。

### 1.3.3 模块依赖

Abp框架会自动解析模块之间的依赖关系，但是我们还是建议你通过重载 `GetDependencies` 方法来明确的声明依赖关系。

```
[DependsOn(typeof(MyBlogCoreModule))]/通过注解来定义依赖关系
public class MyBlogApplicationModule : AbpModule
{
    public override void Initialize()
    {
        IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
    }
}
```

例如上面的代码，我们就声明了 `MyBlogApplicationModule` 和 `MyBlogCoreModule` 的依赖关系（通过属性 attribute），`MyBlogApplicationModule` 这个应用模块依赖于 `MyBlogCoreModule` 核心模块，并且，`MyBlogCoreModule` 核心模块会在 `MyBlogApplicationModule` 模块之前进行初始化。

### 1.3.4 插件模块

当模块从启动模块以及其依赖关系进行调查发现的时候，ABP也能够动态的加载其它指定模块。`AbpBootstrapper` 类定义了 `PlugInSources` 属性，我们能够是用该属性添加需要动态加载的模块。插件源可以是任何实现了 `IPlugInSource` 接口的类。`PlugInFolderSource` 类实现了该接口，它可以被用来加载指定文件夹下的程序集。

## ASP.NET Core

ABP的ASP.NET Core模块也可以动态加载模块，你只需要在 `Startup` 类中使用已定义的扩展方法 `AddAbp`，如下所示：

```
services.AddAbp<MyStartupModule>(options =>
{
    options.PlugInSources.Add(new FolderPlugInSource(@"C:\MyPlugIns"));
});
```

我们可以使用扩展方法 `AddFolder` 更方便的实现上述功能：

```
services.AddAbp<MyStartupModule>(options =>
{
    options.PlugInSources.AddFolder(@"C:\MyPlugIns");
});
```

了解更多关于Startup类的信息，请查看 [ASP.NET 文档](#)

## ASP.NET MVC, Web API

对于经典的ASP.NET MVC应用，我们可以在 `global.asax` 重写 `Application_Start` 方法来添加插件文件夹，如下所示：

```
public class MvcApplication : AbpWebApplication<MyStartupModule>
{
    protected override void Application_Start(object sender, EventArgs e)
```

```

    {
        AbpBootstrapper.PlugInSources.AddFolder(@"C:\MyPlugIns");
        //...
        base.Application_Start(sender, e);
    }
}

```

### Additional Assemblies

对于IAssemblyFinder和ITypeFinder的默认实现(这两个接口的实现被ABP用来在应用程序中发现指定的类)仅仅只用来查找模块程序集以及在这些程序集中所使用的类型。我们可以在我们的模块中重写 **GetAdditionalAssemblies** 方法来包含附加程序集。

### 1.3.5 自定义的模块方法

我们自己定义的模块中可能有方法被其他依赖于当前模块的模块调用，下面的例子，假设模块2依赖于模块1，并且想在预初始化的时候调用模块1的方法。这样，就把模块1注入到了模块2，因此，模块2就能调用模块1的方法了。

译者注：ABP的模块系统与Orchard的模块有类似之处，但还是有比较大的差别。Orchard的框架修改了ASP.NET程序集的默认加载方式（模块的DLL没有放在Bin文件夹下，是放在WEB项目根文件夹下面的Modules文件夹下），实现了功能模块的热插拔，而ABP的模块程序集还是放在Bin文件夹下的，没有实现热插拔。

```

public class MyModule1 : AbpModule
{
    public override void Initialize() //初始化模块
    {
        IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly()); //这里，进行依赖注入的注册。
    }

    public void MyModuleMethod1()
    {
        //这里写自定义的方法。
    }
}

[DependsOn(typeof(MyModule1))]
public class MyModule2 : AbpModule
{
    private readonly MyModule1 _myModule1;

    public MyModule2(MyModule1 myModule1)
    {
        _myModule1 = myModule1;
    }

    public override void PreInitialize()
    {
        _myModule1.MyModuleMethod1(); //调用MyModuleMethod1的方法。
    }

    public override void Initialize()
    {
        IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
    }
}
```


#### ## 1.4 ABP总体介绍 - 启动配置


```

在应用启动之前，abp框架提供了模块基本的配置和方法，大家参照下面这个例子就可以了。

>译者注：

在看这一节的内容之前，建议大家先下载module-zero这个例子代码，这个例子就是一个用户和角色的模块，并且使用的实例。配置在每一个应用

#### ### 1.4.1 配置ABP

配置是通过在自己模块的PreInitialize方法中来实现的（对于module的PreInitialize方法，在上一篇中已经向大家做了简单的说明）代码示例

```

```csharp
public class SimpleTaskSystemModule : AbpModule
{
    public override void PreInitialize()
    {
        //在你的应用中添加语言包，这个是英语和作者的土耳其语。
        Configuration.Localization.Languages.Add(new LanguageInfo("en", "English", "famfamfam-flag-england", true));
        Configuration.Localization.Languages.Add(new LanguageInfo("tr", "Türkçe", "famfamfam-flag-tr"));
    }
}

```

```
Configuration.Localization.Sources.Add(
    new XmlLocalizationSource(
        "SimpleTaskSystem",
        HttpContext.Current.Server.MapPath("~/Localization/SimpleTaskSystem")
    )
);

//配置导航和菜单
Configuration.Navigation.Providers.Add<SimpleTaskSystemNavigationProvider>();
}

public override void Initialize()
{
    IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
}
}
```

和orchard类似，abp框架一开始就被设计成模块化的，不同的模块可以通过abp框架来进行配置。举个例子吧，不同的模块都可以添加导航，通过导航添加菜单项到自己定义的主菜单，具体的细节大家可以参照：

- 本地化
- 导航

替换内置服务

**Configuration.ReplaceService** 方法可以被用来重写某个内置的服务。例如，你可以替换掉IAbpSession服务使用你自己扩展的服务。如下所示：

```
Configuration.ReplaceService<IAbpSession, MySession>(DependencyLifeStyle.Transient);
```

**ReplaceService** 方法通过传递一个 **Action** 来实现替换，这样我们就可以使用自定义的扩展类型替换掉原来的类型(你也可以直接使用Castle Windsor直接调用更高级的API函数)。

同一个服务可以被替换多次(尤其是在不同的模块)。最后替换的类型才是最终的使用类型。

### 1.4.2 配置模块

和.net框架原生的启动配置相比较，abp有哪些不一样呢？abp框架的模块可以通过IAbpModuleConfigurations接口进行个性化的扩展，这样的话，模块配置更加简单、方便。

示例代码如下：

```
using Abp.Web.Configuration;

public override void PreInitialize()
{
    Configuration.Modules.AbpWeb().SendAllExceptionsToClients = true;
}
```

在上面这个例子中，我们通过配置AbpWeb模块，发送异常到客户端。当然了，不是每一个模块都需要这种配置，通常情况下我们需要，是当一个模块需要在多个不同的应用中重复使用，我们才进行这样的配置。

### 1.4.3 为一个模块创建配置

如下代码，假如我们有一个命名为MyModule的模块，并且这各模块有一些自己的配置。那么我们首先要创建一些类，这些类定义为属性（译者注：属性有自动的get和set访问器。），代表了不同的配置。

```
public class MyModuleConfig
{
    public bool SampleConfig1 { get; set; }

    public string SampleConfig2 { get; set; }
}
```

接下来，我们通过依赖注入，注册这个类。IocManager.Register();



译者注：在IocManager中注册了一个类，换句话说，我们通过IocManager可以得到这个类MyModuleConfig的实例。至于IOC的原理这里就不在详细说了，总之，就是可以得到一个类的实例。

最后，我们通过创建一个扩展的方法IModuleConfigurations来得到配置的引用。如下代码：

```
public static class MyModuleConfigurationExtensions
{
    public static MyModuleConfig MyModule(this IModuleConfigurations moduleConfigurations)
    {
        return moduleConfigurations.AbpConfiguration
            .GetOrCreate("MyModuleConfig",
                () => moduleConfigurations.AbpConfiguration.IocManager.Resolve<MyModuleConfig>());
    }
}
```

这是模块配置的接口

这是接口的实现

译者注：模块配置是一个静态类，因为我们需要重复使用它。静态方法Mymodule返回的是一个配置接口，参数是IModuleConfigurations接口。

现在，在其他模块中也可以配置我们自定义的这个MyModule模块了。 Configuration.Modules.MyModule().SampleConfig1 = false; Configuration.Modules.MyModule().SampleConfig2 = "test";

在某种意义上，MyModule需要这些配置，通过注入MyModuleConfig我们就可以使用这些值。

```
public class MyService : ITransientDependency
{
    private readonly MyModuleConfig _configuration;

    public MyService(MyModuleConfig configuration)
    {
        _configuration = configuration;
    }

    public void DoIt()
    {
        if (_configuration.SampleConfig2 == "test")
        {
            //...
        }
    }
}
```

这意味着，在abp框架的系统中，所有的模块都可以集中配置。## 1.5 ABP总体介绍 - 多租户

### 1.5.1 什么是多租户

维基百科：“软件多租户是指一个软件架构的实例软件运行在一个服务器上，但存在多个租户。租户是一组共享一个公共的用户访问特定权限的软件实例。多租户架构,软件应用程序旨在提供每个租户专用的实例包括数据、配置、用户管理、租户个体功能和非功能属性。多租户与多实例架构,独立的软件实例代表不同的租户”操作

多租户一般用来创建SaaS(软件即服务)应用程序（云计算），下面有一些案例：

### 1.5.2 多个部署多个数据库

这实际上并不是多租户，如果为每个客户（租户）配置一个单独的数据库和应用程序的一个实例，即在单个服务器中部署但提供给多个客户（租户）使用，我们需要确保应用程序的多个实例不会因为系统相同的配置环境而发生冲突。这种已有的设计方式也不是真正为多租户服务的，它的好处是更容易的创建，但是存在一些安装、使用和维护的问题。

### 1.5.3 单个部署多个数据库

应用这种方式，我们能够仅在服务器上运行应用程序的一个实例。当某个用户登录时，从主数据库中检测此租户的信息并获取对应的数据库信息（连接字符串），然后我们连接到对应的数据库中。

这样的应用程序也可以算是多租户设计，但在安装、使用和维护上也存在一些问题。

### 1.5.4 单个部署单个数据库

这是真正的多租户构架，我们只在服务器上部署应用程序的单一实例且只有一个数据库。在各表中使用TenantId来隔离其它租户的信息。



这样的好处是易于安装和维护，但创建这样的应用程序比较困难。因为，需要防止租户读写其它租户的信息。在用户读取数据时候可以添加TenantId过滤器过滤数据，同样，系统会检测用户的写入操作。这是很繁琐的，而且也容易出错。ABP可以帮助我们自动数据过滤。

## 1.5.5 ABP中的多租户

ABP提供了一个创建单一部署、单一数据库、多租户的构架。

### 1. Host VS 租户

首先，我们先定义两个多租户系统中的术语：

- 租户：客户有它自己的用户,角色,权限,设置...并使用应用程序与其他租户完全隔离。多租户应用程序将有一个或多个租户。如果这是一个CRM应用程序,不同的租户也他们自己的帐户、联系人、产品和订单。所以,当我们说一个租户的用户,我们的意思是用户拥有的租户。
- Host: Host是单例的（只有唯一一个Host）。Host负责创建和管理租户。所以Host用户独立与租户且可以控制租户。

### 2. Session

ABP定义IAbpSession接口来获取当前用户和租户id。这个接口中使用多租户当前租户的id。因此,它可以基于当前租户的id过滤数据。

一些规则：

- 如果两个用户id和TenantId是null,那么当前用户没有登录到系统中。所以,我们不知道这是一个主机用户或租户的用户。在这种情况下,用户不能访问授权的内容。
- 用户id(如果不为空,TenantId为空的,然后我们可以知道当前用户是一个主机用户。
- 用户id(如果不为空,TenantId也不为空,我们可以知道当前用户是一个租户的用户。

有关更多的Session内容可查看：[Session](#)

### 3. 数据过滤

当我们从数据库检索实体，我们必须添加一个TenantId过滤当前的租户实体。当你实现了接口：IMustHaveTenant或IMayHaveTenant中的一个时，ABP将自动完成数据过滤。

#### IMustHaveTenant Interface

这个接口通过定义TenantId水下那个来区分实体不同的租户。示例：

```
public class Product : Entity, IMustHaveTenant
{
    public int TenantId { get; set; }

    public string Name { get; set; }

    //...other properties
}
```

因此，ABP能发现这是一个与租户相关的实体，并自动隔离其它租户的实体。

#### IMayHaveTenant interface

我们可能需要共享Host和租户中间的实体类型。一个实体可能属于租户或Host,MayHaveTenant接口还定义了TenantId(类似于IMustHaveTenant),但在这种情况下可以为空。一个示例实现

```
public class Role : Entity, IMayHaveTenant
{
    public int? TenantId { get; set; }

    public string RoleName { get; set; }

    //...other properties
}
```

我们可以使用相同的角色类存储主机角色和租户的角色。TenantId属性,在这种情况下,如果这是一个主机实体或租户entity。null值意味着这是一个Host的实体,一个非空值意味着这实体租户Id是的TenantId。

IMayHaveTenant并不普遍。例如,一个产品类不能IMayHaveTenant自相关产品实际应用功能,管理租户无关。所以,小心使用IMayHaveTenant接口, 因为很难保持代码共享的主机和租户。

保存实体

租户的用户不应该创建/编辑其他租户的实体。ASP。净样板检查它在保存更改数据库相关数据过滤器是否启用。

## 1.6 ABP总体介绍 - 集成OWIN

### 1.6.1 集成OWIN

如果在你的项目中你用到了 **OWIN** , 你只需要Nuget来添加 **Abp.Owin**到你主项目中(一般来说是 **Web** 项目) 并且在OWIN Startup调用 **UserApp()** 的扩展方法, 如下所示:

```
[assembly: OwinStartup(typeof(Startup))]
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.UseAbp();

        //其它配置...
    }
}
```


## 2.1 ABP公共结构 - 依赖注入


```

如果你已经了解依赖注入的概念、构造函数和属性注入模式, 你可以跳过这一节。

维基百科: “依赖注入是一种软件设计模式, 指一个或多个依赖 (或服务) 被注入, 或通过引用传递, 传入一个依赖对象 (或客户端) 并成为客户端

如果不使用依赖注入技术, 很难进行依赖管理、模块化开发和应用程序模块化。

#### ### 2.1.1 传统方式的问题

在一个应用程序中, 类之间相互依赖。假设我们有一个应用程序服务, 使用仓储 (**repository**) 类插入实体到数据库。在这种情况下, 应用程序

```
``` csharp
public class PersonAppService
{
    private IPersonRepository _personRepository;

    public PersonAppService()
    {
        _personRepository = new PersonRepository();
    }

    public void CreatePerson(string name, int age)
    {
        var person = new Person { Name = name, Age = age };
        _personRepository.Insert(person);
    }
}
```
```

PersonAppService使用PersonRepository插入Person到数据库。这段代码的问题:

- PersonAppService通过IPersonRepository调用CreatePerson方法, 所以这方法依赖于IPersonRepository, 代替了PersonRepository具体类。但PersonAppService (的构造函数) 仍然依赖于PersonRepository。组件应该依赖于接口而不是实现。这就是所谓的依赖性倒置原则。
- 如果PersonAppService亲自创建PersonRepository, 这变得依赖到了一个特定的IPersonRepository接口实现, 且不能使用另一个实现进行工作。因此, 从实现分离接口变得无意义, 硬依赖使得代码基于紧耦合和低重用。硬依赖 (hard-dependency) 使得代码紧密耦合和较低的可重用。
- 我们可能需要在未来改变创建PersonRepository的方式。即, 我们可能想让它创建为单例(单一共享实例而不是为每个使用创建一个对象)。或者我们可能想要创建多个类实现IPersonRepository并根据条件创建对象。在这种情况下, 我们需要修改所有依赖于IPersonRepository的类。
- 有了这样的依赖, 很难(或不可能)对PersonAppService进行单元测试。

为了克服这些问题, 可以使用工厂模式。因此, 创建的仓储类是抽象的。看下面的代码:

```
public class PersonAppService
{
    private IPersonRepository _personRepository;

    public PersonAppService()
    {
        _personRepository = PersonRepositoryFactory.Create();
    }

    public void CreatePerson(string name, int age)
    {
        var person = new Person { Name = name, Age = age };
        _personRepository.Insert(person);
    }
}
```

PersonRepositoryFactory是一个静态类，创建并返回一个IPersonRepository。这就是所谓的服务定位器模式。以上依赖问题得到解决，因为PersonAppService不需要创建一个IPersonRepository的实现的对象，这个对象取决于PersonRepositoryFactory的Create方法。但是，仍然存在一些问题：

- □ 此时，PersonAppService取决于PersonRepositoryFactory。这是更容易接受，但仍有一个硬依赖（hard-dependency）。
- □ 为每个库或每个依赖项乏味的写一个工厂类/方法。
- □ 测试性依然不好，由于很难使得PersonAppService使用mock实现IPersonRepository。

## 2.1.2 解决方案

有一些最佳实践(模式)用于类依赖。

### 1. 构造函数注入(Constructor injection)

重写上面的例子，如下所示：

```
public class PersonAppService
{
    private IPersonRepository _personRepository;

    public PersonAppService(IPersonRepository personRepository)
    {
        _personRepository = personRepository;
    }

    public void CreatePerson(string name, int age)
    {
        var person = new Person { Name = name, Age = age };
        _personRepository.Insert(person);
    }
}
```

这被称为构造函数注入。现在，PersonAppService不知道哪些类实现IPersonRepository以及如何创建它。谁需要使用PersonAppService，首先创建一个IPersonRepository PersonAppService并将其传递给构造函数，如下所示：

```
var repository = new PersonRepository();
var personService = new PersonAppService(repository);
personService.CreatePerson("Yunus Emre", 19);
```

构造函数注入是一个完美的方法，使一个类独立创建依赖对象。但是，上面的代码有一些问题：

- □ 创建一个PersonAppService变得困难。想想如果它有4个依赖，我们必须创建这四个依赖对象，并将它们传递到构造函数PersonAppService。
- □ 从属类可能有其他依赖项(在这里，PersonRepository可能有依赖关系)。所以，我们必须创建PersonAppService的所有依赖项，所有依赖项的依赖关系等等。如此，依赖关系使得我们创建一个对象变得过于复杂了。

幸运的是，依赖注入框架能够自动化管理依赖关系。

### 2. 属性注入(Property injection)

采用构造函数的注入模式是一个完美的提供类的依赖关系的方式。通过这种方式，只有提供了依赖你才能创建类的实例。同时这也是一个强大的方式显式地声明，类需要什么样的依赖才能正确的工作。

但是，在有些情况下，该类依赖于另一个类，但也可以没有它。这通常是适用于横切关注点(如日志记录)。一个类可以没有工作日志，但它可以写日志如果你提供一个日志对象。在这种情况下，你可以定义依赖为公共属性，而不是让他们放在构造函数。想想，如果我们想在PersonAppService写日志。我们可以重写类如下：

```
public class PersonAppService
{
    public ILogger Logger { get; set; }

    private IPersonRepository _personRepository;

    public PersonAppService(IPersonRepository personRepository)
    {
        _personRepository = personRepository;
        Logger = NullLogger.Instance;
    }

    public void CreatePerson(string name, int age)
    {
        Logger.Debug("Inserting a new person to database with name = " + name);
        var person = new Person { Name = name, Age = age };
        _personRepository.Insert(person);
    }
}
```

NullLogger.Instance 是一个单例对象，实现了ILogger接口，但实际上什么都没做(不写日志。它实现了ILogger实例，且方法体为空)。现在，PersonAppService可以写日志了，如果你为PersonAppService实例设置了Logger，如下面：

```
Var personService = new PersonAppService(new PersonRepository());
personService.Logger = new Log4NetLogger();
personService.CreatePerson("Yunus Emre", 19);
```

假设Log4NetLogger实现ILogger实例，使得我们可以使用Log4Net库写日志。因此，PersonAppService可以写日志。如果我们不设置Logger，PersonAppService就不写日志。因此，我们可以说PersonAppService ILogger实例是一个可选的依赖。

几乎所有的依赖注入框架都支持属性注入模式。

### 3. 依赖注入框架

有许多依赖注入框架，都可以自动解决依赖关系。他们可以创建所有依赖项(递归地依赖和依赖关系)。所以你只需要依赖注入模式写类和类构造函数&属性，其他的交给DI框架处理！在良好的应用程序中，类甚至独立于DI框架。整个应用程序只会有几行代码或类，显示的与DI框架交互。

ABP的依赖注入基于 Castle Windsor框架。Castle Windsor最成熟的DI框架之一。还有很多这样的框架，如Unity，Ninject，StructureMap，Autofac等等。

在使用一个依赖注入框架时，首先注册你的接口/类到依赖注入框架中，然后你就可以resolve一个对象。在Castle Windsor，它是这样的：

```
var container = new WindsorContainer();

container.Register(
    Component.For<IPersonRepository>().ImplementedBy<PersonRepository>().LifestyleTransient(),
    Component.For<IPersonAppService>().ImplementedBy<PersonAppService>().LifestyleTransient()
);

var personService = container.Resolve<IPersonAppService>();
personService.CreatePerson("Yunus Emre", 19);
```

我们首先创建了WindsorContainer。然后注册PersonRepository 和 PersonAppService及它们的接口。然后我们要求容器创建一个IPersonAppService实例。它创建PersonAppService对象及其依赖项并返回。在这个简单的示例中，使用DI框架也许不是那么简洁，但想象下，在实际的企业应用程序中你会有很多类和依赖关系。当然，注册的依赖项只在程序启动的某个地方创建一次。

请注意，我们只是将对象声明为临时对象(transient)。这意味着每当我们创建这些类型的一个对象时，就会创建一个新的实例。有许多不同的生命周期(如Singleton单例模式)。

### 4. ABP依赖注入的基础结构

在编写应用程序时遵循最佳实践和一些约定，ABP几乎让依赖注入框架使用变得无形。

### 注册(Registering)

在ABP中，有很多种不同的方法来注册你的类到依赖注入系统。大部分时间，常规方法就足够了。

#### 常规注册(Conventional registrations)

按照约定，ABP自动注册所有 Repositories， Domain Services， Application Services， MVC 控制器和Web API控制器。例如，你可能有一个IPersonAppService 接口和实现类PersonAppService：

```
public interface IPersonAppService : IApplicationService
{
    //...
}

public class PersonAppService : IPersonAppService
{
    //...
}
```

ABP会自动注册它，因为它实现IApplicationService接口(它只是一个空的接口)。它会被注册为transient (每次使用都创建实例)。当你注入(使用构造函数注入)IPersonAppService接口成一个类，PersonAppService对象会被自动创建并传递给构造函数。

注意：命名约定在这里非常重要。例如你可以将名字PersonAppService改为 MyPersonAppService或另一个包含“PersonAppService”后缀的名称，由于IPersonAppService包含这个后缀。但是你可以不遵循PeopleService命名你的服务类。如果你这样做，它将不会为IPersonAppService自动注册(它需要自注册（self-registration）到DI框架，而不是接口)，所以，如果你想要你应该手动注册它。

ABP按照约定注册程序集。所以，你应该告诉ABP按照约定注册你的程序集。这很容易：

```
IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
```

Assembly.GetExecutingAssembly()得到一个对包括此代码的程序集的引用。你可以通过RegisterAssemblyByConvention方法注册其他程序集。这同在你的模块初始化（AbpModule.Initialize()）时完成。请查看ABP的模块系统获得更多信息。

你可以通过实现IConventionalRegisterer接口和调用IocManager.AddConventionalRegisterer方法编写自己的约定注册类。你应该将它添加到模块的pre-initialize方法中。

#### 帮助接口(Helper interfaces)

你可以注册一个特定的类，不遵循传统的约定制度规则。ABP提供了ITransientDependency和ISingletonDependency接口的快捷方法。例如：

```
public interface IPersonManager
{
    //...
}

public class MyPersonManager : IPersonManager, ISingletonDependency
{
    //...
}
```

以这种方式，你可以很容易地注册MyPersonManager为transient。当需要注入IPersonManager时，MyPersonManager会被使用。注意，依赖被声明为单例。因此，创建的MyPersonManager同一个对象被传递给所有需要的类。只是在第一次使用时创建，那么应用程序的整生命周期使用的是同一实例。

#### 自定义/直接注册(Custom/Direct registration)

如果之前描述的方法还是不足以应对你的情况，你可以使用Castle Windsor注册类和及依赖项。因此，你将拥有Castle Windsor注册的所有能力。

可以实现IWindsorInstaller接口进行注册。你可以在应用程序中创建一个实现IWindsorInstaller接口的类：

```
public class MyInstaller : IWindsorInstaller
{
```

```
public void Install(IWindsorContainer container, IConfigurationStore store)
{
    container.Register(Classes.FromThisAssembly().BasedOn<IMySpecialInterface>().LifestylePerThread().WithSer
}
}
```

Abp自动发现和执行这个类。最后，你可以通过使用IocManager.IocContainer属性得到WindsorContainer。有关更多信息，阅读Windsor的文档。

#### 解析 (Resolving)

注册通知IOC(控制反转)容器关于你的类，它们的依赖项和生命周期。在你的应用程序需要使用IOC容器创建对象时，ASP.NET提供了一些方法解决依赖关系。

#### 构造函数 & 属性注入(Constructor & Property Injection)

作为最佳实践，应该使用构造函数和属性注入去获取类的依赖。例子：

```
public class PersonAppService
{
    public ILogger Logger { get; set; }

    private IPersonRepository _personRepository;

    public PersonAppService(IPersonRepository personRepository)
    {
        _personRepository = personRepository;
        Logger = NullLogger.Instance;
    }

    public void CreatePerson(string name, int age)
    {
        Logger.Debug("Inserting a new person to database with name = " + name);
        var person = new Person { Name = name, Age = age };
        _personRepository.Insert(person);
        Logger.Debug("Successfully inserted!");
    }
}
```

IPersonRepository从构造函数注入，ILogger实例从公共属性注入。这样，你的代码不会体现依赖注入系统。这是使用DI系统最适当的方式。

#### IIocResolver 和 IIocManager接口

有时可能需要直接创建所需的依赖项，而不是构造函数和属性注入。（应该尽可能避免这种情况）。Abp提供一些服务使得这样的注入很容易实现。例子：

```
public class MySampleClass : ITransientDependency
{
    private readonly IIocResolver _iocResolver;

    public MySampleClass(IIocResolver iocResolver)
    {
        _iocResolver = iocResolver;
    }

    public void DoIt()
    {
        //Resolving, using and releasing manually
        var personService1 = _iocResolver.Resolve<PersonAppService>();
        personService1.CreatePerson(new CreatePersonInput { Name = "Yunus", Surname = "Emre" });
        _iocResolver.Release(personService1);

        //Resolving and using in a safe way
        using (var personService2 = _iocResolver.ResolveAsDisposable<PersonAppService>())
        {
            personService2.Object.CreatePerson(new CreatePersonInput { Name = "Yunus", Surname = "Emre" });
        }
    }
}
```



MySampleClass是一个应用程序的示例类。IocResolver通过构造函数注入，然后用它来创建和释放对象。有几个解决方法的重载可以根据需要使用。Release方法用于释放组件(对象)。如果你是手动创建一个对象，调用Release方法释放对象非常重要。否则，你的应用程序会有内存泄漏问题。为了保证对象被释放，尽可能使用ResolveAsDisposable(就像上面的例子所示)。它会在using代码块结束的时候自动调用Release方法。

如果你想直接使用IOC容器(Castle Windsor)来处理依赖关系项，可以通过构造函数注入 IocManager并使用它 IocManager.IocContainer 属性。如果你是在一个静态上下文或不能注入IocManager，还有最后一个方法，你可以使用单例对象IocManager.Instance，你可以在任何地方获取到，它无处不在。但是，在这种情况下你的代码将变得不容易测试。

### 3.1.3 附件

#### 1. IShouldInitialize 接口

有些类在第一次使用前需要初始化。IShouldInitialize有Initialize()方法。如果你实现它，那么你的Initialize()方法自动会被自动调用在创建对象之后(在使用之前)。当然，为了使用这个特性，你应该注入/创建此对象。

#### 2. ASP.NET MVC & ASP.NET Web API 集成

当然，我们必须调用依赖注入系统处理依赖关系图的根对象。在一个ASP.NET MVC应用程序，通常是一个控制器类。我们可以使用构造函数注入模式注入控制器。当一个请求来到我们的应用程序中，控制器和所有依赖项被IOC容器递归创建。所以，谁做了这些？这是被Abp扩展的ASP.NET MVC默认控制器工厂自动完成的。ASP.NET Web API 也是相似的。你不用关心对象的创建和释放。

#### 3. 最后说明 (Last notes)

Abp简化并自动使用依赖注入，只要你遵守规则和使用上面的结构。大多数时候这样就够了。但是如果不能满足你的需求，你可以直接使用Castle Windsor的所有能力来执行任何任务(如自定义注册，注入钩子，拦截器等等)。

## 2.2 ABP公共结构 - 会话管理

### 2.2.1 简介

如果一个应用程序需要登录，则它必须知道当前用户执行了什么操作。因此ASP.NET在展示层提供了一套自己的SESSION会话对象，而ABP则提供了一个可以在任何地方获取当前用户和租户的IAbpSession接口

注意：关于IAbpSession接口：需要获取会话信息则必须实现IAbpSession接口。虽然你可以用自己的方式去实现它（IAbpSession），但是它在module-zero项目中已经有了完整的实现。

IAbpSession也被集成且使用在ABP其他模块中，例如：设置管理，权限认证。

### 2.2.2 注入会话

IAbpSession通常是以属性注入的方式存在于需要它的类中，不需要获取会话信息的类中则不需要它。如果我们使用属性注入方式，我们可以用NullAbpSession.Instance作为默认值来初始化它（IAbpSession），如下所示：

```
public class MyClass : ITransientDependency
{
    public IAbpSession AbpSession { get; set; }

    public MyClass()
    {
        AbpSession = NullAbpSession.Instance;
    }

    public void MyMethod()
    {
        var currentUserId = AbpSession.UserId;
        //...
    }
}
```

由于授权是应用层的任务，因此我们应该在应用层和应用层的上一层使用IAbpSession（我们不在领域层使用IAbpSession是很正常的）。

ApplicationService, AbpController 和 AbpApiController 这3个基类已经注入了AbpSession属性，因此在Application Service的实例方法中，能直接使用AbpSession属性。



## 2.2.3 使用会话属性

AbpSession定义的一些关键属性:

- **UserId**: 当前用户的标识ID, 如果没有当前用户则为null。如果调用的代码是已授权, 那么它不可能为空。
- **TenantId**: 当前租户的标识ID, 如果没有当前租户则为null(如果用户没有登录或者他是一个Host用户)。
- **ImpersonatorUserId**: 模拟用户的标识ID, 如果当前会话被其他用户模拟登录。如果不是一个模拟登录, 那么该值为空。
- **ImpersonatorTenantId**: 模拟用户租户的标识ID, 如果当前会话被其他用户模拟登录。如果不是一个模拟登录, 那么该值为空。
- **MultiTenancySide**: 可能是Host或Tenant。

UserId和TenantId是可空类型。当然也提供了不为空时获取数据的 **GetUserId()** 和 **GetTenantId()** 方法。当你确定有当前用户时, 你可以使用GetUserId()方法。

如果当前用户为空, 使用该方法则会抛出一个异常。GetTenantId()的使用方式和GetUserId()类似。

模拟属性不像其它通用属性一样用于审计日志。

## 2.2.4 User Identifier

你可以使用 **ToUserIdentifier()** 扩展方法从IAbpSession创建一个UserIdentifier对象。因为UserIdentifier被大多数API使用, 所以使用该方法可以很方便的为当前用户创建一个UserIdentifier。## 2.3 ABP公共结构 - 缓存管理

### 2.3.1 简介

ABP提供了一个抽象的缓存基类 (CacheBase), 并且在内部使用 MemoryCache来实现了这个抽象类的所有功能, 当然你也能够用其他的缓存提供者来实现它。(例如: Redis 或者 Memcached)

### 2.3.2 ICacheMananger

ABP对外提供了一个缓存接口ICacheMananger。我们通过构造函数注入这个接口来获取缓存。示例如下:

```
public class TestAppService : ApplicationService
{
    private readonly ICacheManager _cacheManager;

    public TestAppService(ICacheManager cacheManager)
    {
        _cacheManager = cacheManager;
    }

    public Item GetItem(int id)
    {
        //Try to get from cache
        return _cacheManager
            .GetCache("MyCache")
            .Get(id.ToString(), () => GetFromDatabase(id)) as Item;
    }

    public Item GetFromDatabase(int id)
    {
        //... retrieve item from database
    }
}
```

在这个示例中, 我们注入了 ICacheManager接口, 并且获取了一个名称为MyCache的缓存。

注意: **GetCache**方法

千万不要在你的构造函数中使用GetCache方法, 如果你的类是一个瞬时对象 (transient) 这可能导致你的缓存被dispose掉。

### 2.3.3 ICache

ICacheManager.GetCache方法返回了一个ICache对象。每一个缓存都是基于名称单例存在的。只有首次访问时才会被创建, 以后你每次用相同的名称去获取的缓存都是相同的。所以我们可以不同的类中使用相同的名称来共享相同的缓存。

在示例代码中, 我们简单的使用了ICache.Get方法, 它有两个参数:

- key: 要获取的缓存项的唯一标识符
- factory: 如果根据给定的key获取到的缓存项为空, 那么factory将会创建一个标识符为key的缓存, 并且返回该缓存

ICache接口还有其它方法, 例如: GetOrDefault, Set, Remove和Clear。当然也有这些方法的异步 (async) 版本。

### 2.3.4 ITypedCache

ICache接口用key (字符串类型) 来获取缓存value (object类型)。ITypedCache为ICahe提供了一个类型安全的包装; 为了使类型安全转换 (ICache到ITypedCache), 我们可以用扩展方法AsTyped, 而不需要写其它强制类型转换的代码, 如下所示:

```
ITypedCache<int, Item> myCache = _cacheManager.GetCache("MyCache").AsTyped<int, Item>();
```

### 2.3.5 Configuration

缓存的过期时间默认是60分钟。如果你想改变所有的缓存或者指定的缓存来的默认过期时间, 你可以这样做, 实现如下:

```
//Configuration for all caches
Configuration.Caching.ConfigureAll(cache =>
{
    cache.DefaultSlidingExpireTime = TimeSpan.FromHours(2);
});

//Configuration for a specific cache
Configuration.Caching.Configure("MyCache", cache =>
{
    cache.DefaultSlidingExpireTime = TimeSpan.FromHours(8);
});
```

这段代码你应该放在模块 (module) 的PreInitialize方法中。如上所示: MyCache将会在8小时后过期, 而其他的缓存将在2小时后过期。

这些配置将会在首次创建缓存的时候生效。配置不仅仅局限于DefaultSlidingExpireTime, 你可以利用ICache接口中的属性获取方法来自由的配置并且初始化它们。

### 2.3.6 Entity Caching

ABP的缓存系统是以通用为目的, 它有一个 **EntityCache** 基类, 如果你需要的话, 这个基类可以帮助你缓存实体。使用这个基类, 我们可以通过ID取得实体, 并且我们通过ID来缓存实体, 这样以后就不需要频繁的查询数据库去取得实体。假设我们有个Person实体, 像下面一样:

```
public class Person : Entity
{
    public string Name { get; set; }

    public int Age { get; set; }
}
```

并且, 假设我们通过该实体的Id, 需要频繁调用取得Person实体的Name。首先, 我们应该创建一个类来存储 **cache items**:

```
[AutoMapFrom(typeof(Person))]
public class PersonCacheItem
{
    public string Name { get; set; }
}
```

我们 不应该直接存储实体到缓存中 因为缓存的时候需要序列化缓存对象而实体可能不能被序列化(尤其是实体的导航属性)。这就是为什么我们定义了一个简单的像DTO的类来存储数据到缓存中。我们添加了 **AutoMapFrom** 特性, 这是因为我们想使用 **AutoMapper** 来自动的转换 **Person** 实体为 **PersonCacheItem** 对象。如果我们不使用 **AutoMapper**, 那么我们应该重写 **EntityCache** 类的 **MapToCacheItem** 方法手动转换/映射它。

然而这不是必须的, 我们可能想定义一个接口为缓存类:

```
public interface IPersonCache : IEntityCache<PersonCacheItem>
{
}
```

```
}
```

最后，我们可以创建缓存类来缓存Person实体：

```
public class PersonCache : EntityCache<Person, PersonCacheItem>, IPersonCache, ITransientDependency
{
    public PersonCache(ICacheManager cacheManager, IRepository<Person> repository)
        : base(cacheManager, repository)
    {
    }
}
```

这样就OK了，我们的person缓存已经准备好可以使用了。缓存类可以使瞬时(如同这个例子)或者是单例。这不是说缓存数据是瞬态的。在你的应用程序中它一直是全局缓存并且是线程安全的。

现在，无论在什么地方我们需要取得Person的Name，我们可以通过Person的Id从缓存中取得它。如下所示：

```
public class MyPersonService : ITransientDependency
{
    private readonly IPersonCache _personCache;

    public MyPersonService(IPersonCache personCache)
    {
        _personCache = personCache;
    }

    public string GetPersonNameById(int id)
    {
        return _personCache[id].Name; //alternative: _personCache.Get(id).Name;
    }
}
```

我们很容易的注入 IPersonCache 接口，通过该接口取得缓存项和Name属性。

那么EntityCache是怎么工作的？

- 在首次调用的时候我们通过仓储从数据库中取得实体。那么随后的调用都是从缓存中取得。
- 如果实体被更新或者删除，它会自动的无效实体。因此，它会在下次调用的时候重新从数据库中检索数据。
- 使用 IObjectMapper 接口来映射实体到缓存项。IObjectMapper 接口在 AutoMapper 中被实现。所以，如果你使用了自动映射，那么就需要 AutoMapper模块。你可以重写 MapToCacheItem 方法手动映射它到缓存项。
- 使用缓存类的FullName作为缓存的Name，你可以通过传入的缓存名到基类的构造函数来改变它。
- 它是线程安全的。

如果你有更复杂的缓存需求，那么你需要扩展 EntityCache 类或者创建你自己的解决方案。

## Redis Cache 集成

默认Cache Manager是使用 in-memory 来缓存。所以，这可能会成为一个问题，如果有多个并发的Web服务运行在同一个应用中。在这种情况下，你可能想要一个分布式/中央缓存服务器。那么，你可以使用Redis来作为你的缓存服务。

首先，你需要安装 Abp.RedisCache nuget package 到你的项目中(你可以安装它到你的Web项目)。那么你需要为 AbpRedisCacheModule 添加 DependsOn 特性，并且在你模块的 PreInitialize 方法中调用 UseRedis 扩展方法，如下所示：

```
//...其他名称空间的引用
using Abp.Runtime.Caching.Redis;

namespace MyProject.AbpZeroTemplate.Web
{
    [DependsOn(
        //...其他依赖模块
        typeof(AbpRedisCacheModule))]
    public class MyProjectWebModule : AbpModule
    {
        public override void PreInitialize()
        {
            //...其他配置
        }
    }
}
```

```
        Configuration.Caching.UseRedis();
    }

    //...其他代码
}
}
```

Abp.RedisCache package 使用 **"localhost"** 作为默认的连接字符串。你可以添加连接字符串到你的配置文件中来覆盖它。如：

```
<add name="Abp.Redis.Cache" connectionString="localhost"/>
```

你也可以添加配置到appSettings来设置Redis数据库的Id。如：

```
<add key="Abp.Redis.Cache.DatabaseId" value="2"/>
```

在同一个服务器上使用不同的数据库Id是非常有用的这可以创建不同的Key Spaces(隔离缓存)。

**UseRedis** 有一个重载方法，你可以通过这个方法传入配置参数，这可以覆盖掉配置文件中的配置。

关于Redis的其他配置可以查看 [Redis文档](#)。

当然译者也简单的对[StackExchange.Redis](#)进行了翻译，吐槽一下这个文档的作者可能真的是一个真正的程序员，你懂的。

注意：在ABP中使用Redis缓存你需要安装Redis服务并使其运行。## 2.4 ABP公共结构 - 日志管理

### 2.4.1 服务器端

ABP使用Castle Windsor's logging facility日志记录工具，并且可以使用不同的日志类库，比如：Log4Net, NLog, Serilog... 等等。对于所有的日志类库，Castle提供了一个通用的接口来实现，我们可以很方便的处理各种特殊的日志库，而且当业务需要的时候，很容易替换日志组件。

译者注：Castle是什么?Castle是针对.NET平台的一个开源项目，从数据访问框架ORM到IOC容器，再到WEB层的MVC框架、AOP，基本包括了整个开发过程中的所有东西。ASP.NET Boilerplate的ioc容器就是通过Castle实现的。

Log4Net是asp.net下面最流行的一个日志库组件，ASP.NET Boilerplate 模板也使用了Log4Net日志库组件，但是呢，我们这里仅仅通过一行关键代码就实现Log4Net 的依赖注入（具体说明在下面的配置文件），所以，如果你想替换成自己的日志组件，也很容易。

#### 1. 获取日志记录器(logger)

不管你选择哪一个日志库组件，通过代码来进行日志记录都是一样的。（这里吐槽，Castle's 通用 ILogger 接口实在太牛逼了）。

下面进入正题：(注:下面的代码是abp框架的Castle.Core源码分析以及实现)

首先呢，我们要先处理日志记录器对象logger，ASP.NET Boilerplate框架使用了dependency injection依赖注入技术,我们可以很方便的使用依赖注入生成日志记录器对象logger。

接下来我们看一下 ASP.NET Boilerplate是怎么实现日志记录功能的吧：

```
//1: 导入日志的命名空间, Castle.Core.Logging
using Castle.Core.Logging;

public class TaskAppService : ITaskAppService
{
    /*
    2:通过依赖注入获取日志记录器对象。
    这里先定义了一个ILogger类型的public属性Logger，这个对象就是我们用来记录日志的对象。在创建了TaskAppService对象（就是我们应
    */
    public ILogger Logger { get; set; }

    public TaskAppService()
    {
        /*
        3: 如果没有日志记录器，将日志记录器返回一个空的实例，不写日志。这是依赖注入的最佳实现方式，
        如果你不定义这个空的日志记录器，当我们获取对象引用并且实例化的时候，就会产生异常。
        这么做，保证了对象不为空。所以，换句话说，不设置日志记录器，就不记录日志，返回一个null的对象。
        NullLogger对象实际上什么都木有，空的。这么做，才能保证我们定义的类在实例化时正常运作。
        */
    }
}
```

```

    Logger = NullLogger.Instance;
}

public void CreateTask(CreateTaskInput input)
{
    //4: 写入日志
    Logger.Info("Creating a new task with description: " + input.Description);

    //TODO: save task to database...
}
}

```

写入日志以后，我们可以查看日志文件，就像下面的格式：

```
INFO 2014-07-13 13:40:23,360 [8] SimpleTaskSystem.Tasks.TaskAppService - Creating a new task with description:Remembe
```

## 2. 通过基类使用日志记录(Logger)

ABP提供了MVC Controllers、Web API Controllers和Application service classes的基类（自己定义的控制器和应用服务，都必须继承ABP的基类，换句话说，当你自定义的Web API controllers、mvc controllers，Application service classes都继承了ABP框架对应的基类，你就可以直接使用日志记录器）。

```

public class HomeController : SimpleTaskSystemControllerBase
{
    public ActionResult Index()
    {
        Logger.Debug("A sample log message...");
        return View();
    }
}

```

说明：SimpleTaskSystemControllerBase这个基类控制器是我们自己定义的基类控制器，他必须继承自 AbpController。

这样实现，日志记录器才能正常工作。当然了，你也可以实现自己的基类，这样的话你也可以不使用依赖注入了。

## 3. 配置(Configuration)

如果你在官网上通过ASP.NET Boilerplate templates 来生成了你的工程，Log4Net的所有配置都自动生成了。

默认的配置格式如下：

- Log level: 日志记录等级，有DEBUG, INFO, WARN, ERROR or FATAL5个。
- Date and time: 日志记录时间。
- Thread number: 每行日志写时候的线程号。
- Logger name: 日志记录器的名字，通常情况就是类名称。
- Log text: 你写入的日志内容。

配置文件：log4net.config 一般都在项目的web目录下面。

```

<?xml version="1.0" encoding="utf-8" ?>
<log4net>
  <appender name="RollingFileAppender" type="log4net.Appender.RollingFileAppender" >
    <file value="Logs/Logs.txt" />
    <appendToFile value="true" />
    <rollingStyle value="Size" />
    <maxSizeRollBackups value="10" />
    <maximumFileSize value="10000KB" />
    <staticLogFileName value="true" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-5level %date [%-5.5thread] %-40.40logger - %message%newline" />
    </layout>
  </appender>
  <root>
    <appender-ref ref="RollingFileAppender" />
    <level value="DEBUG" />
  </root>
  <logger name="NHibernate">

```

```
<level value="WARN" />
</logger>
</log4net>
```

Log4Net是一个非常强大和易用的日志库组件，你可以写各种日志，比如写到txt文件，写入到数据库等等。你能设置最小的日志等级，就像上面这个针对NHibernate的配置。不同的记录器写不同的日志，等等。

具体的用法大家可以参照：<http://logging.apache.org/log4net/release/config-examples.html>

最后，在工程的Global.asax 文件中，来定义Log4Net的配置文件：

```
public class MvcApplication : AbpWebApplication
{
    protected override void Application_Start(object sender, EventArgs e)
    {
        IocManager.Instance.IocContainer.AddFacility<LoggingFacility>(f => f.UseLog4Net().WithConfig("log4net.config"
        base.Application_Start(sender, e);
    }
}
```

几行代码就调用了Log4Net这个日志记录组件，工程中的Log4Net库是在 nuget package包中的，你也可以换成其他日志组件库，但是代码不用做任何改变。因为，我们的框架是通过依赖注入实现日志记录器的。

#### 4. Abp.Castle.Log4Net 包

ABP使用了 **Castle Logging Facility** 来记录日志，正如上面所述，它不直接的依赖 **log4net**。在这里我们有一个关于Castle Log4Net的集成问题，它不支持最新的log4net。我们创建了一个 **Abp.Castle.Log4Net** 包来解决这个问题。添加这个包到你的项目后，我们应该在 **Application\_Start** 里修改我们的代码，如下所示：

```
public class MvcApplication : AbpWebApplication
{
    protected override void Application_Start(object sender, EventArgs e)
    {
        IocManager.Instance.IocContainer.AddFacility<LoggingFacility>(f => f.UseAbpLog4Net().WithConfig("log4net.conf
        base.Application_Start(sender, e);
    }
}
```

唯一的不同点就是我们使用的是 **UseAbpLog4Net()** 方法(该方法定义在 **Abp.Castle.Logging.Log4Net**命名空间中) 来替换之前的 **UseLog4Net()** 方法。当我们使用Abp.Castle.Log4Net包时，我们将再也不需要使用 **Castle.Windsor-log4net** 和 **Castle.Core-log4net** 包。

#### 2.4.2 客户端

最后，更厉害的是，你还可以在客户端调用日志记录器。在客户端，ABP有对应的 javascript 日志API，这意味着你可以记录下来浏览器的日志，实现代码如下：

```
abp.log.warn('a sample log message...');
```

注意：客户端javascript的api，这里要说明的是，你可以使用console.log在客户端输出日志，但是这个API 不一定支持所有的浏览器，还有可能导致你的脚本出现异常，你可以使用我们的api，我们的是安全的，你甚至可以重载或者扩展这些api。

- abp.log.debug('...');
- abp.log.info('...');
- abp.log.warn('...');
- abp.log.error('...');
- abp.log.fatal('...');

## 2.5 ABP公共结构 - 设置管理

### 2.5.1 简介



每个应用程序需要存储一些设置并在应用程序的某个地方使用这些设置。ABP框架提供强大的基础架构，我们可以在服务端或者客户端设置，来存储/获取应用程序、租户和用户级别的配置。

设置通常是存储在数据库（或另一个来源）中，用名称-值（name-value）字符串对应的结构来表示。我们可以把非字符串值转换成字符串值来存储。

注意：关于ISettingStore接口:为了使用设置管理必须实现 ISettingStore 接口。你可以用自己的方式实现它，在module-zero项目中有完整的实现可以参考。

## 2.5.2 定义设置

使用设置之前必须先定义。ABP框架是模块化设计，所以不同的模块可以有不同的设置。为了定义模块自己的设置，每个模块都应该创建继承自SettingProvider 的派生类。设置提供程序示例如下所示：

```
public class MySettingProvider : SettingProvider
{
    public override IEnumerable<SettingDefinition> GetSettingDefinitions(SettingDefinitionProviderContext context)
    {
        return new[]
        {
            new SettingDefinition(
                "SmtpServerAddress",
                "127.0.0.1"
            ),

            new SettingDefinition(
                "PassiveUsersCanNotLogin",
                "true",
                scopes: SettingScopes.Application | SettingScopes.Tenant
            ),

            new SettingDefinition(
                "SiteColorPreference",
                "red",
                scopes: SettingScopes.User,
                isVisibleToClients: true
            )
        };
    }
}
```

GetSettingDefinitions 方法返回 SettingDefinition 对象。SettingDefinition 类的构造函数中有如下参数：

- Name (必填):必须具有全系统唯一的名称。比较好的办法是定义字符串常量来设置Name。
- Default value: 设置一个默认值。此值可以是null 或空字符串。
- Scopes: 定义设置的范围 (见下文)。
- Display name: 一个可本地化的字符串，用于以后在UI中显示设置的名称。
- Description: 一个可本地化的字符串，用于以后在UI中显示设置的描述。
- Group: 可用于设置组。这仅仅是UI使用，不用于设置管理。
- IsVisibleToClients: 设置为 true 将使设置在客户端可用。

在创建设置提供程序(SettingProvider)之后，我们应该在预初始化(PreInitialize)方法中注册我们的模块：

Configuration.Settings.Providers.Add();设置提供程序会自动注册依赖注入。所以，设置提供程序可以注入任何依赖项 (如存储库) 来生成设置定义的一些其它来源。

## 2.5.3 设置范围

有三个设置范围 (或级别) 在 SettingScopes 枚举中定义：

- Application: 应用程序范围设置用于用户/租户独立的设置。例如，我们可以定义一个名为"SmtpServerAddress"的设置，当发送电子邮件时，获取服务器的 IP 地址。如果此设置有一个单一的值 (不基于用户改变)，那么我们可以定义它为应用程序范围。
- Tenant: 如果应用程序是多租户的，我们可以定义特定于租户的设置。
- User: 我们可以使用的用户范围的设置为每个用户存储/获取设置的值。

SettingScopes 枚举具有Flags属性，所以我们可以定义一个具有多个作用域的设置。



设置范围是分层的。例如，如果我们定义设置范围为"Application | Tenant | User"并尝试获取当前设置的值；

我们获取特定用户的值，如果它定义了 (重写) User;如果没有定义User，我们获取特定的租户值，如果它定义了 (重写) Tenant。如果也没有，我们获取应用的值，如果它定义了Application。如果还是没有，我们得到的默认值。

默认值可以是 null 或空字符串。如果可以，建议为设置提供一个默认值。

## 2.5.4 获取设置值

定义设置后，我们可以在服务器和客户端获取到它的当前值。

### 1. 服务器端 (Server side)

ISettingManager 用于执行设置操作。我们可以在应用程序中任何地方注入和使用它。ISettingManager 定义了很多获取设置值方法。

最常用的方法是 GetSettingValue (或GetSettingValueAsync 为异步调用)。它将返回当前设置的基于默认值、应用程序、租户和用户设置范围的值(如设置范围之前的一段中所述)。例子：

```
//Getting a boolean value (async call)
var value1 = await SettingManager.GetSettingValueAsync<bool>("PassiveUsersCanNotLogin");
//Getting a string value (sync call)
var value2 = SettingManager.GetSettingValue("SmtpServerAddress");
```

GetSettingValue 有泛型和异步版本，如上所示。也有方法来获取特定的租户或用户的设置值或所有设置值的列表。

由于ISettingManager使用广泛，一些特定的基类 (如 ApplicationService、DomainService 和 AbpController) 有一个名为 SettingManager的属性。如果我们从这些类继承，就无需显式地注入它。

### 2. 客户端(Client side)

如果定义设置时将 IsVisibleToClients 设置为 true，就可以在客户端使用 javascript得到它的当前值。abp.setting 命名空间定义所需的函数和对象。示例：

```
var currentColor = abp.setting.get("SiteColorPreference");
```

也有 getInt 和 getBoolean 这样的方法。你可以使用 abp.setting.values 对象获取所有值。请注意，如果你在服务器端更改设置，客户端不会知道这种变化，除非刷新页面或者以某种方式重新加载页面或者通过代码手动更新。

## 3. 更改设置

ISettingManager定义了ChangeSettingForApplicationAsync, ChangeSettingForTenantAsync 和 ChangeSettingForUserAsync 方法（以及同步版本）来更改应用程序，租户和用户分别的设置。

## 4. 关于缓存

缓存在服务器端设置管理，所以，我们不应直接使用存储库或数据库更新语句改变设置的值。

（2.1、2.2由半冷翻译，2.3由天道翻译，2.4由李伟翻译）## 2.6 ABP公共结构 - 时区设置

### 2.6.1 简介

虽然有些应用的需求是单时区，然而另一些是需要对不同的时区进行处理的。为了满足这样的需求以及对时间的处理。ABP提供了处理时间操作的通用基础设施。

### 2.6.2 Clock

Clock 这个类是主要用来处理 DateTime 的值。它具有以下静态属性和方法：

- **Now**：根据当前设置的提供器来获取当前时间
- **Kind**：取得当前提供器的 DateTimeKind
- **SupportsMultipleTimezone**：取得一个值用来判断该应用当前所使用的提供器是否支持多时区转换(只有 ClockProviders.Utc才支持多时区之间的转换)
- **Normalize** 对给定的时间使用当前提供器来进行转换

所以我们不应该使用 `DateTime.Now`，而是使用 `Clock.Now`，如下所示：

```
DateTime now = Clock.Now;
```

ABP中内置了3种Provider，如下所示：

- **ClockProviders.Unspecified** (UnspecifiedClockProvider): 这是默认的provider并且它的表现行为就像 `DateTime.Now`。
- **ClockProviders.Utc** (UtcClockProvider): 它使用UTC时间，`Clock.Now` 等效于 `DateTime.UtcNow`。Normalize方法会将给定的时间转换为UTC时间并且设置它的Kind为 `DateTimeKind.Utc`。它支持多时区操作。
- **ClockProviders.Local** (LocalClockProvider): 程序宿主的计算机时间。Normalize方法会将给定的时间转换为本地时间并且设置它的Kind为 `DateTimeKind.Local`。

为了支持多时区转换，你可以设置Clock.Provider为：

```
Clock.Provider = ClockProviders.Utc;
```

对于上面的设置，我们通常是在应用程序主入口就设置好了。例如：`main`函数，web应用的`Application_Start`函数。

### 2.6.3 Client Side

我们可以在客户端脚本中使用 `abp.clock`，当你在服务器端设置好 `Clock.Provider`，ABP 会自动的在客户端设置好 `abp.clockprovider`。ABP创建了一个脚本对象：`abp.timing.timeZoneInfo` 它包含了当前用户所设置的时区信息。这个信息包含了Windows和IANA时区的id和一些额外的windows时区信息，详细请查看源码 [TimingScriptManager](#) 的 `GetUsersTimezoneScriptsAsync`函数。使用这些信息可以将服务器的UTC时间转换为客户端需要显示的时间。

注意：在客户端进行时间转换，首先你得设置你的应用默认为Utc，并且每个用户可以设置自己的时区，这个默认是使用 `SettingManager`来设置的。然后你可以使用moment的timezone插件将服务器端时间转换为本地时间。首先全局设置：`moment.tz.setDefault(abp.timing.timeZoneInfo.iana.timeZoneId)`；然后你通过动态API，或者WebAPI取得JSON后，将JSON中的时间如此转换：`moment(item.creationTime).format('LLL')`，或者可以这样：`abp.timing.convertToUserTimezone(dateTime).format()`；前提是你需要使用`abp.moment.js`。作者的文档写的也不是很好，这是我开发过程中结合源码补充的。如果要为每个用户设置不同的时区，最好是将时区信息保存到用户表，登录的时候保存到Claim中。那么在MVC中转换的时候我们就可以用到 `TimezoneHelper.ConvertFromUtc`。详细可以见提问：<https://github.com/aspnetboilerplate/aspnetboilerplate/issues/1320>。

如果使用用户表来保存每个用户的时区，最好是自定义一个AbpSession：

```
public interface ICustomAbpSession : IAbpSession
{
    string TimezoneId { get; }

    string ImpersonatorTimezoneId { get; }

    string GetUsersTimezoneScript();
}

public class CustomAbpSession : ClaimsAbpSession, ICustomAbpSession
{
    public CustomAbpSession(IMultiTenancyConfig multiTenancy) : base(multiTenancy)
    {
    }

    public virtual string TimezoneId
    {
        get
        {
            var timezoneIdClaim = PrincipalAccessor.Principal?.Claims.FirstOrDefault(c => c.Type == CustomAbpClaimNames.TimezoneId);
            return string.IsNullOrEmpty(timezoneIdClaim?.Value) ? null : timezoneIdClaim.Value;
        }
    }

    public virtual string ImpersonatorTimezoneId
    {
        get
        {
            var impersonatorTimezoneIdClaim = PrincipalAccessor.Principal?.Claims.FirstOrDefault(c => c.Type == CustomAbpClaimNames.ImpersonatorTimezoneId);
            return string.IsNullOrEmpty(impersonatorTimezoneIdClaim?.Value) ? null : impersonatorTimezoneIdClaim.Value;
        }
    }
}
```

```

    }

    //使用这个代码来重置 TimingScriptManager.cs 的GetUsersTimezoneScriptsAsync函数取得的脚本。
    //当然需要在 _Layout.cshtml 中调用这个方法, 如果不想这样写, 也可以继承ITimingScriptManager接口, 重新实现它。
    public virtual string GetUsersTimezoneScript()
    {
        if (!Clock.SupportsMultipleTimezone)
            return string.Empty;

        var timezoneId = TimezoneId;
        var timezone = TimeZoneInfo.FindSystemTimeZoneById(timezoneId);

        var timezoneInfo = " {" +
            "         windows: {" +
            "             timeZoneId: '" + timezoneId + "'," +
            "             baseUtcOffsetInMilliseconds: '" + timezone.BaseUtcOffset.TotalMillisec
            "             currentUtcOffsetInMilliseconds: '" + timezone.GetUtcOffset(Clock.Now).
            "             isDaylightSavingTimeNow: '" + timezone.IsDaylightSavingTime(Clock.Now)
            "         }," +
            "         iana: {" +
            "             timeZoneId: '" + TimezoneHelper.WindowsToIana(timezoneId) + "'" +
            "         }," +
            "     }";

        return " abp.timing.timeZoneInfo = " + timezoneInfo;
    }
}

```

## 2.6.4 时区

ABP定义了一个 **Abp.Timing.TimeZone** 配置名, 用来存储Host, Tenant和用户所选择的时区。ABP假定设定的时区是一个有效的 **Windows timezone id**。ABP也定义了一个时区映射文件, 将Windows时区转换为 **IANA** 时区, 这是因为有些通用库所使用的是 IANA timezone id。为了支持多时区, 必须使用 **UtcClockProvider**。如果使用 **UtcClockProvider**, 那么所有的时间值将会以UTC时间进行存储, 并且以UTC时间发送到客户端。那么在客户端我们能将UTC时间转换为客户所设置的时区。

注意: 我遇到过这样的问题, 在Windows Server 2012 如果系统的时区是协调世界时(Coordinated Universal Time)。并且默认用户时区是系统时区时, `TimeZoneInfo.FindSystemTimeZoneById(timezoneId)`, ABP中取得IANA时区会报异常。

## 2.6.5 Binders and Converters

- ABP能自动normalize来自如MVC, WebAPI以及ASP.NET Core应用的时间, 这是基于当前的ClockProvider的。
- ABP能基于当前的ClockProvider自动normalize来自数据库的时间, 当 **EntityFramework** 以及 **NHibernate** 模块被使用的时候。

如果 **UtcClockProvider** 被使用, 所有的时间都会作为UTC时间存储在数据库。并且所有来自客户端的时间都会被当做UTC时间来接收除非被明确的指定为其他时区。

## 3.1 ABP领域层 - 实体

实体是 DDD（领域驱动设计）的核心概念之一。Eric Evans 是这样描述的“很多对象不是通过它们的属性定义的, 而是通过一连串连续性事件和标识定义的”（引用领域驱动设计一书）。

译者注: 对象不是通过它们的属性来下根本性的定义, 而应该是通过它的线性连续性和标识性定义的。所以, 实体是具有唯一标识的ID且存储在数据库中。实体通常被映射成数据库中的一个表。

### 3.1.1 实体类

在 ABP 中, 实体继承自 **Entity** 类, 请看下面示例:

```

public class Person : Entity
{
    public virtual string Name { get; set; }
    public virtual DateTime CreationTime { get; set; }
    public Task()
    {
        CreationTime = DateTime.Now;
    }
}

```

我们定义一个实体类 `Person`，并且为它定义两个属性。父类 `Entity` 具有主键属性 `Id`。所有继承 `Entity` 类的子类都将具有主键为 `Id` 的属性。

`Id` 数据类型可以被更改。默认是 `int` 类型。如果你想给 `Id` 定义其它类型，你应该像下面示例一样来指定 `Id` 的类型。

```
public class Person : Entity<long>
{
    public virtual string Name { get; set; }
    public virtual DateTime CreationTime { get; set; }
    public Task()
    {
        CreationTime = DateTime.Now;
    }
}
```

你可以设置为 `string`，`Guid` 或者其它你想要的数据类型。实体类重写了 `equality (==)` 操作符用来判断两个实体对象是否相等（主要是判断两个实体的 `Id` 主键 是否相等）。还定义了一个 `IsTransient()` 方法来检测当前 `Id` 的值是否与指定的类型的缺省值相等。

### 3.1.2 接口约定

在多数应用程序中，实体一般都具有像 `CreationTime` 的属性，用来指示该实体是什么时候被创建的。APB 提供了一些有用的接口来实现这些类似的功能。（通俗的说只要实现指定的接口就能实现指定的功能）。

#### 1. 审计（Auditing）

实现 `IHasCreationTime` 接口。当该实体被插入到数据库时，APB 会自动设置该属性的值为当前时间。

```
public interface IHasCreationTime
{
    DateTime CreationTime { get; set; }
}
```

我们可以给 `Person` 类实现 `IHasCreationTime` 接口：

```
public class Person : Entity<long>, IHasCreationTime
{
    public virtual string Name { get; set; }
    public virtual DateTime CreationTime { get; set; }
    public Task()
    {
        CreationTime = DateTime.Now;
    }
}
```

`ICreationAudited` 扩展自 `IHasCreationTime` 并且该接口具有属性 `CreatorUserId`：

```
public interface ICreationAudited : IHasCreationTime
{
    long? CreatorUserId { get; set; }
}
```

当保存一个新的实体时，APB 会自动设置 `CreatorUserId` 的属性值为当前用户的 `Id`。你可以很容易的实现 `ICreationAudited` 接口，通过派生自实体类 `CreationAuditedEntity`。它有一个实现不同 `Id` 主键 数据类型的泛型版本。

下面是一个为实现类似修改功能的接口

```
public interface IModificationAudited
{
    DateTime? LastModificationTime { get; set; }
    long? LastModifierUserId { get; set; }
}
```

当更新一个实体时，APB 会自动设置这些属性的值。你只需要在你的实体类里面实现这些属性。如果你想实现所有的审计属性，你可以直接扩展 `IAudited` 接口；示例如下：

```
public interface IAudited : ICreationAudited, IModificationAudited
{
}
}
```

作为一个快速开发方式，你可以直接派生自 `AuditedEntity` 类，不需要再去实现 `IAudited` 接口，`AuditedEntity` 类有一个实现不同 ID 数据类型的泛型版本(默认是 `int`)。

## 2. 软删除(Soft delete)

软删除是一个通用的模式，它标记一个实体已经被删除了，而不是实际从数据库中删除记录。例如：你可能不想从数据库中硬删除一条用户记录，因为它被许多其它的表所关联。为了实现软删除的目的我们可以实现该接口 `ISoftDelete`：

```
public interface ISoftDelete
{
    bool IsDeleted { get; set; }
}
```

ABP 实现了开箱即用的软删除模式。当一个实现了软删除的实体正在被删除，ABP 会察觉到这个动作，并且阻止其真正删除，设置 `IsDeleted` 属性值为 `true` 并且更新数据库中的实体。也就是说，被软删除的记录不可以从数据库中检索出，ABP 会为我们自动过滤软删除的记录。（例如：`Select` 查询，这里指通过 ABP 查询，不是通过数据库中的查询分析器查询。）

如果你用了软删除，你有可能也想实现这个功能，就是记录谁删除了这个实体。要实现该功能你可以实现 `IDeletionAudited` 接口，请看下面示例：

```
public interface IDeletionAudited : ISoftDelete
{
    long? DeleterUserId { get; set; }
    DateTime? DeletionTime { get; set; }
}
```

正如你所看到的 `IDeletionAudited` 扩展自 `ISoftDelete` 接口。当一个实体被删除的时候 ABP 会自动的为这些属性设置值。如果你想为实体类扩展所有的审计接口（例如：创建（`creation`），修改（`modification`）和删除（`deletion`）），你可以直接实现 `IFullAudited` 接口，因为该接口已经继承了这些接口。请看下面示例：

```
public interface IFullAudited : IAudited, IDeletionAudited
{
}
}
```

作为一个快速开发方式，你可以直接从 `FullAuditedEntity` 类派生你的实体类，因为该类已经实现了 `IFullAudited` 接口。

为了导航定义属性到你的 `User` 实体，所有的审计接口和类都有一个泛型模板（例如：`ICreationAudited`和 `FullAuditedEntity`），这里的 `TUser` 指的是进行创建，修改和删除的用户的实体类的类型，详细请看源代码（`Abp.Domain.Entities.Auditing` 空间下的 `FullAuditedEntity` 类），`TPrimaryKey` 指的是 `Entity` 基类 `Id` 类型，默认是 `int`。

## 3. 激活状态/闲置状态(Active/Passive)

有些实体需要被标记为激活状态或者闲置状态。那么你可以为实体采取 `active/passive` 状态的方式来实现。基于这个原因而创建的实体，你可以扩展 `IPassivable` 接口来实现该功能。该接口定义了 `IsActive` 的属性。

如果你首次创建的实体被标记为激活状态，你可以在构造函数设置 `IsActive` 属性值为 `true`。这不同于软删除（`IsDeleted`）。如果实体被软删除，它不能从数据库中被检索到（ABP 已经过滤了软删除记录）。但是对于激活状态/闲置状态的实体，这完全取决于你怎样去获取这些被标记了的实体。

### 3.1.3 IEntity 接口

事实上 `Entity` 实现了 `IEntity` 接口（`Entity` 实现了 `IEntity` 接口）。如果你不想从 `Entity` 类派生，你能直接的实现这些接口。其他实体类也可以实现相应的接口。但是不建议你用这种方式。除非你有一个很好的理由不从 `Entity` 类派生。

## 3.2 ABP领域层 - 仓储

仓储定义：“在领域层和数据映射层的中介,使用类似集合的接口来存取领域对象”(Martin Fowler)。

实际上,仓储被用于领域对象在数据库上的操作(实体Entity和值对象Value types)。一般来说,我们针对不同的实体(或聚合根Aggregate Root)会创建相对应的仓储。

### 3.2.1 IRepository接口

在ABP中,仓储类要实现IRepository接口。最好的方式是针对不同仓储对象定义各自不同的接口。

针对Person实体的仓储接口声明的示例如下所示:

```
public interface IPersonRepository : IRepository<Person> { }
```

IPersonRepository继承自IRepository,用来定义Id的类型为int(Int32)的实体。如果你的实体Id数据类型不是int,你可以继承IRepository接口,如下所示:

```
public interface IPersonRepository : IRepository<Person, long> { }
```

对于仓储类, IRepository定义了许多泛型的方法。比如: Select,Insert,Update,Delete方法(CRUD操作)。在大多数的时候,这些方法已足已应付一般实体的需要。如果这些方对于实体来说已足够,我们便不需要再去创建这个实体所需的仓储接口/类。在Implementation章节有更多细节。

#### 1. 查询(Query)

IRepository定义了从数据库中检索实体的常用方法。

取得单一实体(Getting single entity)

```
TEntity Get(TPrimaryKey id);
Task<TEntity> GetAsync(TPrimaryKey id);
TEntity Single(Expression<Func<TEntity, bool>> predicate);
TEntity FirstOrDefault(TPrimaryKey id);
Task<TEntity> FirstOrDefaultAsync(TPrimaryKey id);
TEntity FirstOrDefault(Expression<Func<TEntity, bool>> predicate);
Task<TEntity> FirstOrDefaultAsync(Expression<Func<TEntity, bool>> predicate);
TEntity Load(TPrimaryKey id);
```

Get方法被用于根据主键值(Id)取得对应的实体。当数据库中根据主键值找不到相符合的实体时,它会抛出异常。Single方法类似Get方法,但是它的输入参数是一个表达式而不是主键值(Id)。因此,我们可以写Lambda表达式来取得实体。示例如下:

```
var person = _personRepository.Get(42);
var person = _personRepository.Single(p => o.Name == "Halil ibrahim Kalkan");
```

注意,Single方法会在给出的条件找不到实体或符合的实体超过一个以上时,都会抛出异常。

FirstOrDefault也一样,但是当没有符合Lambda表达式或Id的实体时,会返回null(取代抛出异常)。当有超过一个以上的实体符合条件,它只会返回第一个实体。

Load并不会从数据库中检索实体,但它会创建延迟执行所需的代理对象。如果你只使用Id属性,实际上并不会检索实体,它只有在你存取想要查询实体的某个属性时才会从数据库中查询实体。当有性能需求的时候,这个方法可以用来替代Get方法。Load方法在NHibernate与ABP的整合中也有实现。如果ORM提供者(Provider)没有实现这个方法,Load方法运行的会和Get方法一样。

ABP有些方法具有异步(Async)版本,可以应用在异步开发模型上(见Async方法相关章节)。

取得实体列表(Getting list of entities)

```
List<TEntity> GetAllList();
Task<List<TEntity>> GetAllListAsync();
List<TEntity> GetAllList(Expression<Func<TEntity, bool>> predicate);
Task<List<TEntity>> GetAllListAsync(Expression<Func<TEntity, bool>> predicate);
IQueryable<TEntity> GetAll();
```

GetAllList被用于从数据库中检索所有实体。重载并且提供过滤实体的功能,如下:

```
var allPeople = _personRepository.GetAllList();
var somePeople = _personRepository.GetAllList(person => person.IsActive && person.Age > 42);
```



GetAll返回IQueryable类型的对象。因此我们可以在调用完这个方法之后进行Linq操作。示例:

```
var query = from person in _personRepository.GetAll()
where person.IsActive
orderby person.Name
select person;
var people = query.ToList();
List<Person> personList2 = _personRepository.GetAll().Where(p => p.Name.Contains("H")).OrderBy(p => p.Name).Skip(40).
```

如果调用GetAll方法,那么几乎所有查询都可以使用Linq完成。甚至可以用它来编写Join表达式。

说明: 关于IQueryable 当你调用GetAll这个方法在Repository对象以外的地方,必定会开启数据库连接。这是因为IQueryable允许延迟执行。它会直到你调用ToList方法或在foreach循环上(或是一些存取已查询的对象方法)使用IQueryable时,才会实际执行数据库的查询。因此,当你调用ToList方法时,数据库连接必需是启用状态。我们可以使用ABP所提供的UnitOfWork特性在调用的方法上来实现。注意,Application Service方法预设都已经是UnitOfWork。因此,使用了GetAll方法就不需要如同Application Service的方法上添加UnitOfWork特性。

有些方法拥有异步版本,可应用在异步开发模型(见关于async方法章节)。

自定义返回值(Custom return value)

ABP也有一个额外的方法来实现IQueryable的延迟加载效果,而不需要在调用的方法上添加UnitOfWork这个属性卷标。

```
T Query<T>(Func<IQueryable<Tentity>,T> queryMethod);
```

查询方法接受Lambda(或一个方法)来接收IQueryable并且返回任何对象类型。示例如下:

```
var people = _personRepository.Query(q => q.Where(p => p.Name.Contains("H")).OrderBy(p => p.Name).ToList());
```

因为是采用Lambda(或方法)在仓储对象的方法中执行,它会在数据库连接开启之后才被执行。你可以返回实体集合,或一个实体,或一个具部份字段(注: 非Select \*)或其它执行查询后的查询结果集。

## 2. 新增(insert)

IRepository接口定义了简单的方法来提供新增一个实体到数据库:

```
TEntity Insert(TEntity entity);
Task<TEntity> InsertAsync(TEntity entity);
TPriamaryKey InsertAndGetId(TEntity entity);
Task<TPriamaryKey> InsertAndGetIdAsync(TEntity entity);
TEntity InsertOrUpdate(TEntity entity);
Task<TEntity> InsertOrUpdateAsync(TEntity entity);
TPriamaryKey InsertOrUpdateAndGetId(TEntity entity);
Task<TPriamaryKey> InsertOrUpdateAndGetIdAsync(TEntity entity);
```

新增方法会新增实体到数据库并且返回相同的已新增实体。InsertAndGetId方法返回新增实体的标识符(Id)。当我们采用自动递增标识符值且需要取得实体的新产生标识符值时非常好用。InsertOrUpdate会新增或更新实体,选择那一种是根据Id是否有值来决定。最后,InsertOrUpdateAndGetId会在实体被新增或更新后返回Id值。

所有的方法都拥有异步版本可应用在异步开发模型(见关于异步方法章节)

## 3. 更新(UPDATE)

IRepository定义一个方法来实现更新一个已存在于数据库中的实体。它更新实体并返回相同的实体对象。

```
TEntity Update(TEntity entity);
Task<TEntity> UpdateAsync(TEntity entity);
```

## 4. 删除(Delete)

IRepository定了一些方法来删除已存在数据库中实体。

```
void Delete(TEntity entity);
Task DeleteAsync(TEntity entity);
```



```
void Delete(TPrimaryKey id);
Task DeleteAsync(TPrimaryKey id);
void Delete(Expression<Func<TEntity, bool>> predicate);
Task DeleteAsync(Expression<Func<TEntity, bool>> predicate);
```

第一个方法接受一个现存的实体,第二个方法接受现存实体的Id。

最后一个方法接受一个条件来删除符合条件的实体。要注意,所有符合predicate表达式的实体会先被检索而后删除。因此,使用上要很小心,这是有可能造成许多问题,假如果有太多实体符合条件。

所有的方法都拥有async版本来应用在异步开发模型(见关于异步方法章节)。

## 5. 其它方法(others)

IRepository也提供一些方法来取得数据表中实体的数量。

```
int Count();
Task<int> CountAsync();
int Count(Expression<Func<TEntity, bool>> predicate);
Task<int> CountAsync(Expression<Func<TEntity, bool>> predicate);
Long LongCount();
Task<long> LongCountAsync();
Long LongCount(Expression<Func<TEntity, bool>> predicate);
Task<long> LongCountAsync(Expression<TEntity, bool>> predicate);
```

所有的方法都拥有async版本被应用在异步开发模型(见关于异步方法章节)。

## 6. 关于异步方法(Async methods)

ABP支持异步开发模型。因此,仓储方法拥有Async版本。在这里有一个使用异步模型的application service方法的示例:

```
public class PersonAppService : AbpWpfDemoAppServiceBase, IPersonAppService
{
    private readonly IRepository<Person> _personRepository;
    public PersonAppService(IRepository<Person> personRepository)
    {
        _personRepository = personRepository;
    }

    public async Task<GetPersonOutput> GetAllPeople()
    {
        var people = await _personRepository.GetAllListAsync();
        return new GetPeopleOutput {
            People = Mapper.Map<List<PersonDto>>(people);
        };
    }
}
```

GetAllPeople方法是异步的并且使用GetAllListAsync与await关键字。Async不是在每个ORM框架都有提供。上例是从EF所提供的异步能力。如果ORM框架没有提供Async的仓储方法那么它会以同步的方式操作。同样地,举例来说,InsertAsync操作起来和EF的新增是一样的,因为EF会直到单元作业(unit of work)完成之后才会写入新实体到数据库中(DbContext.SaveChanges)。

### 3.2.2 仓储的实现

ABP在设计上是采取不指定特定ORM框架或其它存取数据库技术的方式。只要实现IRepository接口,任何框架都可以使用。

仓储要使用NHibernate或EF来实现都很简单。见实现这些框架在ABP仓储对象上一文:

- NHibernate
- EntityFramework

当你使用NHibernate或EntityFramework,如果提供的方法已足够使用,你就不需要为你的实体创建仓储对象了。我们可以直接注入IRepository(或IRepository)。下面的示例为application service使用仓储对象来新增实体到数据库:

```
public class PersonAppService : IPersonAppService
{
    private readonly IRepository<Person> _personRepository;

    public PersonAppService(IRepository<Person> personRepository) {
```

```
        _personRepository = personRepository;
    }

    public void CreatePerson(CreatePersonInput input) {
        person = new Person { Name = input.Name, EmailAddress = input.EmailAddress; };
        _personRepository.Insert(person);
    }
}
```

PersonAppService的建构子注入了IRepository并且使用其Insert方法。当你有需要为实体创建一个定制的仓储方法,那么你就应该创建一个仓储类给指定的实体。

### 3.2.3 管理数据库连接

数据库连接的开启和关闭,在仓储方法中,ABP会自动化的进行连接管理。

当仓储方法被调用后,数据库连接会自动开启且启动事务。当仓储方法执行结束并且返回以后,所有的实体变化都会被储存,事务被提交并且数据库连接被关闭,一切都由ABP自动化的控制。如果仓储方法抛出任何类型的异常,事务会自动地回滚并且数据连接会被关闭。上述所有操作在实现了IRepository接口的仓储类所有公开的方法中都可以被调用。

如果仓储方法调用其它仓储方法(即便是不同仓储的方法),它们共享同一个连接和事务。连接会由仓储方法调用链最上层的那个仓储方法所管理。更多关于数据库管理,详见UnitOfWork文件。

### 3.2.4 仓储的生命周期

所有的仓储对象都是暂时性的。这就是说,它们是在有需要的时候才会被创建。ABP大量的使用依赖注入,当仓储类需要被注入的时候,新的类实体会由注入容器会自动地创建。见相依赖注入文件有更多信息。

### 3.2.5 仓储的最佳实践

- 对于一个T类型的实体,是可以使用IRepository。但别任何情况下都创建定制化的仓储,除非我们真的很需要。预定义仓储方法已经足够应付各种案例。
- 假如你正创建定制的仓储(可以实现IRepository)
- 仓储类应该是无状态的。这意味着,你不该定义仓储等级的状态对象并且仓储方法的调用也不应该影响到其它调用。
- 当仓储可以使用相依赖注入,尽可能较少或是不相根据于其它服务。

## 3.3 ABP领域层 - 领域服务

### 3.3.1 简介

领域服务(或者服务,在DDD模式中)是被用来执行领域操作或者业务规则的。Eric Evans 在他的DDD书中这样说过:一个好的Service应该有以下三个特征:

1. 与领域概念相关的操作不是Entity或Value Object 的一个自然部分;
2. 接口是根据领域模型的其它元素定义的;
3. 操作是无状态的。

领域服务和Application Services 是不同的, Application Services 返回的是DTO, 而领域服务返回的是领域对象(实体或者值类型)。

领域服务可以被应用服务和其它的领域服务调用,但是不可以被表现层直接调用(表现层可以直接调用应用服务)。

### 3.3.2 IDomainService 和 DomainService

ABP 定义了一个 IDomainService 接口,所有的领域服务都必须实现该接口(记住这是一个约定),一旦实现了这个接口,那么领域服务就会通过Dependency Injection 自动的注册到系统中作为一个暂时对象(Transient)。

领域服务也可以继承自 DomainService 类(这是可选的)。因此,它可以用一些继承而来的属性来做日志记录,本地化等等;即使你不继承该类,如果你需要这些属性也是可以注入的。

### 3.3.3 实例

假设我们有一个任务管理系统,并且我们有这样的业务规则,把任务分配到个人。

#### 1. 创建一个接口

首先，我们为此服务定义一个接口（不是必须的，但是一个好的实践）：

```
public interface ITaskManager : IDomainService
{
    void AssignTaskToPerson(Task task, Person person);
}
```

正如你所看到的，TaskManager 用到了领域对象：Task 和 Person。这里有一些领域服务的命名约定；例如：TaskManager, TaskService 或者 TaskDomainService 等等。

## 2. 实现服务

实现如下：

```
public class TaskManager : DomainService, ITaskManager
{
    public const int MaxActiveTaskCountForAPerson = 3;

    private readonly ITaskRepository _taskRepository;

    public TaskManager(ITaskRepository taskRepository)
    {
        _taskRepository = taskRepository;
    }

    public void AssignTaskToPerson(Task task, Person person)
    {
        if (task.AssignedPersonId == person.Id)
        {
            return;
        }

        if (task.State != TaskState.Active)
        {
            throw new ApplicationException("Can not assign a task to a person when task is not active!");
        }

        if (HasPersonMaximumAssignedTask(person))
        {
            throw new UserFriendlyException(L("MaxPersonTaskLimitMessage", person.Name));
        }

        task.AssignedPersonId = person.Id;
    }

    private bool HasPersonMaximumAssignedTask(Person person)
    {
        var assignedTaskCount = _taskRepository.Count(t => t.State == TaskState.Active && t.AssignedPersonId == person.Id);
        return assignedTaskCount >= MaxActiveTaskCountForAPerson;
    }
}
```

我们有如下两个业务规则： 1. 分配给Person的任务状态应该是Active状态 2. Person最多只能接受3个任务

你可能感到奇怪，为啥我在做第一次检测的时候为什么我抛出了一个ApplicationException异常，第二次检测时抛出 UserFriendlyException 异常。这个和领域服务没有半毛钱关系；我这样做仅仅是为了提供一个示例。这完全取决于你。我认为用户界面必须获取到任务状态和分配数量错误时的错误消息。并且我认为这是一个应用级的错误，我们可以不向用户展示这个难以理解的错误，我们应该向用户展示一个可读性好的错误消息。这仅仅是一个示例。

### 3.3.4 应用层调用领域服务

下面示例为我们展示了应用层是如何调用TaskManager:

```
public class TaskAppService : ApplicationService, ITaskAppService
{
    private readonly IRepository<Task, long> _taskRepository;
    private readonly IRepository<Person> _personRepository;
    private readonly ITaskManager _taskManager;

    public TaskAppService(IRepository<Task, long> taskRepository, IRepository<Person> personRepository, ITaskManager
```

```
{
    _taskRepository = taskRepository;
    _personRepository = personRepository;
    _taskManager = taskManager;
}

public void AssignTaskToPerson(AssignTaskToPersonInput input)
{
    var task = _taskRepository.Get(input.TaskId);
    var person = _personRepository.Get(input.PersonId);

    _taskManager.AssignTaskToPerson(task, person);
}
}
```

任务服务层用给定的DTO和仓储资源去检索相关的Task和Person，并且将检索到的结果传递给TaskManager（领域服务）。

### 3.3.5 探讨

基于上面的示例，你可能有一些疑问。

#### 1. 为什么不只在应用层实现这些逻辑？

你可能会说为什么不在服务层来实现领域服务里面的业务逻辑。

我们可以简单的说因为这根本不是应用层的任务。因为它不是一个use-case(用例)，而是一个业务操作。我们可以用同样（分配任务给用户）的逻辑在不同的用例中。我们可能会有另外的应用场景，以某种方式更新任务并且这个更新可能包含了分配任务给另外的人。所以，我们可以在这里用相同的领域逻辑。（说白了就是业务规则重用）还有就是，我们可以有2中不同的UI（手持设备应用和Web应用）可以共享相同的领域。

如果你的业务领域相对简单，那么你可以不考虑使用领域服务来实现这些逻辑。在DDD模式中这不是一个最佳实践，但ABP不会强迫你使用这种设计模式。

#### 2. 为什么一定要使用领域服务？

看如下示例：

```
public void AssignTaskToPerson(AssignTaskToPersonInput input)
{
    var task = _taskRepository.Get(input.TaskId);

    task.AssignedPersonId = input.PersonId;
}
```

写这个应用开发人员可能不知道这里是一个TaskManager，并直接给任务的AssignedPersonId 分配了 PersonId。那么，怎么阻止这个的发生呢？在DDD社区有很多关于应该采用那种设计模式的探讨。我们不会做深入的探讨。但是我们会用一个简单的方式来实现。

我们可以改变Task实体，如下所示：

```
public class Task : Entity<long>
{
    public virtual int? AssignedPersonId { get; protected set; }

    //...other members and codes of Task entity

    public void AssignToPerson(Person person, ITaskPolicy taskPolicy)
    {
        taskPolicy.CheckIfCanAssignTaskToPerson(this, person);

        AssignedPersonId = person.Id;
    }
}
```

我们给属性AssignedPersonId 的set设置为protected。所以，这个属性不可以被外部类修改。添加一个AssignToPerson方法，该方法接受参数类型Person和ITaskPolicy。ITaskPolicy 接口有一个CheckIfCanAssignTaskToPerson 方法来验证任务是否能分配给Person，如果验证不通过将会抛出一个适当的异常。那么应用层的方法将会如下所示：

```
public void AssignTaskToPerson(AssignTaskToPersonInput input)
{
    var task = _taskRepository.Get(input.TaskId);
    var person = _personRepository.Get(input.PersonId);

    task.AssignToPerson(person, _taskPolicy);
}
```

现在, 没有第二种方式将任务分配给个人。我们应该总是使用AssignToPerson 并且不可以跳过该业务规则。

## 3.4 ABP领域层 - 工作单元

### 3.4.1 通用连接和事务管理方法

连接和事务管理是使用数据库的应用程序最重要的概念之一。当你开启一个数据库连接,什么时候开始事务,如何释放连接...诸如此类的。

正如大家都知道的,.Net使用连接池（connection pooling）。因此,创建一个连接实际上是从连接池中取得一个连接,会这么做是因为创建新连接会有成本。如果没有任何连接存在于连接池中,一个新的连接对象会被创建并且添加到连接池中。当你释放连接,它实际上是将这个连接对象送回到连接池。这并不是实际意义上的释放。这个机制是由.Net所提供的。因此,我们应该在使用完之后释放掉连接对象。这就是最佳实践。

在应用程序中, 有两个通用的方式来创建/释放一个数据库连接:

第一个方法:在Web请求到达的时候, 创建一个连接对象。(Application\_BeginRequest这个位于global.asax中的事件),使用同一个连接对象来处理所有的数据库操作, 并且在请求结束的时候关闭/释放这个连接 (Application\_EndRequest事件)。

这是个简易但却没效率的方法,原因:

- 或许这个Web请求不需要操作数据库,但是连接却会开启。这对于连接池来说是个毫无效率的使用方式。
- 这可能会让Web请求的运行时间变长,并且数据库操作还会需要一些执行。这也是一种没效率的连接池使用方式。
- 这对于Web应用来说是可行的。如果你的应用程序是Windows Service,这可能就无法被实现了。

同样的这是一个使用事务式的数据库操作最佳场景。如果有一个操作发生失败,所有的操作都会回滚。因为事务会锁住数据库中的一些数据列(事件数据表),它必定要是短暂的。

第二个方法: 创建一个连接当需要的时候(只要在使用它之前)并且释放它在使用它之后。这是相当高效的,但是就得乏味而且反复的去进行（创建/释放连接）。

### 3.4.2 ABP的连接和事务管理

ABP综合上述两个连接管理的方法, 并且提供一个简单而且高效的模型。

#### 1. 仓储类(Repository classes)

仓储是主要的数据库操作的类。ABP开启了一个数据库连接并且在进入到仓储方法时会启用一个事务。因此,你可以安全地使用连接于仓储方法中。在仓储方法结束后,事务会被提交并且会释放掉连接。假如仓储方法抛出任何异常,事务会被回滚并且释放掉连接。在这个模式中,仓储方法是单元性的(一个工作单元unit of work)。ABP在处理上述那些动作都是全自动的。在这里,有一个简单的仓储:

```
public class ContentRepository : NhRepositoryBase<Content>, IContentRepository {
    public List<Content> GetActiveContents(string searchCondition) {
        var query = from content in Session.Query<Content>()
                    where content.IsActive && !content.IsDeleted
                    select content;

        if(string.IsNullOrEmpty(searchCondition)) {
            query = query.Where(content => content.Text.Contains(searchCondition));
        }

        return query.ToList();
    }
}
```

这个示例使用NHibernate作为ORM框架。如上所示,不需要撰写任何数据库连接操作(NHibernate中的Session)的程序代码。

假如仓储方法调用另一个仓储方法(一般来说,若工作单元方法调用另一个工作单元的方法),都使用同一个连接和事务。第一个被调用到的仓储方法负责管理连接和事务,而其余被它调用的仓储方法则只单纯使用不管理。

## 2. 应用服务(Application service classes)

一个应用服务的方法也被考虑使用工作单元。如果我们拥有一个应用服务方法如下:

```
public class PersonAppService : IPersonAppService {
    private readonly IPersonRepository _personRepository;
    private readonly IStatisticsRepository _statisticsRepository;

    public PersonAppService(IPersonRepository personRepository, IStatisticsRepository statisticsRepository) {
        _personRepository = personRepository;
        _statisticsRepository = statisticsRepository;
    }

    public void CreatePerson(CreatePersonInput input) {
        var person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };
        _personRepository.Insert(person);
        _statisticsRepository.IncrementPeopleCount();
    }
}
```

在CreatePerson方法中,我们新增一个person使用person仓储并且使用statistics仓储增加总people数量。两个仓储共享同一个连接和事务于这个例子中,因为这是一个应用服务的方法。ABP开启一个数据库连接并且开启一个事务于进入到CreationPerson这个方法,若没有任何异常抛出,接着提交这个事务于方法结尾时,若有异常被抛出,则会回滚这个事务。在这种机制下,所有数据库的操作在CreatePerson中,都成了单元性的了(工作单元)。

## 3. 工作单元(Unit of work)

工作单元在后台替仓储和应用服务的方法工作。假如你想要控制数据库的连接和事务,你就需要直接操作工作单元。下面有两个直接使用的示例:

首要且最好的使用UnitOfWorkAttribute的方式如下:

```
[UnitOfWork]
public void CreatePerson(CreatePersonInput input) {
    var person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };
    _personRepository.Insert(person);
    _statisticsRepository.IncrementPeopleCount();
}
```

因此,CreatePerson方法转变成工作单元并且管理数据库连接和事务,两个仓储对象都使用相同的工作单元。要注意,假如这是应用服务的方法则不需要添加UnitOfWork属性, 见工作单元方法:第三章,3.3.5。

第二个示例是使用IUnitOfWorkManager.Begin(...)方法如下所示:

```
public class MyService {
    private readonly IUnitOfWorkManager _unitOfWorkManager;
    private readonly IPersonRepository _personRepository;
    private readonly IStatisticsRepository _statisticsRepository;

    public MyService(IUnitOfWorkManager unitOfWorkManager, IPersonRepository personRepository, IStatisticsRepository
        _unitOfWorkManager = unitOfWorkManager;
        _personRepository = personRepository;
        _statisticsRepository = statisticsRepository;
    }

    public void CreatePerson(CreatePersonInput input) {
        var person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };
        using(var unitOfWork = _unitOfWorkManager.Begin()) {
            _personRepository.Insert(person);
            _statisticsRepository.IncrementPeopleCount();
            unitOfWork.Complete();
        }
    }
}
```



你可以注入并且使用`IUnitOfWorkManager`，如上所示。因此,你可以创建更多的有限范围（limited scope）的工作单元。在这个机制中,你通常可以手动调用`Complete`方法。如果你不调用,事务会回滚并且所有的异常都不会被储存。`Begin`方法被重写从而设置工作单元的选项。

这很棒,不过除非你有很好的理由,否则还是少用`UnitOfWork`属性。

### 3.4.3 工作单元

#### 1. 禁用工作单元(Disabling unit of work)

你或许会想要禁用应用服务方法的工作单元(因为它默认是启用的)。要想做到这个,使用`UnitOfWorkAttribute`的`IsDisabled`属性。示例如下:

```
[UnitOfWork(IsDisabled = true)]
public virtual void RemoveFriendship(RemoveFriendInput input) {
    _friendshipRepository.Delete(input.Id);
}
```

平常时, 你不会需要这么做,这是因为应用服务的方法都应该是单元性且通常是使用数据库。在有些情况下,你或许会想要禁用应用服务的工作单元:

- 你的方法不需要任何数据库操作且你不想要开启那些不需要的数据库连接
- 你想要使用工作单元于`UnitOfWorkScope`类的有限范围内,如上所述

注意,如果工作单元方法调用这个`RemoveFriendship`方法,禁用被忽略且它和调用它的方法使用同一个工作单元。因此,使用禁用这个功能要很小心。同样地,上述程序代码工作的很好,因为仓储方法默认即为工作单元。

#### 2. 无事务的工作单元(Non-transactional unit of work)

工作单元默认上是具事务性的(这是它的天性)。因此,ABP启动/提交/回滚一个显性的数据库等级的事务。在有些特殊案例中,事务可能会导致问题,因为它可能会锁住有些数据列或是数据表于数据库中。在此这些情境下, 你或许会想要禁用数据库等级的事务。`UnitOfWork`属性可以从它的建构子中取得一个布尔值来让它如非事务型工作单元般工作着。示例为:

```
[UnitOfWork(false)]
public GetTasksOutput GetTasks(GetTasksInput input) {
    var tasks = _taskRepository.GetAllWithPeople(input.AssignedPersonId, input.State);
    return new GetTasksOutput {
        Tasks = Mapper.Map<List<TaskDto>>(tasks)
    };
}
```

建议可以这么做`[UnitOfWork(isTransaction:false)]`。(具有可读性并且明确)。

注意,ORM框架(像是NHibernate和EntityFramework)会在单一命令中于内部进行数据储存。假设你更新了一些的实体于非事务的UoW。即便于这个情境下所有的更新都会于单一数据库命令的工作单元尾部完成。但是,如果你直接执行SQL查询,它会立即被执行。

这里有一个非事务性UoW的限制。如果你已经位于事务性UoW区域内,设定`isTransactional`为`false`这个动作会被忽略。

使用非事务性UoW要小心,因为在大多数的情况下,数据整合应该是具事务性的。如果你的方法只是读取数据,不改变数据,那么当然可以采用非事务性。

#### 3. 工作单元调用其它工作单元(A unit of work method calls another)

若工作单元方法(一个贴上`UnitOfWork`属性标签的方法)调用另一个工作单元方法,他们共享同一个连接和事务。第一个方法管理连接,其它的方法只是使用它。这在所有方法都执行在同一个线程下是可行的(或是在同一个Web请求内)。实际上,当工作单元区域开始,所有的程序代码都会在同一线程中执行并共享同一个连接事务,直到工作单元区域终止。这对于使用`UnitOfWork`属性和`UnitOfWorkScope`类来说都是一样的。如果你创建了一个不同的线程/任务,它使用自己所属的工作单元。

#### 4. 自动化的saving changes (Automatically saving changes)

当我们使用工作单元到方法上,ABP自动的储存所有变化于方法的末端。假设我们需要一个可更新person名称的方法:

```
[UnitOfWork]
public void UpdateName(UpdateNameInput input) {
    var person = _personRepository.Get(input.PersonId);
```



```
    person.Name = input.NewName;
}
```

就这样,名称就被修改了!我们甚至没有调用 `_personRepository.Update` 方法。ORM框架会持续追踪实体所有的变化于工作单元内,且反映所有变化到数据库中。

注意,这不需要在应用服务声明 `UnitOfWork`,因为它们默认就是采用工作单元。

## 5. 仓储接口的 `GetAll()` 方法 ( `IRepository.GetAll() method`)

当你在仓储方法外调用 `GetAll` 方法,这必定得有一个开启状态的数据库连接,因为它返回 `IQueryable` 类型的对象。这是需要的,因为 `IQueryable` 延迟执行。它并不会马上执行数据库查询,直到你调用 `ToList()` 方法或在 `foreach` 循环中使用 `IQueryable` (或是存取被查询结果集的情况下)。因此,当你调用 `ToList()` 方法,数据库连接必需是启用状态。示例:

```
[UnitOfWork]
public SearchPeopleOutput SearchPeople(SearchPeopleInput input) {
    //取得 IQueryable<Person>
    var query = _personRepository.GetAll();

    //若有选取,则添加一些过滤条件
    if(!string.IsNullOrEmpty(input.SearchedName)) {
        query = query.Where(person => person.Name.StartsWith(input.SearchedName));
    }

    if(input.IsActive.HasValue) {
        query = query.Where(person => person.IsActive == input.IsActive.Value);
    }

    //取得分页结果集
    var people = query.Skip(input.SkipCount).Take(input.MaxResultCount).ToList();

    return new SearchPeopleOutput { People = Mapper.Map<List<PersonDto>>(people) };
}
```

在这里, `SearchPeople` 方法必需是工作单元,因为 `IQueryable` 在被调用 `ToList()` 方法于方法本体内,并且数据库连接必须于 `IQueryable.ToList()` 被执行时开启。

一如 `GetAll()` 方法,如果需要数据库连接且没有仓储的情况下,你就必须要使用工作单元。注意,应用服务方法默认就是工作单元。

工作单元属性的限制 (`UnitOfWork attribute restrictions`)

在下面情境下你可以使用 `UnitOfWork` 属性标签:

- 类所有 `public` 或 `public virtual` 这些基于界面的方法 (像是应用服务是基于服务界面)
- 自我注入类的 `public virtual` 方法 (像是 MVC Controller 和 Web API Controller)
- 所有 `protected virtual` 方法。

建议将方法标示为 `virtual`。你无法应用在 `private` 方法上。因为, ABP 使用 `dynamic proxy` 来实现,而私有方法就无法使用继承的方法来实现。当你不使用依赖注入且自行初始化类,那么 `UnitOfWork` 属性 (以及任何代理) 就无法正常运作。

## 3.4.5 选项

有许多可以用来控制工作单元的选项。

首先,我们可以在 `startup configuration` 中改变所有工作单元的所有默认值。这通常是用了我们模块中的 `PreInitialize` 方法来实现。

```
public class SimpleTaskSystemCoreModule : AbpModule {
    public override void PreInitialize() {
        Configuration.UnitOfWork.IsolationLevel = IsolationLevel.ReadCommitted;
        Configuration.UnitOfWork.Timeout = TimeSpan.FromMinutes(30);
    }

    //...其它模块方法
}
```

## 3.4.6 方法

工作单元系统运作是无缝且不可视的。但是,在有些特例下,你需要调用它的方法。

## SaveChanges

ABP储存所有的变化于工作单元的尾端,你不需要做任何事情。但是,有些时候,你或许会想要在工作单元的过程中就储存所有变化。在这个案例中,你可以注入IUnitOfWorkManager并且调用IUnitOfWorkManager.Current.SaveChanges()方法。示例中以Entity Framework在储存变化时取得新增实体的Id。注意,当前工作单元是具事务性的,所有在事务中的变化会在异常发生时都被回滚,即便是已调用SaveChange。

### 3.4.7 事件

工作单元具有Completed/Failed/Disposed事件。你可以注册这些事件并且进行所需的操作。注入IUnitOfWorkManager并且使用IUnitOfWorkManager.Current 属性来取得当前已激活的工作单元并且注册它的事件。

你或许会想要执行有些程序代码于当前工作单元成功地完成。示例:

```
public void CreateTask(CreateTaskInput input) {
    var task = new Task { Description = input.Description };
    if(input.AssignedPersonId.HasValue) {
        task.AssignedPersonId = input.AssignedPersonId.Value;
        _unitOfWorkManager.Current.Completed += (sender, args) => { // };
    }
    _taskRepository.Insert(task);
}
```

## 3.5 ABP领域层 - 领域事件

在C#中,一个类可以定义其专属的事件并且其它类可以注册该事件并监听,当事件被触发时可以获得事件通知。这对于桌面应用程序或独立的Windows Service来说非常有用。但是,对于Web应用程序来说会有点问题,因为对象是根据请求(request)被创建并且它们的生命周期都很短暂。我们很难注册其它类别的事件。同样地,直接注册其它类别的事件也造成了类之间的耦合性。

在应用系统中,领域事件被用于解耦并且重用(re-use)商业逻辑。

### 3.5.1 事件总线

事件总线为一个单体(singleton)的对象,它由所有其它类所共享,可通过它触发和处理事件。要使用这个事件总线,你需要引用它。你可以用两种方式来实现:

#### 1. 获取默认实例( Getting the default instance)

你可以直接使用EventBus.Default。它是全局事件总线并且可以如下方式使用:

```
EventBus.Default.Trigger(...); //触发事件
```

#### 2. 注入IEventBus事件接口(Injecting IEventBus)

除了直接使用EventBus.Default外,你还可以使用依赖注入(DI)的方式来取得IEventBus的参考。这利于进行单元测试。在这里,我们使用属性注入的范式:

```
public class TaskAppService : ApplicaService {
    public IEventBus EventBus { get; set; }
    public TaskAppService() {
        EventBus = NullEventBus.Instance;
    }
}
```

注入事件总线,采用属性注入比建构子注入更适合。事件是由类所描述并且该事件对象继承自EventData。假设我们想要触发某个事件于某个任务完成后:

```
public class TaskCompletedEventData : EventData {
    public int TaskId { get; set; }
}
```

这个类所包含的属性都是类在处理事件时所需要的。EventData类定义了EventSource(那个对象触发了这个事件)和EventTime(何时触发)属性。

### 3.5.2 定义事件

ABP定义AbpHandledExceptionData事件并且在异常发生的时候自动地触发这个事件。这在你想要取得更多关于异常的信息时特别有用(即便ABP已自动地纪录所有的异常)。你可以注册这个事件并且设定它的触发时机是在异常发生的时候。

ABP也提供在实体变更方面许多的通用事件数据类型: EntityCreatedEventData, EntityUpdatedEventData和EntityDeletedEventData。它们被定义在Abp.Events.Bus.Entities命名空间中。当某个实体新增/更新/删除后,这些事件会由ABP自动地触发。如果你有一个Person实体,可以注册到EntityCreatedEventData,事件会在新的Person实体创建且插入到数据库后被触发。这些事件也支持继承。如果Student类继承自Person类,并且你注册到EntityCreatedEventData中,接着你将会在Person或Student新增后收到触发。

### 3.5.3 触发事件

触发事件的范例如下:

```
public class TaskAppService : ApplicationService {
    public IEventBus EventBus { get; set; }
    public TaskAppService() {
        EventBus = NullEventBus.Instance;
    }

    public void CompleteTask(CompleteTaskInput input) {
        //TODO: 已完成数据库上的任务
        EventBus.Trigger(new TaskCompletedEventData { TaskId = 42 });
    }
}
```

这里有一些触发方法的重载:

```
EventBus.Trigger<TaskCompletedEventData>(new TaskCompletedEventData { TaskId = 42});
EventBus.Trigger(this, new TaskCompletedEventData { TaskId = 42 });
EventBus.Trigger(typeof(TaskCompletedEventData), this, new TaskCompletedEventData { TaskId = 42});
```

### 3.5.4 事件处理

要进行事件的处理,你应该要实现IEventHandler接口如下所示:

```
public class ActivityWriter : IEventHandler<TaskCompletedEventData>, ITransientDependency {
    public void HandleEvent(TaskCompletedEventData eventData) {
        WriteActivity("A task is completed by id = " + eventData.TaskId);
    }
}
```

EventBus已集成到依赖注入系统中。就如同我们在上例中实现ITransientDependency那样,当TaskCompleted事件触发,它会创建一个新的ActivityWriter类的实体并且调用它的HandleEvent方法,并接着释放它。详情请见依赖注入(DI)一文。

#### 1. 基础事件的处理(Handling base events)

EventBus支持事件的继承。举例来说,你可以创建TaskEventData以及两个继承类:TaskCompletedEventData和TaskCreatedEventData:

```
public class TaskEventData : EventData {
    public Task Task { get; set; }
}

public class TaskCreatedEventData : TaskEventData {
    public User CreatorUser { get; set; }
}

public class TaskCompletedEventData : TaskEventData {
    public User CompletorUser { get; set; }
}
```

然而,你可以实现IEventHandler来处理这两个事件:

```
public class ActivityWriter : IEventHandler<TaskEventData>, ITransientDependency {
    public void HandleEvent(TaskEventData eventData) {
        if(eventData is TaskCreatedEventData) {
            ...
        }else{
            ...
        }
    }
}
```

当然,你也可以实现IEventHandler来处理所有的事件,如果你真的想要这样做的话(译者注:作者不太建议这种方式)。

## 2. 处理多个事件(Handling multiple events)

在单个处理器(handler)中我们可以处理多个事件。此时,你应该针对不同事件实现IEventHandler。范例如下:

```
public class ActivityWriter :
    IEventHandler<TaskCompletedEventData>,
    IEventHandler<TaskCreatedEventData>,
    ITransientDependency
{
    public void HandleEvent(TaskCompletedEventData eventData) {
        //TODO: 处理事件
    }
    public void HandleEvent(TaskCreatedEventData eventData) {
        //TODO: 处理事件
    }
}
```

### 3.5.5 注册处理器

我们必需注册处理器(handler)到事件总线中来处理事件。

#### 1. 自动型Automatically

ABP扫描所有实现IEventHandler接口的类,并且自动注册它们到事件总线中。当事件发生,它通过依赖注入(DI)来取得处理器(handler)的引用对象并且在事件处理完毕之后将其释放。这是比较建议的事件总线使用方式于ABP中。

#### 2. 手动型(Manually)

也可以通过手动注册事件的方式,但是会有些问题。在Web应用程序中,事件的注册应该要在应用程序启动的时候。当一个Web请求(request)抵达时进行事件的注册,并且反复这个行为。这可能会导致你的应用程序发生一些问题,因为注册的类可以被调用多次。同样需要注意的是,手动注册无法与依赖注入系统一起使用。

ABP提供了多个事件总线注册方法的重载(overload)。最简单的一个重载方法是等待委派(delegate)或Lambda。

```
EventBus.Register<TaskCompletedEventData>(eventData =>
{
    WriteActivity("A task is completed by id = " + eventData.TaskId);
});
```

因此,事件:task completed会发生,而这个Lambda方法会被调用。第二个重载方法等待的是一个对象,该对象实现了IEventHandler:

```
Eventbus.Register<TaskCompletedEventData>(new ActivityWriter());
```

相同的例子,如果ActivityWriter因事件而被调用。这个方法也有一个非泛型的重载。另一个重载接受两个泛化的参数:

```
EventBus.Register<TaskCompletedEventData, ActivityWriter>();
```

此时,事件总线创建一个新的ActivityWriter于每个事件。当它释放的时候,它会调用ActivityWriter.Dispose方法。

最后,你可以注册一个事件处理器工厂(event handler factory)来负责创建处理器。处理器工厂有两个方法: GetHandler和 ReleaseHandler,范例如下:

```
public class ActivityWriterFactory : IEventHandlerFactory {
    public IEventHandler GetHandler() {
        return new ActivityWriter();
    }
    public void ReleaseHandler(IEventHandler handler) {
        //TODO: 释放ActivityWriter实体(处理器)
    }
}
```

ABP也提供了特殊的工厂类,IocHandlerFactory,通过依赖注入系统, IocHandlerFactory可以用来创建或者释放(dispose)处理器。ABP可以自动化注册IocHandlerFactory。因此,如果你想要使用依赖注入系统,请直接使用自动化注册的方式。

### 3.5.6 取消注册事件

当你手动注册事件总线,你或许想要在以后取消注册。最简单的取消事件注册的方式即为registration.Dispose()。举例如下:

```
//注册一个事件
var registration = EventBus.Register<TaskCompletedEventData>(eventData => WriteActivity("A task is completed by id = " + eventData.Id));
//取消注册一个事件
registration.Dispose();
```

当然,取消注册可以在任何地方任何时候进行。保存(keep)好注册的对象并且在你想要取消注册的时候释放(dispose)掉它。所有注册方法的重载(overload)都会返回一个可释放(disposable)的对象来取消事件的注册。

事件总线也提供取消注册方法。使用范例:

```
//创建一个处理器
var handler = new ActivityWriter();
//注册一个事件
EventBus.Register<TaskCompletedEventData>(handler);
//取消这个事件的注册
EventBus.Unregister<TaskCompletedEventData>(handler);
```

它也提供重载的方法给取消注册的委派和工厂。取消注册处理器对象必须与之前注册的对象是同一个。

最后,EventBus提供一个UnregisterAll()方法来取消某个事件所有处理器的注册,而UnregisterAll()方法则是所有事件的所有处理器。

(3.1由Carl翻译, 3.2-3.5由台湾-小张翻译) ## 3.6 ABP领域层 - 数据过滤器

### 3.6.1 介绍

在数据库开发中,我们一般会运用软删除(soft-delete)模式,即不直接从数据库删除数据,而是标记这笔数据为已删除。因此,如果实体被软删除了,那么它就应该不会在应用程序中被检索到。要达到这种效果,我们需要在每次检索实体的查询语句上添加SQL的Where条件IsDeleted = false。这是个乏味的工作,但它是个容易被忘掉的事情。因此,我们应该要有个自动的机制来处理这些问题。

ABP提供数据过滤器(Data filters),它使用自动化的,基于规则的过滤查询。ABP已经有一些预定义的过滤器,当然也可以自行创建你专属的过滤器。

注意: 只针对EntityFramework:ABP数据过滤器仅实现在EntityFramework。还无法在其它ORM工具中使用。见其它ORM章节于本文末端。

### 3.6.2 预定义过滤器

#### 1. 软删除接口(ISoftDelete)

软删除过滤器(Soft-delete filter)会过滤从数据库查询出来的实体且是自动套用(从结果集中提取出来)。如果实体需要被软删除,它需要实现ISoftDelete接口,该接口仅定义了一个IsDeleted属性。例:

```
public class Person : Entity, ISoftDelete {
    public virtual string Name { get; set; }
    public virtual bool IsDeleted { get; set; }
}
```

Person实体实际上并没有从数据库中被删除,当删除此实体时,IsDeleted属性值会被设定为true。当你使用IRepository.Delete方法时,ABP会自动完成这些工作(你可以手动设定IsDeleted为true,但是Delete方法更加自然且是较建议的方式)。

当实现了ISoftDelete之后,当你已经从数据库中取得了People列表,已被删除的People实体并不会被检索到。在这里有一个示例类,该类使用了person仓储来取得所有的People实体:

```
public class MyService {  
    private readonly IRepository<Person> _personRepository;  
  
    public MyService(IRepository<Person> personRepository) {  
        _personRepository = personRepository;  
    }  
  
    public List<Person> GetPeople() {  
        return _personRepository.GetAllList();  
    }  
}
```

GetPeople方法仅取得Person实体,该实体其IsDeleted = false(非删除状态)。所有的仓储方法以及导航属性都能够正常运作。我们可以添加一些其它的Where条件,Join...等等。它将会自动地添加IsDeleted=false条件到生成的SQL查询语句中。

注意: 何时启用?ISoftDelete过滤器总是启用,除非你直接禁用它。

提醒:如果你实现了IDeletionAudited接口(该接口继承自ISoftDelete),删除创建时间和被删除的用户Id, 这些都会由ABP进行自动的处理。

## 2. 多租接口(IMustHaveTenant)

如果你创建一个多租户的应用程序(储存所有租户的数据于单一一个数据库中),你肯定不会希望某个租户看到其它租户的资料。你可以实现IMustHaveTenant接口于此案例中,示例如下:

```
public class Product : IMustHaveTenant {  
    public virtual int TenantId { get; set; }  
  
    public virtual string Name { get; set; }  
}
```

IMustHaveTenant定义了TenantId来区别不同的租户实体。ABP使用IAbpSession来取得当前TenantId并且自动地替当前租户进行过滤查询的处理。

注意: 何时启用?IMustHaveTenant默认是启用的。如果当前使用并没有登入到系统或是当前用户是一个管理级使用者(管理级使用者即为一个最高权限的使用者,它可以管理所有租户和租户的资料),ABP会自动地禁用IMustHaveTenant过滤器。因此,所有的租户的数据都可以被应用程序所检索。注意,这与安全性无关,你应该要对敏感数据进行验证授权处理。

## 3. 多租接口(IMayHaveTenant)

如果一个实体类由多个租户(tenant)以及管理级使用者(host)所共享(这意味着该实体对象或许由租户(tenant)或是管理级使用者(host)所掌控),你可以使用IMayHaveTenantfilter。IMayHaveTenant接口定义了TenantId但是它是可空类(nullable)。

```
public class Product : IMayHaveTenant {  
    public virtual int? TenantId { get; set; }  
  
    public virtual string Name { get; set; }  
}
```

当为null值,则代表这是一个管理级使用者(host)所掌控的实体,若为非null值,则代表这个实体是由租户(tenant)所掌控,而其Id值即为TenantId。ABP使用IAbpSession接口来取得当前TenantId。IMayHaveTenant过滤器并不如IMustHaveTenant普遍常用。但是当作为管理级使用者(host)和租户(tenant)所需要的通用结构使用时,你或许会需要用到它。

何时启用?IMayHaveTenant接口总是启用的,除非你直接禁用它。

### 3.6.3 禁用过滤器

可以在工作单元(unit of work)中调用DisableFilter方法来禁用某个过滤器,如下所示:

```
var people1 = _personRepository.GetAllList();  
using(_unitOfWorkManager.Current.DisableFilter(AbpDataFilters.SoftDelete)) {
```



```
var people2 = _personRepository.GetAllList();
}

var people3 = _personRepository.GetAllList();
```

DisableFilter方法取得一或多个过滤器名称,且类型皆为string。AbpDataFilters.SoftDelete是一个常数字符串其包含了ABP标准的软删除过滤器。

people2亦可取得已标记为删除的People实体,而people1和people3将会是唯一的非已标记为删除的People实体。若配合使用using语法,你可以禁用其控制范围内(Scope)的过滤器。如果你不使用 using 语法,此过滤器会被一直禁用,直到工作单元(unit of work)结束或者再度启用它。(意思是:如果你使用"using"关键字声明,过滤器是启用状态;当前工作单元(unit of work)结束后,过滤器是禁止状态。如果不使用"using"关键字声明,默认过滤器是禁用状态,此时可以手动启用过滤器。)

你可以注入IUnitOfWorkManager并且在上述示例中使用。同样的,你可以使用CurrentUnitOfWork属性作为一个在应用服务中的简便方式(它是从ApplicationService类继承而来的)。

注意: 关于using语法:假如过滤器在你调用DisableFilter方法并配合using语法之前已是启用,则过滤器会被禁用,并且会自动地在using语法结束后再度启用。但是若过滤器在using语法之前就已经被禁用了,DisableFilter方法实际上并不做任何式,并且过滤器会维持禁用状态即便是using语法的结束后。

### 3.6.4 启用过滤器

你可以在工作单元(unit of work)中使用EnableFilter方法启用过滤器,如同DisableFilter方法一般(两者互为正反两面)。EnableFilter亦会返回disposable来自动地重新禁用过滤器。

### 3.6.5 设定过滤器参数

过滤器可以被参数化(parametric)。IMustHaveTenant过滤器是这类过滤器的一个范本,因为当前租户(tenant)的Id是在执行时期决定的。对于这些过滤器,如果真有需要,我们可以改变过滤器的值。举例如下:

```
CurrentUnitOfWork.SetFilterParameter("PersonFilter", "personId", 42);
```

另一个示例如下:设定IMayHaveTenant过滤器的tenantId值:

```
CurrentUnitOfWork.SetfilterParameter(AbpDataFilters.MayHaveTenant, AbpDataFilters.Parameters.TenantId, 42);
```

### 3.6.6 自定义过滤器

欲创建定制的过滤器并且整合到ABP中,首先我们需要定义一个接口,该接口将会由使用这个过滤器的实体所实现。假设我们想要自动化地依PersonId进行过滤,示例如下:

```
public interface IHasPerson {
    int PersonId { get; set; }
}
```

然后我们就可以实现这个接口在我们的实体上了,示例如下:

```
public class Phone : Entity, IHasPerson {
    [ForeignKey("PersonId")]
    public virtual Person Person { get; set; }

    public virtual int PersonId { get; set; }

    public virtual string Number { get; set; }
}
```

因为ABP使用EntityFramework.DynamicFilters这个过滤器,我们使用它的规则(rule)来定义过滤器。在我们的DbContext类中,我们重写了OnModelCreating并且定义了过滤器如下示例所示:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder) {
    base.OnModelCreating(modelBuilder);
    modelBuilder.Filter("PersonFilter", (IHasPerson entity, int personId) => entity.PersonId == personId, 0 );
}
```

PersonFilter过滤器在这里是一个唯一的过滤器名称。再来就是过滤器接口的参数定义和personId过滤器参数(不一定需要,假如过滤器是属于不可参数化(parametric)型),最后一个参数为personId的默认值。

最后一个步骤,我们需要注册这个过滤器到ABP工作单元(unit of work)系统中,设定的位置在我们模块里的PreInitialize方法。

```
Configuration.UnitOfWork.RegisterFilter("PersonFilter", false);
```

第一个参数是我们刚刚所定义的唯一名称,第二个参数指示这个过滤器预设是启用还是禁用。在声明完这些可参数化(parametric)的过滤器后,我们可以在执行期间指定它的值来操作这个过滤器。

```
using(CurrentUnitOfWork.EnableFilter("PersonFilter")) {  
    CurrentUnitOfWork.SetFilterParameter("PersonFilter", "personId", 42);  
    var phone = _phoneRepository.GetAllList();  
    // ...  
}
```

我们可以从有些数据源中取得personId而不需要写死在程序代码中。上述示例是为了要能够程序化过滤器。过滤器可拥有0到更多的参数。假如是无参数的过滤器,它就不需要设定过滤器的值。同样地,假如它预设是启用,就不需要手动启用(当然的,我们也可以禁用它)。

EntityFramework.DynamicFilters的文件:若需要更多关于动态数据过滤器的相关信息,可以见其在git上的文件  
<https://github.com/jcachat/EntityFramework.DynamicFilters>

我们可以为安全性创建一个定制化的过滤器,主/被动实体,多租户...诸如此类的。

### 3.6.7 其它对象关系映射工具

ABP数据过滤器仅实现在Entity Framework上。对于其它ORM工具则尚不可用。但是,实际上,你可以模仿这个模式到其它使用仓储来取得数据的案例下。这此案例中,你可以创建一个定制的仓储并且覆写GetAll方法,如果有需要的话,可以一起修改其它资料检索方法。

## 4.1 ABP应用层 - 应用服务

应用服务用于将领域(业务)逻辑暴露给展现层。展现层通过传入DTO(数据传输对象)参数来调用应用服务,而应用服务通过领域对象来执行相应的业务逻辑并且将DTO返回给展现层。因此,展现层和领域层将被完全隔离开来。在一个理想的层级项目中,展现层应该从不直接访问领域对象。

### 4.1.1 IApplicationService接口

在ABP中,一个应用服务需要实现IApplicationService接口。最好的实践是针对每个应用服务都创建相应的接口。所以,我们首先定义一个应用服务接口,如下所示:

```
public interface IPersonAppService : IApplicationService  
{  
    void CreatePerson(CreatePersonInput input);  
}
```

IPersonAppService只有一个方法,它将被展现层调用来创建一个新的Person。CreatePersonInput是一个DTO对象,如下所示:

```
public class CreatePersonInput : IInputDto  
{  
    [Required]  
    public string Name { get; set; }  
  
    public string EmailAddress { get; set; }  
}
```

接着,我们实现IPersonAppService接口:

```
public class PersonAppService : IPersonAppService  
{  
    private readonly IRepository<Person> _personRepository;  
  
    public PersonAppService(IRepository<Person> personRepository)
```

```

    {
        _personRepository = personRepository;
    }

    public void CreatePerson(CreatePersonInput input)
    {
        var person = _personRepository.FirstOrDefault(p => p.EmailAddress == input.EmailAddress);
        if (person != null)
        {
            throw new UserFriendlyException("There is already a person with given email address");
        }

        person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };
        _personRepository.Insert(person);
    }
}

```

以下是几个重要提示:

- PersonAppService通过IRepository来执行数据库操作。它通过构造器注入模式来生成。我们在这里使用了依赖注入。
- PersonAppService实现了IApplicationService（通过IPersonAppService继承IApplicationService）。ABP会自动地把它注册到依赖注入系统中，并可以注入到别的类型中使用。
- CreatePerson方法需要一个CreatePersonInput类型的参数。这是一个作为输入的DTO，它将被ABP自动验证其数据有效性。可以查看DTO和数据有效性验证(Validation)文档获取相关细节。

#### 4.1.2 应用服务类型

应用服务(Application Services)需要实现IApplicationService接口。当然，你可以选择将你的应用服务(Application Services)继承自ApplicationService基类，这样你的应用服务也就自然而然的实现IApplicationService接口了。ApplicationService基类提供了方便的日志记录和本地化功能。在此建议你针对你的应用程序创建一个应用服务基类继承自ApplicationService类型。这样你就可以添加一些公共的功能来提供给你的所有应用服务使用。一个应用服务示例如下所示：

```

public class TaskAppService : ApplicationService, ITaskAppService
{
    public TaskAppService()
    {
        LocalizationSourceName = "SimpleTaskSystem";
    }

    public void CreateTask(CreateTaskInput input)
    {
        //记录日志, Logger定义在ApplicationService中
        Logger.Info("Creating a new task with description: " + input.Description);

        //获取本地化文本(L是LocalizationHelper.GetString(...))的简便版本, 定义在 ApplicationService类型)
        var text = L("SampleLocalizableTextKey");

        //TODO: Add new task to database...
    }
}

```

本例中我们在构造函数中定义了LocalizationSourceName，但你可以在基类中定义它，这样你就不需要在每个具体的应用服务中定义它。查看日志记录(logging)和本地化(localization)文档可以获取更多的相关信息。

#### 4.1.3 工作单元

在ABP中，一个应用服务方法默认是一个工作单元。

##### (1) 连接 & 事务管理 (For connection & transaction management)

在应用服务方法中，如果我们需要调用两个仓储方法，那么这些方法必须为一个事务。举个例子：

```

public void CreatePerson(CreatePersonInput input)
{
    var person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };
    _personRepository.Insert(person);
    _statisticsRepository.IncrementPeopleCount();
}

```

我们向Person表插入一个数据，接着在其他表中修改了Person计数字段的值。这两个操作实现于不同的仓储中，但是它们使用了相同的数据连接和事务。这是怎么实现的呢？

对于UOW模式，当事务启动并且开始执行CreatePerson方法的时候，ABP会自动地打开数据库。在方法结束时，如果未发生异常该事务将会被提交，并确保关闭数据库连接。因此，CreatePerson方法中的所有数据库操作将作为一个事务(具有原子性)，当有异常抛出时这些事务中的操作将会回滚。所以，示例中的两个仓储方法使用了相同的数据连接和事务。

当你调用仓储中的GetAll()方法时，它将返回一个IQueryable。数据库连接应会在调用仓储方法后打开。这是因为IQueryable和LINQ的延迟执行。当你调用类似ToList()方法时，数据库查询才会真正的开始执行。来看下面的示例：

```
public SearchPeopleOutput SearchPeople(SearchPeopleInput input)
{
    //获取 IQueryable<Person>
    var query = _personRepository.GetAll();

    //过滤数据
    if (!string.IsNullOrEmpty(input.SearchedName))
    {
        query = query.Where(person => person.Name.StartsWith(input.SearchedName));
    }

    if (input.IsActive.HasValue)
    {
        query = query.Where(person => person.IsActive == input.IsActive.Value);
    }

    //获取分页
    var people = query.Skip(input.SkipCount).Take(input.MaxResultCount).ToList();

    return new SearchPeopleOutput {People = Mapper.Map<List<PersonDto>>(people)};
}
```

由于一个应用服务(Application Services)方法就是一个工作单元，所以数据库连接在方法执行期间都是开启的。如果你在非应用服务(Application Services)中调用GetAll()，你需要显式的使用工作单元模式。如：在Controller的Action方法中要使用GetAll()或调用多个有对数据库操作的AppService方法时，应该将Action方法使用virtual修饰，并在Action的上面通过[UnitOfWork]进行显示开启工作单元模式。

注意我使用了AutoMapper库将List转换成List。可以查看DTO文档获取相关细节。

译者-天道注：这里要说一下，就是uow和非uow模式的区别，两种模式对于数据库连接的打开和关闭是不同的。对于控制器的方法，ABP默认是非uow模式，此时如果调用方法会报错，提示数据库未连接。解决的办法是在方法加上virtual。

## （2）自动保存数据修改 (For automatically saving changes)

对于工作单元方法（应用服务(Application Services)方法），在方法结束时ABP将会自动保存所有数据修改。假设我们需要一个应用服务(Application Services)方法来更新一个Person的Name:

```
public void UpdateName(UpdateNameInput input)
{
    var person = _personRepository.Get(input.PersonId);
    person.Name = input.NewName;
}
```

就是这样，Name被成功修改!我们甚至不需要调用\_personRepository.Update方法。ORM框架在工作单元中会跟踪所有实体修改并将修改更新到数据库中。

### 4.1.4 应用服务的生命周期

所有应用服务(Application Services)实例的生命周期都是暂时的(Transient)。这意味着在每次使用都会创建新的应用服务(Application Services)实例。ABP坚决地使用依赖注入技术。当一个应用服务(Application Services)类型需要被注入时，该应用服务(Application Services)类型的新实例将会被依赖注入容器自动创建。查看依赖注入(Dependency Injection)文档获取更多信息。

## 4.2 ABP应用层 - 数据传输对象

数据传输对象(Data Transfer Objects)用于应用层和展现层的数据传输。

展现层传入数据传输对象(DTO)调用一个应用服务方法,接着应用服务通过领域对象执行一些特定的业务逻辑并且返回DTO给展现层。这样展现层和领域层被完全分离开了。在具有良好分层的应用程序中,展现层不会直接使用领域对象(仓库,实体)。

#### 4.2.1 数据传输对象的作用

为每个应用服务方法创建DTO看起来是一项乏味耗时的工作。但如果你正确使用它们,这将会解救你的项目。为啥呢?

##### (1) 抽象领域层 (Abstraction of domain layer)

在展现层中数据传输对象对领域对象进行了有效的抽象。这样你的层(layers)将被恰当的隔离开来。甚至当你想要完全替换展现层时,你还可以继续使用已经存在的应用层和领域层。反之,你可以重写领域层,修改数据库结构,实体和ORM框架,但并不需要对展现层做任何修改,只要你的应用层没有发生改变。

##### (2) 数据隐藏 (Data hiding)

想象一下,你有一个User实体拥有属性Id, Name, EmailAddress和Password。如果UserAppService的GetAllUsers()方法的返回值类型为List。这样任何人都可以查看所有人的密码,即使你没有将它打印在屏幕上。这不仅仅是安全问题,这还跟数据隐藏有关。应用服务应只返回展现层所需要的,不多不少刚刚好。

##### (3) 序列化 & 惰性加载 (Serialization & lazy load problems)

当你将数据(对象)返回给展现层时,数据有可能会被序列化。举个例子,在一个返回Json的MVC的Action中,你的对象需要被序列化成JSON并发送给客户端。直接返回实体给展现层将有可能出现麻烦。

在真实的项目中,实体会引用其他实体。User实体会引用Role实体。所以,当你序列化User时,Role也将被序列化。而且Role还拥有一个List并且Permission还引用了PermissionGroup等等...你能想象这些对象都将被序列化吗?这很有可能使整个数据库数据意外的被序列化。那么该如何解决呢?将属性标记为不可序列化?不行,因为你不知道属性何时该被序列化何时不该序列化。所以在这种情况下,返回一个可安全序列化,特别定制的数据传输对象是不错的选择哦。

几乎所有的ORM框架都支持惰性加载。只有当你需要加载实体时它才会被加载。比如User类型引用Role类型。当你从数据库获取User时,Role属性并没有被填充。当你第一次读取Role属性时,才会从数据库中加载Role。所以,当你返回这样一个实体给展现层时,很容易引起副作用(从数据库中加载)。如果序列化工具读取实体,它将会递归地读取所有属性,这样你的整个数据库都将会被读取。

在展现层中使用实体还会有更多的问题。最佳的方案就是展现层不应该引用任何包含领域层的程序集。

#### 4.2.2 DTO 约定 & 验证

ABP对数据传输对象提供了强大的支持。它提供了一些相关的(Conventional)类型 & 接口并对DTO命名和使用约定提供了建议。当你像这里一样使用DTO, ABP将会自动化一些任务使你更加轻松。

##### 一个例子 (Example)

让我们来看一个完整的例子。我们相要编写一个应用服务方法根据name来搜索people并返回people列表。Person实体代码如下:

```
public class Person : Entity
{
    public virtual string Name { get; set; }
    public virtual string EmailAddress { get; set; }
    public virtual string Password { get; set; }
}
```

首先,我们定义一个应用服务接口:

```
public interface IPersonAppService : IApplicationService
{
    SearchPeopleOutput SearchPeople(SearchPeopleInput input);
}
```

ABP建议命名input/output对象类似于MethodNameInput/MethodNameOutput,对于每个应用服务方法都需要将Input和Output进行分开定义。甚至你的方法只接收或者返回一个值,也最好创建相应的DTO类型。这样,你的代码才会更具有扩展性,你可以添加更多的属性而不需要更改方法的签名,这并不会破坏现有的客户端应用。

当然,方法返回值有可能是void,之后你添加一个返回值并不会破坏现有的应用。如果你的方法不需要任何参数,那么你不需定义一个Input Dto。但是创建一个Input Dto可能是个更好的方案,因为该方法在将来有可能会需要一个参数。当然是否创建这取决于你。Input和Output DTO类型定义如下:

```

public class SearchPeopleInput : IInputDto
{
    [StringLength(40, MinimumLength = 1)]
    public string SearchedName { get; set; }
}

public class SearchPeopleOutput : IOutputDto
{
    public List<PersonDto> People { get; set; }
}

public class PersonDto : EntityDto
{
    public string Name { get; set; }
    public string EmailAddress { get; set; }
}

```

验证：作为约定，Input DTO实现IInputDto 接口，Output DTO实现IOutputDto接口。当你声明IInputDto参数时，在方法执行前ABP将会自动对其进行有效性验证。这类似于ASP.NET MVC验证机制，但是请注意应用服务并不是一个控制器(Controller)。ABP对其进行拦截并检查输入。查看DTO 验证(DTO Validation)文档获取更多信息。EntityDto是一个简单具有与实体相同的Id属性的简单类型。如果你的实体Id不为int型你可以使用它泛型版本。EntityDto也实现了IDto接口。你可以看到PersonDto并不包含Password属性，因为展现层并不需要它。

跟进一步之前我们先实现IPersonAppService:

```

public class PersonAppService : IPersonAppService
{
    private readonly IPersonRepository _personRepository;

    public PersonAppService(IPersonRepository personRepository)
    {
        _personRepository = personRepository;
    }

    public SearchPeopleOutput SearchPeople(SearchPeopleInput input)
    {
        //获取实体
        var peopleEntityList = _personRepository.GetAllList(person => person.Name.Contains(input.SearchedName));

        //转换成DTO
        var peopleDtoList = peopleEntityList
            .Select(person => new PersonDto
            {
                Id = person.Id,
                Name = person.Name,
                EmailAddress = person.EmailAddress
            }).ToList();

        return new SearchPeopleOutput { People = peopleDtoList };
    }
}

```

我们从数据库获取实体，将实体转换成DTO并返回output。注意我们没有手动检测Input的数据有效性。ABP会自动验证它。ABP甚至会检查Input是否为null，如果为null则会抛出异常。这避免了我们在每个方法中都手动检查数据有效性。

但是你可能不喜欢手动将Person实体转换成PersonDto。这真的是个乏味的工作。Peson实体包含大量属性时更是如此。

### 4.2.3 DTO和实体间的自动映射

还好这里有些工具可以让映射（转换）变得十分简单。AutoMapper就是其中之一。你可以通过nuget把它添加到你的项目中。让我们使用AutoMapper来重写SearchPeople方法:

```

public SearchPeopleOutput SearchPeople(SearchPeopleInput input)
{
    var peopleEntityList = _personRepository.GetAllList(person => person.Name.Contains(input.SearchedName));
    return new SearchPeopleOutput { People = Mapper.Map<List<PersonDto>>(peopleEntityList) };
}

```

这就是全部代码。你可以在实体和DTO中添加更多的属性，但是转换代码依然保持不变。在这之前你只需要做一件事：映射



```
Mapper.CreateMap<Person, PersonDto>();
```

AutoMapper创建了映射的代码。这样，动态映射就不会成为性能问题。真是快速又方便。AutoMapper根据Person实体创建了PersonDto,并根据命名约定来给PersonDto的属性赋值。命名约定是可配置的并且很灵活。你也可以自定义映射和使用更多特性，查看AutoMapper的文档获取更多信息。

使用特性(attributes)和扩展方法来映射 (Mapping using attributes and extension methods)

ABP提供了几种attributes和扩展方法来定义映射。使用它你需要通过nuget将Abp.AutoMapper添加到你的项目中。使用AutoMap特性(attribute)可以有两种方式进行映射，一种是使用AutoMapFrom和AutoMapTo。另一种是使用MapTo扩展方法。定义映射的例子如下：

```
[AutoMap(typeof(MyClass2))] //定义映射（这样有两种方式进行映射）
public class MyClass1
{
    public string TestProp { get; set; }
}

public class MyClass2
{
    public string TestProp { get; set; }
}
```

接着你可以通过MapTo扩展方法来进行映射：

```
var obj1 = new MyClass1 { TestProp = "Test value" };
var obj2 = obj1.MapTo<MyClass2>(); //创建了新的MyClass2对象，并将obj1.TestProp的值赋值给新的MyClass2对象的TestProp属性
上面的代码根据MyClass1创建了新的MyClass2对象。你也可以映射已存在的对象，如下所示：
var obj1 = new MyClass1 { TestProp = "Test value" };
var obj2 = new MyClass2();
obj1.MapTo(obj2); //根据obj1设置obj2的属性
```

#### 4.2.4 辅助接口和类型

ABP还提供了一些辅助接口，定义了常用的标准化属性。

ILimitedResultRequest定义了MaxResultCount属性。所以你可以在你的Input DTO上实现该接口来限制结果集数量。

IPagedResultRequest扩展了ILimitedResultRequest，它添加了SkipCount属性。所以我们在SearchPeopleInput实现该接口用来分页：

```
public class SearchPeopleInput : IInputDto, IPagedResultRequest
{
    [StringLength(40, MinimumLength = 1)]
    public string SearchedName { get; set; }

    public int MaxResultCount { get; set; }
    public int SkipCount { get; set; }
}
```

对于分页请求，你可以将实现IHasTotalCount的Output DTO作为返回结果。标准化属性帮助我们创建可复用的代码和规范。可在Abp.Application.Services.Dto命名空间下查看其他的接口和类型。

### 4.3 ABP应用层 - 数据传输对象验证

应用程序的输入数据首先应该被检验是否有效。在Web应用中，通常进行2次数据检验：包括客户端检验和服务端检验。客户端的检验可以使用户有一个良好的用户体验。首先最好是在客户端检验其表单输入的有效性，并且展示给客户端那些输入是无效的。但是，服务器端的校验是更关键和不可缺失的，不要只做客户端检验而不做服务器端检验。

服务器端的检验通常是在应用层执行，应用层中的方法首先检验数据的有效性，然后才使用这些通过验证的数据。ABP的基础设施提供了自动检验输入数据有效性的方法。

应用层得到一个作为输入的数据传输对象（DTO）。在ABP中有一个IValidate的接口，DTO通过实现这个接口能够被APB自动检验。由于IInputDto扩展自IValidate，所以你可以直接实现IInputDto 接口来检验对数据传输对象（DTO）。

### 4.3.1 使用数据注解

ABP可以使用数据注解来检验数据的有效性。假设我们正在开发一个创建任务的应用服务，并且得到了一个输入，请看下面示例：

```
public class CreateTaskInput : IInputDto
{
    public int? AssignedPersonId { get; set; }

    [Required]
    public string Description { get; set; }
}
```

在这里 Description 属性被标记为 Required。AssignedPersonId 是可选的。在 System.ComponentModel.DataAnnotations 命名空间中，还有很多这样的特性（例如：MaxLength, MinLength, RegularExpression 等等）。

示例如下：

```
public class TaskAppService : ITaskAppService
{
    private readonly ITaskRepository _taskRepository;
    private readonly IPersonRepository _personRepository;

    public TaskAppService(ITaskRepository taskRepository, IPersonRepository personRepository)
    {
        _taskRepository = taskRepository;
        _personRepository = personRepository;
    }

    public void CreateTask(CreateTaskInput input)
    {
        var task = new Task { Description = input.Description };

        if (input.AssignedPersonId.HasValue)
        {
            task.AssignedPerson = _personRepository.Load(input.AssignedPersonId.Value);
        }

        _taskRepository.Insert(task);
    }
}
```

正如你所看到的，这里没有写任何的数据验证代码，因为ABP会自动去检验数据的有效性。ABP也会检验输入数据是否为null。如果为空则会抛出AbpValidationException 异常。所以你不需写检验数据是否为null值的代码。如果有任何属性的输入数据是无效的它也会抛出相同的异常。

这个机制近似于 ASP.NET MVC 的验证功能，注意：这里的应用服务类不是继承自Controller，它是用在Web应用的一个普通类。

### 4.3.2 自定义检验

如果数据注解的方式不能满足你的需求，你可以实现ICustomValidate接口，请看下面示例：

```
public class CreateTaskInput : IInputDto, ICustomValidate
{
    public int? AssignedPersonId { get; set; }

    public bool SendEmailToAssignedPerson { get; set; }

    [Required]
    public string Description { get; set; }

    public void AddValidationErrors(List<ValidationResult> results)
    {
        if (SendEmailToAssignedPerson && (!AssignedPersonId.HasValue || AssignedPersonId.Value <= 0))
        {
            results.Add(new ValidationResult("AssignedPersonId must be set if SendEmailToAssignedPerson is true!")
        }
    }
}
```

ICustomValidate 接口声明了一个可被实现的AddValidationErrors方法。这里我们有一个叫做 SendEmailToAssignedPerson 的属性。如果该属性是真，AssignedPersonId 属性会被检验是否有效，否则该属性可以为空。如果有验证错误，我们必须把这些验证结果添加到结果集合中。

### 4.3.3 设置缺省值

在检验数据有效性后，我们需要执行一个额外的操作来整理DTO参数。ABP定义了一个IShouldNormalize接口，这个接口声明了一个 Normalize的方法。如果你实现了这个接口，在检验数据有效性后，Normalize方法会被调用。假设我们的DTO需要一个排序方向的属性。如果这个Sorting属性没有被提供数据，那么在Normalize我们可以给Sorting设置一个缺省值。

```
public class GetTasksInput : IInputDto, IShouldNormalize
{
    public string Sorting { get; set; }

    public void Normalize()
    {
        if (string.IsNullOrEmpty(Sorting))
        {
            Sorting = "Name ASC";
        }
    }
}
```

## 4.4 ABP应用层 - 权限认证

几乎所有的企业级应用程序都会有不同级别的权限验证。权限验证是用于检查用户是否允许某些指定操作。Abp有基础设施让你来实现权限验证。

注意：关于IPermissionChecker接口

Abp权限系统使用IPermissionChecker去检查授权。同时你可以根据需要实现你自己的方式，在module-zero项目中已经完整实现了。如果IPermissionChecker没有被实现，NullPermissionChecker会被使用于授权所有权限给每个人。

### 4.4.1 定义权限

在使用验证权限前，我们需要为每一个操作定义唯一的权限。Abp的设计是基于模块化，所以不同的模块可以有不同的权限。为了定义权限，一个模块应该创建AuthorizationProvider的派生类。MyAuthorizationProvider继承自AuthorizationProvider，换句话说就是AuthorizationProvider派生出MyAuthorizationProvider。例子如下：

```
public class MyAuthorizationProvider : AuthorizationProvider
{
    public override void SetPermissions(IPermissionDefinitionContext context)
    {
        var administration = context.CreatePermission("Administration");

        var userManagement = administration.CreateChildPermission("Administration.UserManagement");
        userManagement.CreateChildPermission("Administration.UserManagement.CreateUser");

        var roleManagement = administration.CreateChildPermission("Administration.RoleManagement");
    }
}
```

IPermissionDefinitionContext 有方法去获取和创建权限。

一个权限有以下属性：

- **Name:** 系统范围内的唯一名字。把它定义为一个字符串常量是个不错的注意。我们倾向于将"."分割不同的层级，但并不要求这么做。你可以设置你任何喜欢的名字。唯一的规则就是这个名字必须是唯一的。
- **Display Name:** 使用一个本地化的字符串去显示权限到UI。
- **Description:** 和Display Name类似。
- **IsGrantedByDefault:** 此权限是否授权给（已登陆）所有用户，除非显示指定。通常设置为False（默认值）。
- **MultiTenancySides:** 对租户应用程序，一个权限可以基于租户或者主机(原文:host)。这是个枚举标识，因此权限可以应用于不同方面(原文: Both Sides)。
- 一个权限可以有父权限和子权限。当然，这不会影响权限检查，它只是在UI层对权限归类有好处。创建authorizationprovider之后，我们应该在模块的PreInitialize方法对它进行注册。如下：

```
Configuration.Authorization.Providers.Add<MyAuthorizationProvider>()
```

authorizationprovider会自动注册到依赖注入系统中。因此，authorization provider可以注入任何依赖（像是Repository）从而使用其他资源去创建权限定义。

#### 4.4.2 检查权限

##### （1）使用AbpAuthorize特性(Using AbpAuthorize attribute)

AbpAuthorize（AbpMvcAuthorize 对应 MVC Controllers and AbpApiAuthorize 对应 Web API Controllers）特性是最简单和常用的方法去检查权限。请考虑如下application service方法：

```
[AbpAuthorize("Administration.UserManagement.CreateUser")]
public void CreateUser(CreateUserInput input)
{
    //A user can not execute this method if he is not granted for "Administration.UserManagement.CreateUser" perm
}
```

没有获得“Administration.UserManagement.CreateUser”权限的用户不能够调用CreateUser。

AbpAuthorize 特性也检查当前用户是否登录 (使用 IAbpSession.UserId)。因此，如果我们将某个方法声明为AbpAuthorize 特性，它至少会检查用户是否登录。代码如下： [AbpAuthorize]

```
public void SomeMethod(SomeMethodInput input)
{
    //A user can not execute this method if he did not login.
}
```

##### （2）AbpAuthorize属性说明(AbpAuthorize attribute notes)

Abp使用动态方法拦截进行权限验证。因此，使用AbpAuthorize特性的方法会有些限制。如下：

- 不能应用于私有(private)方法
- 不能应用于静态(static)方法
- 不能应用于非注入(non-injected)类（我们必须用依赖注入）。

此外，

- AbpAuthorize特性可以应用于任何的Public方法，如果此方法被接口调用（比如在Application Services中通过接口调用）
- 方法是虚(virtual)方法，如果此方法直接被类引用进行调用（像是ASP.NET MVC 或 Web API 的控制器）。
- 方式是虚(virtual)方法，如果此方法是protected。

注意：有三种AbpAuthorize 特性：

- 在应用程序服务中（application layer），我们使用Abp.Authorization.AbpAuthorize；
- 在MVC控制器（web layer）中，我们使用Abp.Web.Mvc.Authorization.AbpMvcAuthorize；
- 在ASP.NET Web API，我们使用 Abp.WebApi.Authorization.AbpApiAuthorize。

这三个类继承自不同的地方。

- 在MVC中，它继承自MVC自己的Authorize类。
- 在Web API，它继承自Web API 的Authorize类。因此，它最好是继承到MVC和Web API中。
- 但是，在Application 层，它完全是由Abp自己实现没有扩展子任何类。

##### （3）使用IPermissionChecker

AbpAuthorize 适用于大部分的情况，但是某些情况下，我们还是需要自己在方法体里进行权限验证。我们可以注入和使用IPermissionChecker对象。如下边的代码所示：

```
public void CreateUser(CreateOrUpdateUserInput input)
{
    if (!PermissionChecker.IsGranted("Administration.UserManagement.CreateUser"))
    {
        throw new AbpAuthorizationException("You are not authorized to create user!");
    }
}
```

```
    }

    //A user can not reach this point if he is not granted for "Administration.UserManagement.CreateUser" permission
}
```

当然，你可以写入任何逻辑，由于IsGranted方法只是简单返回true或false（它还有异步版本哦）。如你简单的检查一个权限并抛出一个异常如上边代码那样，你可以用Authorize方法：

```
public void CreateUser(CreateOrUpdateUserInput input)
{
    PermissionChecker.Authorize("Administration.UserManagement.CreateUser");

    //A user can not reach this point if he is not granted for "Administration.UserManagement.CreateUser" permission
}
```

由于权限验证通常实现与Application层，ApplicationService基础类注入和定义了PermissionChecker属性。因此，权限检查器允许你在Application Service类使用，而不需要显示注入。

## 4.5 ABP应用层 - 功能管理

### 4.5.1 简介

大多数的SaaS(多租户)应用拥有多个版本并且这些版本的功能各不相同。因此，他们能为客户提供不同的价格和功能选项。

我们可以很容易的用ABP来实现这个功能管理系统。我们能定义一些功能，检查功能是否为租户开启。这个就像ABP的设计思想（例如权限和菜单设计）。

#### 关于 IFeatureValueStore

我们可以利用 IFeatureValueStore获取功能值。在module-zero项目里面已有了全部实现，当然你也能用自己的方式来扩展它们。如果没有实现该接口，NullFeatureValueStore（默认实现）会被使用，它会为所有功能值返回null（默认值将会被用在这个案例中）。

### 4.5.2 功能类型

系统已有两个基本的功能类型。

#### 1. Boolean 功能

可以是true或者false。利用这个类型我们可以为租户启用（enable）或者禁用（disable）某些功能。

#### 2. Value 功能

可以为任意值。我们可以用它来存储并者取得一个字符串，我们也可以很容易的用数字作为字符串来存储。

例如，我们有一个任务管理系统，可以限制一个月内的任务创建数量的需求。那么我们可以根据需求创建两个版本系统：一个是每个月允许创建1000个任务，而另外一个则是每个月允许创建5000个任务。所以，这个功能应该用Value类型来存储，而不是用简单的Boolean类型（true或者false）。

#### 3. 定义功能

功能应该在检查之前被定义。模块可以定义它自己的功能，通过从FeatureProvider 派生。下面代码为我们展示了一个非常简单的具有3个功能的功能提供器。

```
public class AppFeatureProvider : FeatureProvider
{
    public override void SetFeatures(IFeatureDefinitionContext context)
    {
        var sampleBooleanFeature = context.Create("SampleBooleanFeature", defaultValue: "false");
        sampleBooleanFeature.CreateChildFeature("SampleNumericFeature", defaultValue: "10");
        context.Create("SampleSelectionFeature", defaultValue: "B");
    }
}
```

在创建了功能提供器后，我们应该在我们的模块方法 PreInitialize中注册它。如下所示：

```
Configuration.Features.Providers.Add<AppFeatureProvider>();
```

## 4. 基本功能属性

定义功能至少需要两个属性：

- **Name:** 用来区分功能的唯一标识符（string类型）；
- **Default value:** 默认值。当我们需要这个功能值的时候我们就会用到它，并且该值对当前租户是不可用的。

在这里，我们定义了一个叫做SampleBooleanFeature的boolean类型的功能，默认值是false（禁用）。我们还定义了两个value类型的功能（SampleNumericFeature 是SampleBooleanFeature的子功能）。

建议：创建字符串常量的功能名称并且我们可以在任何地方使用它，还可以有效避免功能名称输入错误（记忆力有时候不行咯）。

## 5. 其它功能属性

唯一标识（name）和默认值属性已经足够满足需求了，但这里还有一些其它的属性来满足更细粒化的控制需求。

- **Scope:** 枚举类型FeatureScopes的某个值，可以是Edition（如果该功能只对Edition等级的用户开放），Tenant（如果该功能仅对Tenant等级的用户开放）或者All（All等级可以拥有Edition和Tenant等级的所有功能，但是租户的设置会覆盖版本的设置）；默认设置是All级别。
- **DisplayName:** 本地化字符串，向用户展示功能名称。
- **Description:** 本地化字符串，向用户的详细描述该功能。
- **InputType:** UI层的输入类型（控件类型：Checkbox, Combobox等）。
- **Attributes:** 与功能相关的自定义任意类型的数据字典。

下面代码详细展示了如何使用上面所描述的属性

```
public class AppFeatureProvider : FeatureProvider
{
    public override void SetFeatures(IFeatureDefinitionContext context)
    {
        var sampleBooleanFeature = context.Create(
            AppFeatures.SampleBooleanFeature,
            defaultValue: "false",
            displayName: L("Sample boolean feature"),
            inputType: new CheckboxInputType()
        );

        sampleBooleanFeature.CreateChildFeature(
            AppFeatures.SampleNumericFeature,
            defaultValue: "10",
            displayName: L("Sample numeric feature"),
            inputType: new SingleLineStringInputType(new NumericValueValidator(1, 1000000))
        );

        context.Create(
            AppFeatures.SampleSelectionFeature,
            defaultValue: "B",
            displayName: L("Sample selection feature"),
            inputType: new ComboboxInputType(
                new StaticLocalizableComboboxItemSource(
                    new LocalizableComboboxItem("A", L("Selection A")),
                    new LocalizableComboboxItem("B", L("Selection B")),
                    new LocalizableComboboxItem("C", L("Selection C"))
                )
            )
        );
    }

    private static ILocalizableString L(string name)
    {
        return new LocalizableString(name, AbpZeroTemplateConsts.LocalizationSourceName);
    }
}
```

注意：这个输入类型的定义不是被ABP使用，ABP提供这些选项让我们能够更便捷的来创建我们所需要的功能。



### 4.5.3 功能层次

正如上面示例所展示的，功能可以有子功能。父功能通常被定义为boolean类型的功能，如果父功能被启用，那么子功能将会是可用的。ABP不会强制这样使用，但是建议你这样使用。

### 4.5.4 功能检测

在系统中我们定义这些功能，是用来检测这些功能是否应该对每个用户（租户）启用（允许）或者禁用（阻止）。我们可以用不同的方法来检测它们。

#### 2. 使用RequiresFeature特性

我们可以在类或者方法上面用RequiresFeature特性，如下所示：

```
[RequiresFeature("ExportToExcel")]
public async Task<FileDto> GetReportToExcel(...)
{
    ...
}
```

如果ExportToExcel功能为当前租户（从IAbpSession获取当前租户）启用，那么这个方法会被执行；如果没有被启用，那么将会自动的抛出一个AbpAuthorizationException 异常。

当然RequiresFeature特性应该使用的是boolean类型功能，否则你会得到一个异常。

- 不能用在私有方法上
- 不能用在静态方法上
- 不能用在非注入类的方法上（我们必须使用Dependency Injection）。

还有：

- 可以用在任何public方法上，如果这个方法是通过扩展接口来实现的（就像Application Services扩展指定接口一样）
- 方法应该是virtual方法，如果被直接从类引用中调用（就像 ASP.NET MVC 或者 Web API Controllers）。
- 方法应该是virtual方法，如果它是一个protected。

#### 3. 使用IFeatureChecker

我们可以注入和使用IFeatureChecker接口来手动的检测功能（它是被自动注入，并对Application Services，MVC和Web API Controllers 直接可用）。

#### 4. IsEnabled

可以简单的检测，如果给出的功能是启用或者禁用。如下所示：

```
public async Task<FileDto> GetReportToExcel(...)
{
    if (await FeatureChecker.IsEnabledAsync("ExportToExcel"))
    {
        throw new AbpAuthorizationException("You don't have this feature: ExportToExcel");
    }

    ...
}
```

IsEnabledAsync 还有其他的方法也有同步版本。

当然，IsEnabled方法应该使用boolean类型功能。否则，你会得到一个异常。

正如示例所示，如果你只是想检测功能并且抛出异常，你只需要使用CheckEnabled方法。

#### 5. GetValue

获取当前功能的值并且转换为所需类型，如下所示：

```
var createdTaskCountInThisMonth = GetCreatedTaskCountInThisMonth();
if (createdTaskCountInThisMonth >= FeatureChecker.GetValue("MaxTaskCreationLimitPerMonth").To<int>())
{
```

```
        throw new AbpAuthorizationException("You exceed task creation limit for this month, sorry :(");
    }
```

## 6. 客户端

在客户端，我们能使用`abp.features`命名空间去取得当前功能的值。

### isEnabled

```
var isEnabled = abp.features.isEnabled('SampleBooleanFeature');
```

### getValue

```
var value = abp.features.getValue('SampleNumericFeature');
```

## 4.5.5 功能管理

如果你需要定义一些功能，你可以注入和使用`IFeatureManager`。

## 4.5.6 版本须知

ABP没有建立版本系统，因为这样一个系统需要数据库支持（存储版本，版本功能，租客版映射等等）。因此，版本系统被实现在`module zero`。使用它你会很容易的拥有自己的版本系统，或者你可以自己完全的实现一个。## 4.5 ABP应用层—审计日志##

维基百科定义：审计跟踪（也称为审核日志）是一个安全相关的时间顺序记录，记录这些记录的目的是为已经影响在任何时候的详细操作，提供程序运行的证明文件记录、源或事件。

ABP提供了能够为应用程序交互自动记录日志的基础设施，它能记录你调用的方法的方法调用者信息和参数信息。从根本上来说，存储区域包含：

- tenant id（相关的租户Id），
- user id（请求用户Id），
- server name（请求的服务名称【调用方法对应的类】），
- method name（调用方法名称），
- parameters（方法的参数【JSON格式】），
- execution time（执行时间），
- duration（执行耗时时间【通常是毫秒】），
- IP address（客户端IP地址），
- computer name（客户机名称），
- exception（异常【如果方法抛出异常】）等信息。
- 有了这些信息，我们不仅能够知道谁进行了操作，还能够估算出应用程序的性能及抛出的异常。甚至更多的，你可以得到有关应用程序的使用情况统计。

审计系统使用`IAbpSession`接口来获取当前用户Id和租户ID。

注意：关于`IAuditingStore`接口

审计系统使用`IAuditingStore`接口来保存审计信息。`module-zero`项目是这个接口的完整实现，当然你也可以通过自己的方式来实现这个接口。如果你不想自己实现这个接口，`SimpleLogAuditingStore`类可以直接拿来使用，它是实现方式是将审计信息写入日志中。

## 4.5.1 配置

可以在你的模块初始化方法（`PreInitialize`）中使用`Configuration.Auditing`的属性来配置审计，`Auditing`属性默认是启用状态（即`true`）。你可以禁用它，如下图所示：

```
public class MyModule : AbpModule
{
    public override void PreInitialize()
    {
        Configuration.Auditing.IsEnabled = false;
    }
}
```

```
//...
}
```

以下是审计配置的属性：

- **IsEnabled**: 用于设置完全启用或禁用审计系统。默认值: **true**.
- **IsEnabledForAnonymousUsers**: 如果设置成`true`，未登陆的用户的审计日志也会保存。默认值: **false**.
- **MvcControllers**: 在ASP.NET MVC 控制器中使用审计日志
- **IsEnabled**: 在ASP.NET MVC中启用 (**true**) 或禁用 (**false**) 审计日志. 默认值: **true**.
- **IsEnabledForChildActions**: 为MVC actions启用 (**true**) 或禁用 (**false**) 审计日志. 默认值: **false**.
- **Selectors**: 选择使用其他类来处理审计日志的存储。

正如你所看到的，审计系统单独为mvc控制器提供了审计配置使之可以使用不同的方法来使用它。

**Selectors**是一个断言(推断类型)选择器列表，用于选择那种方式来保存审计日志。每一个选择器包含一个唯一的名称和一个断言。断言列表中默认的选择器，使用的是应用程序服务类。如下图所示：

```
Configuration.Auditing.Selectors.Add(
    new NamedTypeSelector(
        "Abp.ApplicationServices",
        type => typeof (IApplcationService).IsAssignableFrom(type)
    )
);
```

你可以在自己的模块初始化方法 (**PreInitialize**) 中添加自己的断言选择器。同样的，如果你不喜欢使用应用程序服务来保存审计日志，你也可以通过名称(name)来移除断言选择器，这就是为什么断言选择器的名称必须是唯一的 (你也可以通过Linq的方式查找到选择器来移除它)。

## 4.5.2 通过属性来启用和禁用审计日志

当你使用配置项来配置断言选择器时，你可以通过使用**Audited**和**DisableAuditing**特性标记到单个类或单个方法来实现审计系统的启用和禁用。例如：

```
[Audited]
public class MyClass
{
    public void MyMethod1(int a)
    {
        //...
    }

    [DisableAuditing]
    public void MyMethod2(string b)
    {
        //...
    }

    public void MyMethod3(int a, int b)
    {
        //...
    }
}
```

上述列子中，**MyClass**类中除了**MyMethod2**明确标记不需要审计外，其他的方法都会被审计。**Audited**特性能够帮助你只想保存审计日志的方法，进行审计日志保存。

## 4.5.3 说明

保存审计日志的方法必须是**public**修饰的，**private**和**protected**修饰的方法将会被忽略。如果调用的方法不在类的引用范围内，那么引用的方法必须是虚方法 (**virtual**)，如果依赖注入的是它自己的接口则不需要是虚方法 (例如注入 **IPersonService** 来使用 **PersonService**类)。

ABP使用动态代理和拦截机制以后，使用虚方法是必须的。对MVC Controller actions 使用审计日志不是正确的做法，因为他们可能不是虚方法。

(4.1、4.2由厦门-浩哥翻译，4.3由Carl翻译，4.4由半冷翻译4.5由冰封翻译)

## 5.1 ABP分布式服务 ASP.NET Web API Controllers

### 5.1.1 简介

ABP已经把 **ASP.NET Web API Controllers** 集成到了 **Abp.Web.Api**，你只要通过Nuget安装它。正如你一直那样做得，你可以创建常规的ASP.NET Web API Controllers，[依赖注入](#)可以解析常规的ApiControllerers。

为了能够享受使用ABP.Web.Api带来的便利，应该使你的controllers派生自 **AbpApiController**。

### 5.1.2 AbpApiController 基类

下面是一个简单的派生自AbpApiController的示例代码：

```
public class UsersController : AbpApiController
{
}
```

#### 1. 本地化

为了使本地化更容易，在AbpApiController中定义了一个方法：**L**；如下所示：

```
public class UsersController : AbpApiController
{
    public UsersController()
    {
        LocalizationSourceName = "MySourceName";
    }

    public UserDto Get(long id)
    {
        var helloWorldText = L("HelloWorld");

        //...
    }
}
```

为了能够使用 **L** 方法，首先我们应该初始化 **LocalizationSourceName**。当然，为了不重复初始化该变量，我们可以定义一个继承自AbpApiController的基类，在该类中初始化它，以后所有的Controller类都继承自该基类。

#### 2. 审计日志

在Api Controller中审计日志默认是未启用状态，所以你应该在Api Controller 或者 Action 上添加 **Audited** 特效来开启该功能。详细请参照 [Audit Logs](#)。

#### 3. Authorization

你可以用AbpApiAuthorize特性来阻止未授权的用户来访问你的Controllers和Actions。示例如下：

```
public class UsersController : AbpApiController
{
    [AbpApiAuthorize("MyPermissionName")]
    public UserDto Get(long id)
    {
        //...
    }
}
```

AbpApiController也定义了一个快速检测权限的方法**IsGranted**。详情请参照[authorization](#)。

#### 4. 工作单元

Web Api Action默认是没有开启工作单元模式的。如果需要开启工作单元，你需要在Action上添加 **UnitOfWork** 的特性，如下所示：

```
public class UsersController : AbpApiController
{
```

```
private readonly IRepository<User, long> _userRepository;

public UsersController(IRepository<User, long> userRepository)
{
    _userRepository = userRepository;
}

[UnitOfWork]
public virtual List<UserDto> Users(string filter)
{
    var users = _userRepository
        .GetAll()
        .Where(u => u.UserName.StartsWith(filter))
        .ToList();

    //...
}
}
```

由于仓储的 `GetAll()` 方法返回的是 `IQueryable` 类型，在调用 `ToList()` 方法的时候需要连接数据库(这是由于 `IQueryable` 的延迟执行特性所决定的)。

注意：使用 `UnitOfWork` 特性的方法，该方法必须是 `virtual` 的，否则拦截器将不能使用。

详情请参照工作单元。

### 5.1.2 其他

你也可以(pre-injected)预先注入 `AbpSession`, `EventBus`, `PermissionManager`, `PermissionChecker`, `SettingManager`, `FeatureManager`, `FeatureChecker`, `LocalizationManager`, `Logger`, `CurrentUnitOfWork` 等基础属性。

## 5.2 ABP表现层 - 动态WebApi层

### 5.2.1 建立动态WebApi控制器

这是一篇关于ASP.NET Web API的文档。如果你对ASP.NET感兴趣，请阅读[ASP.NET Core](#)文档。

Abp框架能够通过应用层自动生成web api:

```
public interface ITaskAppService : IApplicationService
{
    GetTasksOutput GetTasks(GetTasksInput input);
    void UpdateTask(UpdateTaskInput input);
    void CreateTask(CreateTaskInput input);
}
```

并且，我们想要暴露这个服务作为Web API Controller给客户端。那么，Abp框架通过一行关键代码的配置就可以自动、动态的为应用层建立一个web api 控制器:

```
DynamicApiControllerBuilder.For<ITaskAppService>("tasksystem/task").Build();
```

这样就OK了！建好的webapi控制器(/api/services/tasksystem/task)所有的方法都能够在客户端调用。webapi控制器通常是在模块初始化的时候完成配置。

`ITaskAppService`是应用层服务（application service）接口，我们通过封装让接口实现一个api控制器。`ITaskAppService`不仅限于在应用层服务使用，这仅仅是我们习惯和推荐的使用方法。`tasksystem/task`是api 控制器的命名空间。一般来说，应当最少定义一层的命名空间，如：公司名称/应用程序/命名空间/命名空间1/服务名称。`'api/services/`是所有动态web api的前缀。所以api控制器的地址一般是这样滴：`'/api/services/tasksystem/task'`，`GetTasks` 方法的地址一般是这样滴：`'/api/services/tasksystem/task/getTasks'`。因为在传统的js中都是使用驼峰式命名方法，这里也不一样。你也可以删除一个api方法，如下：

```
DynamicApiControllerBuilder
    .For<ITaskAppService>("tasksystem/taskService")
    .ForMethod("CreateTask").DontCreateAction()
    .Build();
```

## ForAll方法

在程序的应用服务层建立多个api控制器可能让人觉得比较枯燥，DynamicApiControllerBuilder提供了建立所有应用层服务的方法，如下所示：

```
DynamicApiControllerBuilder
    .ForAll<IApplicationService>(Assembly.GetAssembly(typeof(SimpleTaskSystemApplicationModule)), "tasksystem")
    .Build();
```

ForAll方法是一个泛型接口，第一个参数是从给定接口中派生的集合，最后一个参数则是services命名空间的前缀。ForAll集合有ITaskAppService和IpersonAppService接口。根据如上配置，服务层的路由是这样的：'/api/services/tasksystem/task'和'/api/services/tasksystem/person'。

服务命名约定：服务名+AppService(在本例中是person+AppService)的后缀会自动删除，生成的webapi控制器名为“person”。同时，服务名称将采用峰驼命名法。如果你不喜欢这种约定，你也可以通过“WithServiceName”方法来自定义名称。如果你不想创建所有的应用服务层，可以使用where来过滤部分服务。

## 重写ForAll

我们可以在ForAll方法后面重写配置，如：

```
DynamicApiControllerBuilder
    .ForAll<IApplicationService>(Assembly.GetAssembly(typeof(SimpleTaskSystemApplicationModule)), "tasksystem")
    .Build();

DynamicApiControllerBuilder
    .For<ITaskAppService>("tasksystem/task")
    .ForMethod("CreateTask").DontCreateAction().Build();
```

在上面代码中，我们为指定的程序集里面的所有Application服务创建了动态WebAPI Controllers。然后为应用服务(ITaskAppService> 重写配置忽略CreateTask方法。

## ForMethods

当我们使用ForAll方法的时候，可以使用 **ForMethods** 方法来更好的调整服务的方法，如：

```
DynamicApiControllerBuilder
    .ForAll<IApplicationService>(Assembly.GetExecutingAssembly(), "app")
    .ForMethods(builder =>
    {
        if (builder.Method.IsDefined(typeof(MyIgnoreApiAttribute)))
        {
            builder.DontCreate = true;
        }
    })
    .Build();
```

在这个示例中，我们使用了一个自定义特性 **MyIgnoreApiAttribute** 来检查所有的方法，如果动态WebAPI Controller的Action上有该特性，那么我们就不会为该Action创建Web API。

## Http 谓词

默认，所有方法的创建是：**POST**。为了能够使用动态Web API Action，客户端应该使用POST发送请求。我们可以改变这个行为以不同的方式。

## WithVerb 方法

我们可以对某个方法使用 **WithVerb**，如下：

```
DynamicApiControllerBuilder
    .For<ITaskAppService>("tasksystem/task")
    .ForMethod("GetTasks").WithVerb(HttpVerb.Get)
    .Build();
```

## Http 特性

我们可以在服务接口的方法上使用HttpGet，HttpPost...等特性：



```
public interface ITaskAppService : IApplicationService
{
    [HttpGet]
    GetTasksOutput GetTasks(GetTasksInput input);

    [HttpPut]
    void UpdateTask(UpdateTaskInput input);

    [HttpPost]
    void CreateTask(CreateTaskInput input);
}
```

为了使用这些特性，我们应该在项目中引用这个包：[Microsoft.AspNet.WebApi.Core](#) nuget package。

命名约定

我们可以使用 **WithConventionalVerbs** 方法来代替Http谓词，如下所示：

```
DynamicApiControllerBuilder
    .ForAll<IApplicationService>(Assembly.GetAssembly(typeof(SimpleTaskSystemApplicationModule)), "tasksystem")
    .WithConventionalVerbs()
    .Build();
```

在这种情况下，ABP可以通过方法名字的前缀来判定Http谓词：

- **Get**: 如果方法的名字是以 **Get** 开头
- **Put**: 如果方法的名字是以 **Put** 或者 **Update** 开头
- **Delete**: 如果方法的名字是以 **Delete** 或者 **Remove** 开头
- **Post**: 如果方法的名字是以 **Post**, **Create** 或者 **Insert** 开头
- **Path**: 如果方法的名字是以 **Path** 开头
- 否则 **Post** 被作为Http谓词的默认设置

## 5.2.2 使用动态JavaScript代理

你可以通过ajax来动态创建web api控制器。Abp框架对通过动态js代理建立web api 控制器做了些简化，你可以通过js来动态调用web api控制器：

```
abp.services.tasksystem.task.getTasks({
    state: 1
}).done(function (data) {
    //use data.tasks here..
});
```

js代理是动态创建的，页面中需要添加引用：

```
<script src="/api/abp.ServiceProxies/GetAll" type="text/javascript"></script>
```

服务方法(service methods)返回约定（可参见JQ的Deferred），服务方法使用Abp框架.ajax代替，可以处理、显示错误。

### 1. Ajax参数

自定义ajax代理方法的参数：

```
Abp.services.tasksystem.task.createTask({
    assignedPersonId: 3,
    description: 'a new task description...'
},{ //override jquery's ajax parameters
    async: false,
    timeout: 30000
}).done(function () {
    Abp.notify.success('successfully created a task!');
});
```

所有的jq.ajax参数都是有效的。

除了标准的JQuery.ajax参数，为了禁用错误信息自动显示，你可以添加AJAX选项：**abpHandleError: false**。

## 2. 单一服务脚本

'/api/abpServiceProxies/GetAll'将在一个文件中生成所有的代理，通过 '/api/abpServiceProxies/Get?name=serviceName' 你也可以生成单一服务代理，在页面中添加：

```
<script src="/api/abpServiceProxies/Get?name=tasksystem/task" type="text/javascript"></script>
```

## 3. Angular框架支持

Abp框架能够公开动态的api控制器作为angularjs服务，如下所示：

```
(function() {  
    angular.module('app').controller('TaskListController', [  
        '$scope', 'abp.services.tasksystem.task',  
        function($scope, taskService) {  
            var vm = this;  
            vm.tasks = [];  
            taskService.getTasks({  
                state: 0  
            }).success(function(data) {  
                vm.tasks = data.tasks;  
            });  
        }  
    ])  
})();
```

我们可以将名称注入服务，然后调用此服务，跟调用一般的js函数一样。注意：我们成功注册处理程序后，他就像一个angular的\$http服务。ABP框架使用angular框架的\$http服务，如果你想通过\$http来配置，你可以设置一个配置对象作为服务方法的一个参数。

要使用自动生成的服务，需要在页面中添加：

```
<script src="~/abp Framework/Framework/scripts/libs/angularjs/Abp Framework.ng.js"></script>  
<script src="~/api/abp Framework/ServiceProxies/GetAll?type=angular"></script>
```

## Enable/Disable

如果你向上面一样使用 **ForAll** 方法，那么你可以使用 **RemoteService** 特性来禁用某个服务或方法。请在 服务的接口 中使用该特性，而不是在服务类中。

### 包装结果

ABP通过 **AjaxResponse** 对象来 包装 动态Web API Actions的返回值。你可以Enable/Disable包装对每个方法或应用服务。如下所示：

```
public interface ITestAppService : IApplicationService  
{  
    [DontWrapResult]  
    DoItOutput DoIt(DoItInput input);  
}
```

我们对DoIt禁用了包装。这个特性应该在添加在接口里面的方法描述上，而不是扩展类上。

如果你想更精确的控制返回值到客户端，禁用包装是非常有用的。特别是在使用第三方客户端库的时候，禁用包装是有必要的，因为ABP的标准包装 **AjaxResponse** 可能不能与之一起使用。在这种情况下，你也应该自己处理异常，因为异常处理会被禁用掉(DontWrapResult特性有个属性：WrapOnError，该属性可以被用来开启异常处理并对异常进行包装)。

注意：动态脚本代理能够使用该结果，如果该结果是未包装，而且能够正常的被处理。

### 关于参数绑定

ABP在运行时创建API Controllers。所以ASP.NET Web API 的模型和参数绑定被使用来绑定模型和参数。你可以读取该文档获取更多信息。

**FormUri** 和 **FromBody** 特性

**FromUri** 和 **FromBody** 特性能够在服务接口中被使用增进对绑定的控制。

数据传输对象 VS 原生类型

对于应用服务和Web API控制器，我强烈建议使用DTO来作为方法参数。但是你也可以使用原生类型(如: string, int, bool ... 或可空类型, 如: int?, bool? ...) 作为服务方法的参数。可以使用多个参数; 但是, 在这些参数中有且只能有一个 **Complex-Type** 参数, 这是ASP.NET Web API 规定的。## 5.3 ABP分布式服务 - 集成OData

### 5.3.1 简介

开放数据协议 (Open Data Protocol, 缩写OData) 是一种描述如何创建和访问Restful服务。你可以在Abp中使用OData, 只需要通过Nuget来安装Abp.Web.Api.OData.

### 5.3.2 安装

#### 1. 使用Nuget安装

首先应该使用Nuget安装Abp.Web.Api.OData到我们的WebApi项目:

```
Install-PackageAbp.Web.Api.OData
```

#### 2. 配置依赖模块

将AbpWebApiODataModule添加到被依赖的模块上, 如下所示:

```
[DependsOn(typeof(AbpWebApiODataModule))]  
public class MyProjectWebApiModule : AbpModule  
{  
    ...  
}
```

了解模块依赖请参考[模块系统](#)

#### 3. 配置实体

为了使实体能够作为一个OData源, 我们应该在模块的 **PreInitialize** 方法中对实体进行OData配置。如下所示:

```
[DependsOn(typeof(AbpWebApiODataModule))]  
public class MyProjectWebApiModule : AbpModule  
{  
    public override void PreInitialize()  
    {  
        var builder = Configuration.Modules.AbpWebApiOData().ODataModelBuilder;  
  
        //Configure your entities here...  
        builder.EntitySet<Person>("Persons");  
    }  
  
    ...  
}
```

在这里, 我们得到了ODataModelBuilder的引用并且配置了Person实体。类似的你可以使用 **EntitySet** 方法来添加其他实体。更多信息请参考[OData文档](#)。

#### 4. 创建控制器

在 **Abp.Web.Api.OData** 中包含了基类: **AbpODataEntityController**(扩展自标准的ODataController), 为了能够更容易的创建你自己的控制器, 你应该继承该基类。如下所示:

```
public class PersonsController : AbpODataEntityController<Person>  
{  
    public PersonsController(IRepository<Person> repository)  
        : base(repository)  
    {  
    }
```

```
}  
}
```

**AbpODataEntityController** 中的所有方法都是virtual，所以你可以重写 *\*Get, Post, Put, Patch, Delete \** 和其它的Actions来实现你自己的逻辑。

### 5.3.3 示例

这里有一些请求上面已定义的Controller的示例。假设你的应用的URL是：<http://localhost:61842>。下面将展示一些基础的例子。

#### 1. 获取实体列表

取得所有的people。

##### Request

GET <http://localhost:61842/odata/Persons>

##### Response

```
{  
  "@odata.context": "http://localhost:61842/odata/$metadata#Persons", "value": [  
    {  
      "Name": "Douglas Adams", "IsDeleted": false, "DeleterUserId": null, "DeletionTime": null, "LastModificationTime": null, "  
    }, {  
      "Name": "John Nash", "IsDeleted": false, "DeleterUserId": null, "DeletionTime": null, "LastModificationTime": null, "Last  
    }  
  ]  
}
```

#### 2. 获取单个实体

根据Id = 2来获取person

##### Request

GET [http://localhost:61842/odata/Persons\(2\)](http://localhost:61842/odata/Persons(2))

##### Response

```
{  
  "@odata.context": "http://localhost:61842/odata/$metadata#Persons/$entity", "Name": "John Nash", "IsDeleted": false, "Del  
}
```

#### 3. 根据导航属性获取单个实体

根据Id = 1来获取person并且包括他的电话信息

##### Request

GET [http://localhost:61842/odata/Persons\(1\)?\\$expand=Phones](http://localhost:61842/odata/Persons(1)?$expand=Phones)

##### Response

```
{  
  "@odata.context": "http://localhost:61842/odata/$metadata#Persons/$entity", "Name": "Douglas Adams", "IsDeleted": false,  
  {  
    "PersonId": 1, "Type": "Mobile", "Number": "4242424242", "CreationTime": "2015-11-07T20:12:39.363+03:00", "CreatorUserI  
  }, {  
    "PersonId": 1, "Type": "Mobile", "Number": "2424242424", "CreationTime": "2015-11-07T20:12:39.363+03:00", "CreatorUserI  
  }  
}
```

```
]
}
```

#### 4. 查询

在这里我们将使用一个高级查询，在过滤，排序后取得排在最前面的2条数据。

##### Request

```
GET http://localhost:61842/odata/Persons?$filter=Name eq 'Douglas Adams'&$orderby=CreationTime&$top=2
```

##### Response

```
{
  "@odata.context": "http://localhost:61842/odata/$metadata#Persons", "value": [
    {
      "Name": "Douglas Adams", "IsDeleted": false, "DeleterUserId": null, "DeletionTime": null, "LastModificationTime": null,
    }, {
      "Name": "Douglas Adams", "IsDeleted": false, "DeleterUserId": null, "DeletionTime": null, "LastModificationTime": null,
    }
  ]
}
```

OData支持分页，排序，过滤，投影功能等等；详情请查看OData网站。

#### 5. 创建新实体

我们会创建一个新的person实体，示例如下：

##### Request

```
POST http://localhost:61842/odata/Persons

{
  Name: "Galileo Galilei"
}
```

注意： "Content\_Type" 头的值是 "application/json"

##### Response

```
{
  "@odata.context": "http://localhost:61842/odata/$metadata#Persons/$entity",
  "Name": "Galileo Galilei",
  "IsDeleted": false,
  "DeleterUserId": null,
  "DeletionTime": null,
  "LastModificationTime": null,
  "LastModifierUserId": null,
  "CreationTime": "2016-01-12T20:36:04.1628263+02:00",
  "CreatorUserId": null,
  "Id": 4
}
```

如果我们再次获取一次实体列表，我们会看到新增的person；我们也可以更新或者删除一个已经存在的实体，这正是OData协议所支持的。

#### 6. 获取元数据

我们可以获取一个实体的元数据，如下所示：

##### Request

```
GET http://localhost:61842/odata/$metadata
```

## Response

```
<?xml version="1.0" encoding="utf-8"?>

<edmx:Edmx Version="4.0" xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx">
  <edmx:DataServices>
    <Schema Namespace="Abp0DataDemo.People" xmlns="http://docs.oasis-open.org/odata/ns/edm">
      <EntityType Name="Person">
        <Key>
          <PropertyRef Name="Id" />
        </Key>
        <Property Name="Name" Type="Edm.String" Nullable="false" />
        <Property Name="IsDeleted" Type="Edm.Boolean" Nullable="false" />
        <Property Name="DeleterUserId" Type="Edm.Int64" />
        <Property Name="DeletionTime" Type="Edm.DateTimeOffset" />
        <Property Name="LastModificationTime" Type="Edm.DateTimeOffset" />
        <Property Name="LastModifierUserId" Type="Edm.Int64" />
        <Property Name="CreationTime" Type="Edm.DateTimeOffset" Nullable="false" />
        <Property Name="CreatorUserId" Type="Edm.Int64" />
        <Property Name="Id" Type="Edm.Int32" Nullable="false" />
        <NavigationProperty Name="Phones" Type="Collection(Abp0DataDemo.People.Phone)" />
      </EntityType>
      <EntityType Name="Phone">
        <Key>
          <PropertyRef Name="Id" />
        </Key>
        <Property Name="PersonId" Type="Edm.Int32" />
        <Property Name="Type" Type="Abp0DataDemo.People.PhoneType" Nullable="false" />
        <Property Name="Number" Type="Edm.String" Nullable="false" />
        <Property Name="CreationTime" Type="Edm.DateTimeOffset" Nullable="false" />
        <Property Name="CreatorUserId" Type="Edm.Int64" />
        <Property Name="Id" Type="Edm.Int32" Nullable="false" />
        <NavigationProperty Name="Person" Type="Abp0DataDemo.People.Person">
          <ReferentialConstraint Property="PersonId" ReferencedProperty="Id" />
        </NavigationProperty>
      </EntityType>
      <EnumType Name="PhoneType">
        <Member Name="Unknown" Value="0" />
        <Member Name="Mobile" Value="1" />
        <Member Name="Home" Value="2" />
        <Member Name="Office" Value="3" />
      </EnumType>
    </Schema>
    <Schema Namespace="Default" xmlns="http://docs.oasis-open.org/odata/ns/edm">
      <EntityContainer Name="Container">
        <EntitySet Name="Persons" EntityType="Abp0DataDemo.People.Person" />
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

元数据被用来服务审查。

### 5.3.4 示例项目

你可以从这里获得示例项目的代码：：<https://github.com/aspnetboilerplate/sample-odata>

## 5.4 ABP分布式服务 - 集成SwaggerUI

### 5.4.1 简介

从官网得知，开启Swagger，你可以获得一个交互式的文档，生成和发现客户端SDK。

### 5.4.2 ASP.NET Core

#### 1. 安装

你可以很容易的集成Swagger到基于ASP.NET Core的ABP项目。

#### 2. 安装Nuget包

安装 **Swashbuckle** nuget包到你的Web项目。



### 3. 配置

在 **Startup.cs** 文件中找到 **ConfigureServices** 方法，并在该方法中为Swagger添加配置代码。

```
public IServiceCollection ConfigureServices(IServiceCollection services)
{
    //其它代码...

    services.AddSwaggerGen();

    //其它代码...
}
```

然后在 **Startup.cs** 的 **Configure** 方法中添加如下配置：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    //其它代码...

    app.UseSwagger();
    app.UseSwaggerUi(); //URL: /swagger/ui
}
```

在你使用 **swagger ui** 测试动态 **Web Api** 的时候，为了同时能够发送 **CSRF token**，你需要添加 **swagger ui** 的 **index.html** 文件到你的web项目。该文件应该精确的放在 **wwwroot\swagger\ui** 文件夹下。然后你需要更改**swagger ui**的**index.html**中已定义的**onComplete** 方法，如下所示：

```
onComplete: function(swaggerApi, swaggerUi){
    if(typeof initOAuth == "function") {
        initOAuth({
            clientId: "your-client-id",
            clientSecret: "your-client-secret-if-required",
            realm: "your-realms",
            appName: "your-app-name",
            scopeSeparator: " ",
            additionalQueryStringParams: {}
        });
    }

    if(window.SwaggerTranslator) {
        window.SwaggerTranslator.translate();
    }

    var csrfToken = abp.security.antiForgery.getToken();
    var csrfCookieAuth = new SwaggerClient.ApiKeyAuthorization(abp.security.antiForgery.tokenHeaderName, csrfToken, "header");
    swaggerUi.api.clientAuthorizations.add(abp.security.antiForgery.tokenHeaderName, csrfCookieAuth);
}
```

了解更多的配置选项请参考Swashbuckle的文档。

## 5.4.3 ASP.NET 5.x

### 1. 安装

你可以很容易的集成Swagger到基于ABP创建的项目中。

### 2. Nuget安装

安装 **Swashbuckle.Core**到你的WebApi项目或者Web项目

### 2. 配置

在你的模块的 **Initialize** 方法中添加配置Swagger的代码，如下所示：

```
public class SwaggerIntegrationDemoWebApiModule : AbpModule
{
    public override void Initialize()
    {
    }
```

```

//其它代码...

ConfigureSwaggerUi();
}

private void ConfigureSwaggerUi()
{
    Configuration.Modules.AbpWebApi().HttpConfiguration
        .EnableSwagger(c =>
        {
            c.SingleApiVersion("v1", "SwaggerIntegrationDemo.WebApi");
            c.ResolveConflictingActions(apiDescriptions => apiDescriptions.First());
        })
        .EnableSwaggerUi(c =>
        {
            c.InjectJavaScript(Assembly.GetAssembly(typeof(AbpProjectNameWebApiModule)), "AbpCompanyName.AbpProjectName.WebApi.SwaggerCustom.js");
        });
}
}

```

注意：在配置swagger ui的时候，我们注入的脚本文件名称是 **Swagger-Custom.js**。当你使用swagger ui来测动态web api的时候，这个脚本文件被用来添加 **CSRF token** 到你的请求头中。你还需要添加这个文件到你的WebApi项目中并使用它的逻辑名在InjectJavaScript方法中注入它。

该文件中的脚本内容如下：

```

var getCookieValue = function(key) {
    var equalities = document.cookie.split('; ');
    for (var i = 0; i < equalities.length; i++) {
        if (!equalities[i]) {
            continue;
        }

        var splitted = equalities[i].split('=');
        if (splitted.length !== 2) {
            continue;
        }

        if (decodeURIComponent(splitted[0]) === key) {
            return decodeURIComponent(splitted[1] || '');
        }
    }

    return null;
};

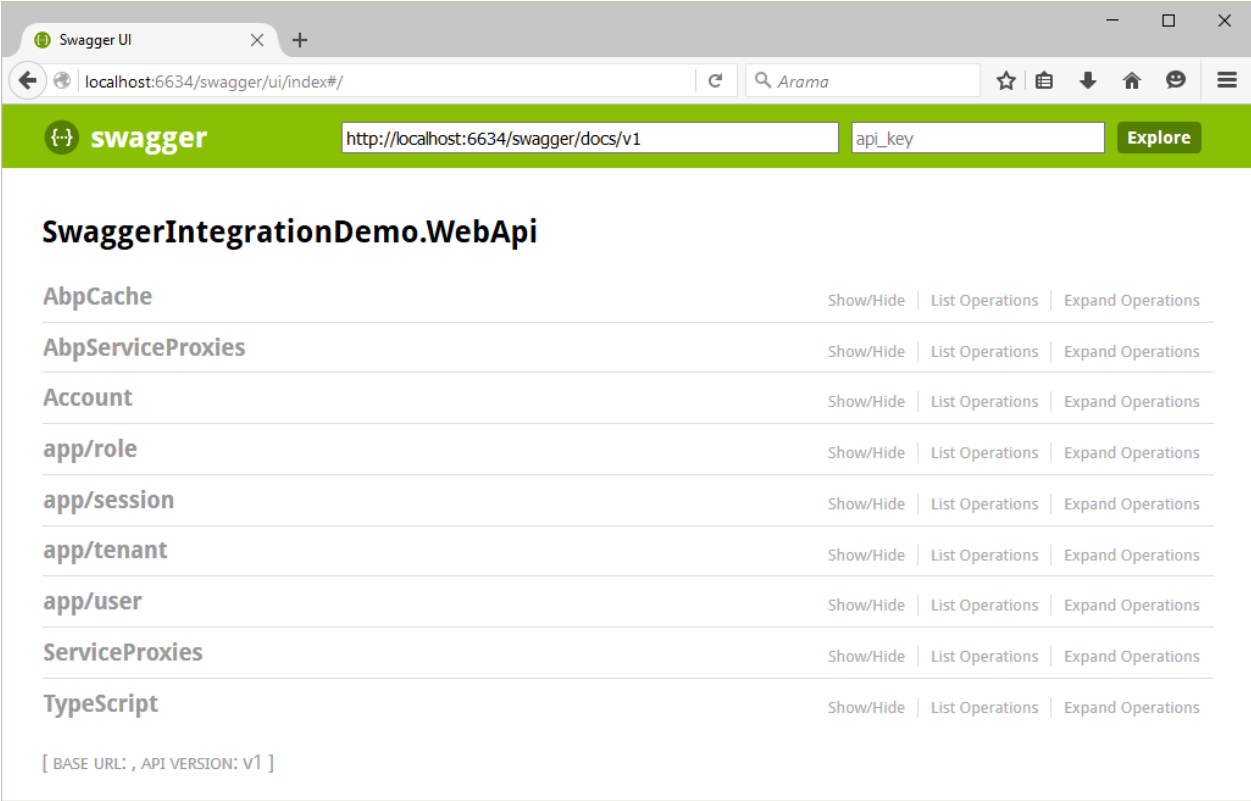
var csrfCookie = getCookieValue("XSRF-TOKEN");
var csrfCookieAuth = new SwaggerClient.ApiKeyAuthorization("X-XSRF-TOKEN", csrfCookie, "header");
swaggerUi.api.clientAuthorizations.add("X-XSRF-TOKEN", csrfCookieAuth);

```

更多配置请参考Swashbuckle文档

#### 5.4.4. 测试

按照如上配置后，让我们来浏览下文档 `/swagger/ui/index`：



你能看到所有的 Web Api Controllers 包括动态 Web Api Controllers，并且现在你可以测试它们。

## 6.1 ABP表现层 - Mvc Controllers

### 6.1.1 简介

ABP已经把ASP.NET MVC Controllers集成到了Abp.Web.Mvc中，正如你一直那样做的，你可以使用常规的MVC Controllers。依赖注入可以解析常规的MVC Controllers。

但是你的Controllers应该从AbpController派生，因为在该基类中ABP已经提供了一些有用的便利方法。

#### 6.1.2 AbpController 基类

下面是一个简单的派生自AbpApiController的示例代码：

```
public class HomeController : AbpController
{
    public ActionResult Index()
    {
        return View();
    }
}
```

#### 1. 本地化

为了使本地化更简单，ABP在AbpController中定义了一个L方法。示例如下：

```
public class HomeController : AbpController
{
    public HomeController()
    {
        LocalizationSourceName = "MySourceName";
    }

    public ActionResult Index()
    {
        var helloWorldText = L("HelloWorld");

        return View();
    }
}
```

为了能够使用 **L** 方法, 首先我们应该初始化 **LocalizationSourceName**。当然, 为了不重复初始化该变量, 我们可以定义一个继承自 **AbpController** 的基类, 在该类中初始化它, 以后所有的 **Controller** 类都继承自该基类。

## 2. 异常处理

异常是自动被处理的, 会被记录到日志并且以一个适当的方式返回到客户端。详情请参考[异常处理](#)。

## 3. 包装返回内容

如果返回的类型是 **JsonResult** (或者异步 **Action** 的 **Task**) **ABP** 默认会将 **Action** 的返回内容进行包装。你可以改变这个, 只要在 **Controller** 的 **Action** 上面使用特性: **WrapResult** 或者 **DontWrapResult**, 详情请参考[Javascript函数库](#)

## 4. 审计日志

如果派生自 **AbpController**, 那么日志会被自动记录。详情请参考[审计日志](#)。

## 5. 权限认证

你可以用 **AbpMvcAuthorize** 特性来阻止未授权的用户来访问你的 **Controllers** 和 **Actions**。示例如下:

```
public class HomeController : AbpController
{
    [AbpMvcAuthorize("MyPermissionName")]
    public ActionResult Index()
    {
        return View();
    }
}
```

**AbpController** 也定义了一个快速检测权限的方法 **IsGranted**。详情请参照[权限认证](#)。

## 6. 工作单元

**MVC Action** 默认是没有开启工作单元模式的。如果需要开启工作单元, 你需要在 **Action** 上添加 **UnitOfWork** 的特性, 如下所示:

```
public class HomeController : AbpController
{
    private readonly IRepository<User, long> _userRepository;

    public HomeController(IRepository<User, long> userRepository)
    {
        _userRepository = userRepository;
    }

    [UnitOfWork]
    public virtual ActionResult Users(string filter)
    {
        var users = _userRepository
            .GetAll()
            .Where(u => u.UserName.StartsWith(filter))
            .ToList();

        return View(users);
    }
}
```

由于仓储的 **GetAll()** 方法返回的是 **IQueryable** 类型, 在调用 **ToList()** 方法的时候需要连接数据库(这是由于 **IQueryable** 的延迟执行特性所决定的)。

注意: 使用 **UnitOfWork** 特性的方法, 该方法必须是 **virtual** 的, 否则拦截器将不能使用。

详情请参照[工作单元](#)。

### 6.1.3 其他

你也可以(pre-injected)预先注入 **AbpSession**, **EventBus**, **PermissionManager**, **PermissionChecker**, **SettingManager**, **FeatureManager**, **FeatureChecker**, **LocalizationManager**, **Logger**, **CurrentUnitOfWork** 等基础属性。## 6.2 ABP表现层 - Mvc视

图

### 6.2.1 简介

在 **Abp.Web.Mvc** 中, ABP集成了MVC视图。你可以像往常一样创建常规的MVC视图。

### 6.2.2 AbpWebViewPage 基类

ABP定义了一个AbpWebViewPage基类, 在该类中定义了一些有用的属性和方法。

如果你使用**startup templates**来创建项目, 那么你的视图会自动的继承该基类。

AbpWebViewPage 定义了本地化方法 **L**, 权限验证方法 **IsGranted**, 功能管理中的方法 **IsFeatureEnabled** 和 **GetFeatureValue** 等等一些常用方法属性。**## 6.3 ABP表现层 - 本地化**

### 6.3.1 程序语言

ABP框架为程序提供了一个灵活的本地化配置模块。首先需要的是声明支持哪些语言, 模块预初始化如下:

```
Configuration.Localization.Languages.Add(new LanguageInfo("en", "English", "famfamfam-flag-england", true));
Configuration.Localization.Languages.Add(new LanguageInfo("tr", "Türkçe", "famfamfam-flag-tr"));
```

在服务器端, 你需要使用ILocalizationManager。在客户端, 你可以使用abp.localization javascript API从当前的语言列表中获取你需要的语言配置文件。

### 6.3.2 本地化源文件

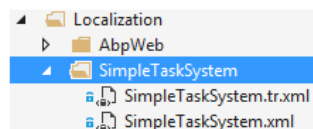
本地化文本可存储在不同的地方, 你也可以在同一个应用程序中使用多个来源。ILocalizationManager接口可实现本地化, 然后注册到ABP的本地化配置文件。可通过xml和资源文件来配置。

#### 1. XML文件(XML files)

本地化配置文件可存储在XML文件。如下所示:

```
<?xml version="1.0" encoding="utf-8" ?>
<localizationDictionary culture="en">
  <texts>
    <text name="TaskSystem" value="Task System" />
    <text name="TaskList" value="Task List" />
    <text name="NewTask" value="New Task" />
    <text name="Xtasks" value="{0} tasks" />
    <text name="CompletedTasks" value="Completed tasks" />
    <text name="EmailWelcomeMessage">Hi,
Welcome to Simple Task System! This is a sample
email content.</text>
  </texts>
</localizationDictionary>
```

XML文件必须使用 **UTF-8** 编码, **culture="en"** 表示这个XML文件包含的是英文文本。对于文本节点: **name** 属性被用来识别一个文本, 你可以用 **value** 属性或者 **inner text(内部文本)** (例如最后一个)来设置本地化文本的值。我们会对每种语言分别创建XML文件, 如下所示:



**SimpleTaskSystem** 是资源文件的名字, SimpleTaskSystem.xml是默认的本地化语言. 当某个文本被请求的时候, ABP会从当前的XML资源文件中取得本地化文本(通过 **Thread.CurrentThread.CurrentCulture** 来找到当前所使用的语言)。如果当前所使用的语言不存在, 那么将会使用默认的XML的语言资源。

为XML本地化资源注册

XML文件能够被存储在 文件系统中 或者能被嵌入进一个程序集中。

对于 文件系统 存储XML文件, 我们可以向下面一样注册一个XML本地化资源:

```
Configuration.Localization.Sources.Add(
    new DictionaryBasedLocalizationSource(
        "SimpleTaskSystem",
        new XmlFileLocalizationDictionaryProvider(
            HttpContext.Current.Server.MapPath("~/Localization/SimpleTaskSystem")
        )
    )
);
```

我们应该在模块的 **PreInitialize** 方法中来配置, 详情请参照模块系统。ABP 会从给定的目录中找到所有的XML文件并且注册这些本地化资源。

对于 嵌入式 XML 文件, 我们应该标记所有的XML文件为 嵌入式资源(**Embedded resource**) (选择XML文件, 打开属性窗口(F4) 并且更改生成操作为嵌入式资源)。然后你可以像下面一样注册本地化资源:

```
Configuration.Localization.Sources.Add(
    new DictionaryBasedLocalizationSource(
        "SimpleTaskSystem",
        new XmlEmbeddedFileLocalizationDictionaryProvider(
            Assembly.GetExecutingAssembly(),
            "MyCompany.MyProject.Localization.Sources"
        )
    )
);
```

**XmlEmbeddedFileLocalizationDictionaryProvider** 会获得一个包含XML文件的程序集(GetExecutingAssembly简单地指向当前的程序集)和XML文件的名称空间(程序集名称+xml文件的文件夹层次)。

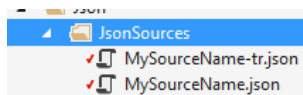
注意: 当给嵌入式XML文件添加语言后缀时, 不要用 "." 例如: "MySource.tr.xml", 而是应该用破折号 "-" 例如: "MySource-tr.xml", 因为在程序寻找资源文件的时候点号会导致资源文件找不到等异常。

## 2. JSON 文件

JSON文件可以被用来存储本地化资源文本, 正如下面所展示的一个简单的JSON本地化示例:

```
{
  "culture": "en",
  "texts": {
    "TaskSystem": "Task system",
    "Xtasks": "{0} tasks"
  }
}
```

JSON文件应使用 **UTF-8** 编码, **culture="en"** 表示这个JSON文件包含的是英文文本。我们会对每种语言分别创建JSON文件, 如下所示:



这里的MySourceName是资源名称, MySourceName.json被定义为默认的资源语言。这类似于XML资源文件的定义。

对JSON本地化资源的注册

JSON文件能够被存储在 文件系统中 或者能被嵌入进一个程序集中。

对于 文件系统 存储JSON文件, 我们可以向下面一样注册一个JSON本地化资源:

```
Configuration.Localization.Sources.Add(
    new DictionaryBasedLocalizationSource(
        "MySourceName",
        new JsonFileLocalizationDictionaryProvider(
            HttpContext.Current.Server.MapPath("~/Localization/MySourceName")
        )
    )
);
```



我们应该在模块的 **PreInitialize** 方法中来配置，详情请参照**模块系统**。ABP会从给定的目录中找到所有的JSON文件并且注册这些本地化资源。

对于 嵌入式 JSON文件，我们应该标记所有的JSON文件为 嵌入式资源(**Embedded resource**) (选择JSON文件，打开属性窗口(F4)并且更改生成操作为嵌入式资源)。然后你可以像下面一样注册本地化资源：

```
Configuration.Localization.Sources.Add(
    new DictionaryBasedLocalizationSource(
        "MySourceName",
        new JsonEmbeddedFileLocalizationDictionaryProvider(
            Assembly.GetExecutingAssembly(),
            "MyCompany.MyProject.Localization.Sources"
        )
    );
```

**JsonEmbeddedFileLocalizationDictionaryProvider** 会获得一个包含JSON文件的程序集(GetExecutingAssembly简单地指向当前的程序集)和JSON文件的名称空间(程序集名称+JSON文件的文件夹层次)。

注意：当给嵌入式JSON文件添加语言后缀时，不要用 "." 例如： "MySource.tr.json", 而是应该用破折号 "-" 例如： "MySource-tr.json", 因为在程序寻找资源文件的时候点号会导致资源文件找不到等异常。

关于XML和JSON本地化资源的混合使用的小技巧

如果你想在你的项目中同时使用xml, json的本地化资源那么该怎么用呢？

首先注意：你的Core项目中的Localization底下的资源文件夹的命名方式应该是：

文件夹得命名方式必须要首字母不同例如：

JsonSource

```
myproject.json
myproject-zh-cn.json
```

XmlSource

```
myproject.xml
myproject-zh-cn.xml
```

或者

JSource

XSource

因为ABP框架的判断逻辑是

resourceName : ABP框架通过程序集去取得的assembly.GetManifestResourceNames()

\_rootNamespace : 是你自己给予的

resourceName.StartsWith(\_rootNamespace)

详细请查看类XmlEmbeddedFileLocalizationDictionaryProvider

在你的模块类中写上这样的代码：

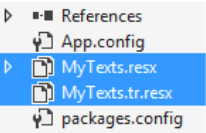
```
public override void PreInitialize()
{
    Configuration.Localization.Sources.Add(
        new DictionaryBasedLocalizationSource(
            AuthConsts.LocalizationSourceName,
            new XmlEmbeddedFileLocalizationDictionaryProvider(
                Assembly.GetExecutingAssembly(),
                "MyCompany.MyProject.Localization.XmlSource"
            )
        );
}
```

```
        )
    }
};

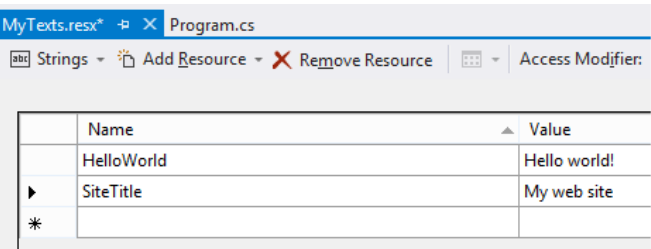
Configuration.Localization.Sources.Extensions.Add(
    new LocalizationSourceExtensionInfo(
        AuthConsts.LocalizationSourceName,
        new JsonEmbeddedFileLocalizationDictionaryProvider(
            Assembly.GetExecutingAssembly(),
            "MyCompany.MyProject.Localization.JsonSource"
        )));
}
```

### 3. 自定义资源(Custom source)

当然也可以将本地化配置存储在.net的资源文件中。我们可以为每种语言创建一个配置文件。如下：



MyTexts.resx包含了默认的语言配置文件，MyTexts.tr.resx包含了土耳其语言配置文件。当打开MyTexts.resx:



配置方法如下：

```
Configuration.Localization.Sources.Add(
    new ResourceFileLocalizationSource(
        "MySource",
        MyTexts.ResourceManager
    ));
```

这里设置的唯一名字是MySource，在模块初始化过程中，MyTexts.ResourceManager将被作为本地化文本导入到资源管理器中。一个默认的本地化文本配置文件可存储到数据库中，可以通过ILocalization接口或者通过DictionaryBasedLocalizationSource类来简化实现。

#### 6.3.3 获得一个本地化配置文件

在创建并注册到ABP本地化系统后，文件将更容易的被本地化。

##### 1. 服务端(In server side)

在服务端，你可以简单的LocalizationHelper.GetString调用源配置文件中的值。

```
var s1 = LocalizationHelper.GetString("SimpleTaskSystem", "NewTask");
```

确保不要有同名的配置文件，你可以先获取源配置文件，然后获取一个源文件中的一个值。

```
var source = LocalizationHelper.GetSource("SimpleTaskSystem");
var s1 = source.GetString("NewTask");
```

这将返回当前的语言配置文件。并重写GetString方法。

##### 2. MVC控制器(In MVC controllers)

通过MVC控制器或视图来本地化：

```
public class HomeController : SimpleTaskSystemControllerBase
{
    public ActionResult Index()
    {
        var helloWorldText = L("HelloWorld");
        return View();
    }
}
```

方法用于本地化字符串，前提是需要提供一个配置文件的名称。在SimpleTaskSystemControllerBase:

```
public abstract class SimpleTaskSystemControllerBase : AbpController
{
    protected SimpleTaskSystemControllerBase()
    {
        LocalizationSourceName = "SimpleTaskSystem";
    }
}
```

注意：它来自AbpController。因此，你能更容易的通过L方法获取本地化配置文件的值。

### 3. 在MVC视图(In MVC views)

在视图中的L方法:

```
<div>
    <form id="NewTaskForm" role="form">
        <div class="form-group">
            <label for="TaskDescription">@L("TaskDescription")</label>
            <textarea id="TaskDescription" data-bind="value: task.description" class="form-control" rows="3" placeholder="Task Description" ></textarea>
        </div>
        <div class="form-group">
            <label for="TaskAssignedPerson">@L("AssignTo")</label>
            <select id="TaskAssignedPerson" data-bind="options: people, optionsText: 'name', optionsValue: 'id', value: task.assignedPersonId" ></select>
        </div>
        <button data-bind="click: saveTask" type="submit" class="btn btn-primary">@L("CreateTheTask")</button>
    </form>
</div>
```

在实际应用中，你应当从基础类中派生你自己的视图:

```
public abstract class SimpleTaskSystemWebViewPageBase : SimpleTaskSystemWebViewPageBase<dynamic>
{
}

public abstract class SimpleTaskSystemWebViewPageBase<TModel> : AbpWebViewPage<TModel>
{
    protected SimpleTaskSystemWebViewPageBase()
    {
        LocalizationSourceName = "SimpleTaskSystem";
    }
}
```

还需要在web.config设置基础视图:

```
<pages pageBaseType="SimpleTaskSystem.Web.Views.SimpleTaskSystemWebViewPageBase">
```

在JavaScript中

当你从一个ABP模版中建立你的解决方案时，所有从控制器和视图都已经准备:

```
<script src="/AbpScripts/GetScripts" type="text/javascript"></script>
```

ABP在客户端自动生成需要的JavaScript代码来获取本地化配置文件，在JavaScript中可以很容易地获取本地化配置文件:

```
var s1 = abp.localization.localize('NewTask', 'SimpleTaskSystem');
```

NewTask是一个配置文件中的某一行的名字，SimpleTaskSystem是一个配置文件的名字：

```
var source = abp.localization.getSource('SimpleTaskSystem');
var s1 = source('NewTask');
```

本地化方法也能够设置额外的方法参数：

```
abp.localization.localize('RoleDeleteWarningMessage', 'MySource', 'Admin');
//shortcut if source is got using getSource as shown above
source('RoleDeleteWarningMessage', 'Admin');
```

如果使用RoleDeleteWarningMessage = 'Role {0} will be deleted'，本地化文件中的文本将变成 'Role Admin will be deleted'.

### 6.3.4 总结

ABP为本地化提供了不同的源文件的能力，它也为服务器端和客户端提供了一个基础框架来使用相同的本地化配置文件。

Xml和资源文件有各自的优缺点，建议使用xml文件作为可重用的模块，因为xml更容易添加新语言模块的代码。此外，如果你使用xml，建议通过创建日期排序而不是通过文本名称来排序。当有人翻译为另外一种语言时，他可以更容易地看到哪些新添加的文本。

你也可以创建自己的本地化配置文件并集成到ABP中。

## 6.4 ABP表现层 - 导航栏

每一个WEB应用程序都有导航菜单，Abp也为用户提供了通用的创建和显示菜单方式。

### 6.4.1 创建菜单

一个应用程序可能包含不同的模块，而每个模块都可能它自己的菜单项。在Abp中，需要创建一个派生自NavigationProvider的类来定义一个菜单项。

假设我们有一个这样的主菜单：

- Tasks
- Reports
- Administration 1 User Management 2 Role Management

由上可知，Administration菜单项有两个子菜单项。对应的生成方法如下：

```
public class SimpleTaskSystemNavigationProvider : NavigationProvider
{
    public override void SetNavigation(INavigationProviderContext context)
    {
        context.Manager.MainMenu
            .AddItem(
                new MenuItemDefinition(
                    "Tasks",
                    new LocalizableString("Tasks", "SimpleTaskSystem"),
                    url: "/Tasks",
                    icon: "fa fa-tasks"
                )
            ).AddItem(
                new MenuItemDefinition(
                    "Reports",
                    new LocalizableString("Reports", "SimpleTaskSystem"),
                    url: "/Reports",
                    icon: "fa fa-bar-chart"
                )
            ).AddItem(
                new MenuItemDefinition(
                    "Administration",
                    new LocalizableString("Administration", "SimpleTaskSystem"),
                    icon: "fa fa-cogs"
                ).AddItem(
```

```
        new MenuItemDefinition(
            "UserManagement",
            new LocalizableString("UserManagement", "SimpleTaskSystem"),
            url: "/Administration/Users",
            icon: "fa fa-users",
            requiredPermissionName: "SimpleTaskSystem.Permissions.UserManagement"
        )
    ).AddItem(
        new MenuItemDefinition(
            "RoleManagement",
            new LocalizableString("RoleManagement", "SimpleTaskSystem"),
            url: "/Administration/Roles",
            icon: "fa fa-star",
            requiredPermissionName: "SimpleTaskSystem.Permissions.RoleManagement"
        )
    );
}
}
```

MenuItemDefinition可以有一个唯一的名字，一个用于本地化显示的名字，一个url和一个icon，此外，菜单项可能需要与特定用户权限相结合（相关权限系统正在开发，暂时还没有说明文档）。

InavigationProviderContext方法能够获取现有的菜单项、添加菜单或菜单项。因此，不同的模块可以添加各自的菜单。

创建完成导航后，还需要在对应模块预初始化时注册到Abp配置文件中：

```
Configuration.Navigation.Providers.Add<SimpleTaskSystemNavigationProvider>();
```

## 6.4.2 显示菜单

IuserNavigationManager可以注入、获取和显示菜单。可以在服务器端创建菜单。

Abp自动生成的javascript API使得用户能够在客户端获取菜单，对应的方法和对象在命名空间abp.nav中。例如，在客户端使用abp.nav.menus.MainMenu可以用来获取主菜单。

以上文档展示了Abp如何给用户创建和显示菜单，你可以尝试建立模版、查看源代码以便更深入的学习。

## 6.5 ABP表现层 - 异常处理

在 web 应用程序中，异常通常是在 MVC Controller actions 和 Web API Controller actions 中被处理的。当异常发生时，应用程序用户被以某种方式告知该错误和该错误产生的可选原因（就是列举出产生该异常的多种原因，产生错误的原因可能是列举出的一种也可能是多种。）

如果一个常规的HTTP请求产生错误，那么一个错误页面会展示。如果AJAX请求产生错误，服务器端会发送错误消息到客户端，然后客户端处理并显示该错误给用户。

在所有的Web请求中处理异常是一个单调乏味并且重复性的工作。然而在ABP中，你几乎不需为任何异常的指定明确的异常处理，ABP会自动的记录这些异常并且以适当的格式做出响应返回到客户端。也就是说，在客户端处理这些响应并且将错误详细显示给客户。

### 6.5.1 开启错误处理

为了开启错误处理，customErrors 必须设置如下：

```
<customErrors mode="On" />
```

也可以设置 'RemoteOnly' 如果你不想在本地处理这些错误。

### 6.5.2 非Ajax请求

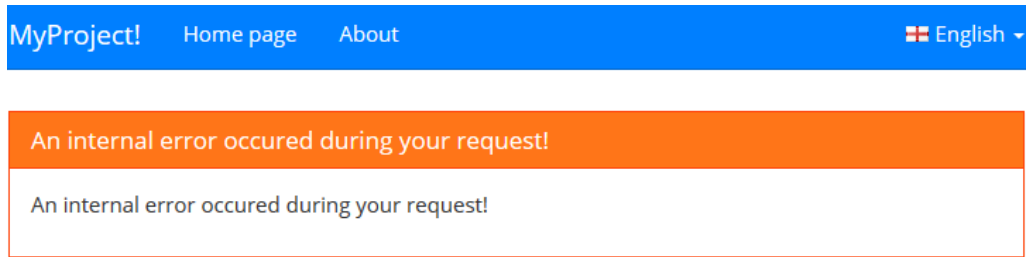
如果不是Ajax 请求，那么将会显示一个错误页面。

#### 1. 显示异常(Showing exceptions)

MVC Controller action抛出了一个异常，如下所示：

```
public ActionResult Index()
{
    throw new Exception("A sample exception message...");
}
```

当然，这个异常能够被指定调用的action中的另外的方法抛出。ABP处理这个异常，记录异常信息并且显示'Error.cshtml' 视图。你能够自定义这个视图来显示该错误。example 错误视图（这个视图是ABP中缺省错误视图模板）。



ABP 隐藏了异常的详细信息，而是向用户展示了一个标准的（本地化的，友好化的）错误信息。除非你明确指定抛出一个 `UserFriendlyException` 异常。

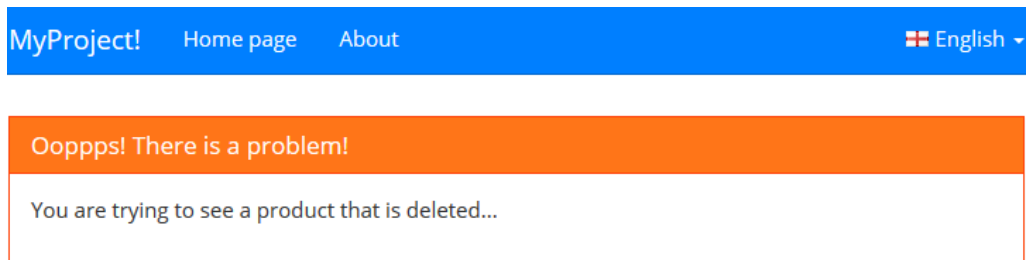
## 2. 用户异常友好化（`UserFriendlyException`）

`UserFriendlyException`是一个特殊的异常类型，被用来直接的显示给用户。请看下面示例：

```
public ActionResult Index()
{
    throw new UserFriendlyException("Ooppps! There is a problem!", "You are trying to see a product that is deleted..");
}
```



ABP记录这个异常并且不隐藏这次的异常信息：



所以，如果你想显示一个特殊的错误信息给用户，你只需要抛出一个 `UserFriendlyException`（或者一个派生自该异常类的类型，也就是说继承这个异常类的子类）。

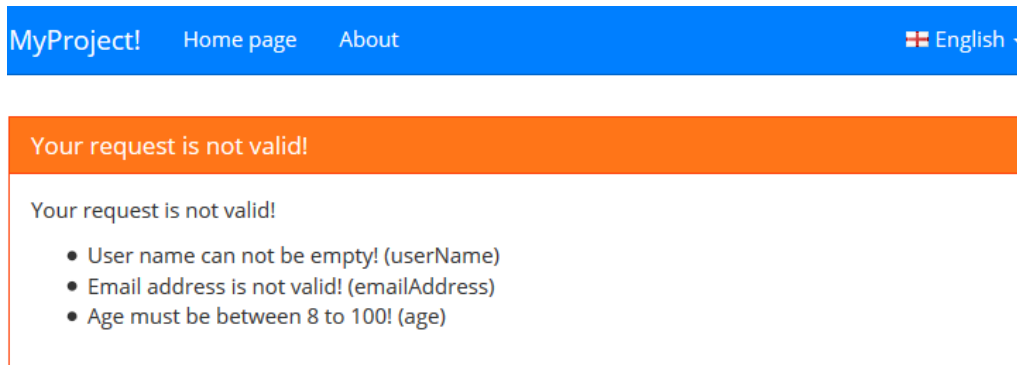
## 3. 错误模型(Error model)

ABP传递一个 `ErrorViewModel` 对象给Error视图：

```
public class ErrorViewModel
{
    public AbpErrorInfo ErrorInfo { get; set; }
    public Exception Exception { get; set; }
}
```

`ErrorInfo` 包含了能够显示给客户的详细的异常信息。`Exception` 对象就是那个被抛出的异常。你能够核实异常并且附加自定义信息来显示，如果你想这样做的话。例如：我们能够核实该异常是否是一个 `AbpValidationException`。





### 6.5.3 AJAX请求

如果请求是一个AJAX请求，ABP会返回一个JSON对象到客户端。ASP.NET MVC Controllers 和 ASP.NET Web API Controllers 也是这么处理的。以JSON方式返回一个异常信息，请看下面示例：

```
{
  "targetUrl": null,
  "result": null,
  "success": false,
  "error": {
    "message": "An internal error occurred during your request!",
    "details": "..."
  },
  "unAuthorizedRequest": false
}
```

success: false 表示有一个错误发生。error对象提供了错误信息和错误的详细描述。

当你在客户端用ABP的基础设施来做一个AJAX请求时，它会用message API自动的处理这个JSON对象并且显示错误信息给用户。更多信息请参照AJAX API 和dynamic web api layer。

### 6.5.4 异常事件

当ABP处理任何Web请求的异常时，它会触发AbpHandledExceptionData事件，当然你必须注册该事件，并且写相应的处理代码。详细信息请参照eventbus documentation。

## 6.6 ABP表现层 - Javascript函数库

ASP.NET Boilerplate的js库提供了一些让javascript开发更方便的方法和对象,以下介绍一下库中的API列表。

### 6.6.1 Ajax

现代的应用经常会使用AJAX，尤其是单页应用，几乎是和服务端通信的唯一手段，执行AJAX通常会有以下步骤：

在客户端，你需要提供一个URL，选择一个和服务端通信的方法（GET,POST,PUT,DELETE）。在请求完成后执行回调函数，请求结果可更是成功或失败，失败时你需要给出提示，成功时你需根据返回值执行操作。通常情况下，在请求开始时，你需要给出类似正在处理或者相关的繁忙等待信息（如页面遮盖），请求完成后恢复。

服务端接收到请求后，对请求参数进行验证，执行服务端代码，如果发生错误或者验证失败，应给出具体的原因，成功时返回客户端想要的的数据。

ABP服务端支持标准的ajax的请求/输出。建议大家使用abp.jquery.js中提供的ajax请求方法，这个方法基于jquery的ajax方法，可以自动处理服务端的异常信息，当然，如果你对js很熟练的话，也可以根据自己的需要写ajax。

ASP.NET Boilerplate的ajax请求实例：

```
//构建要传输的参数对象
var newPerson = {
  name: 'Douglas Adams',
  age: 42
};
//调用abp的ajax方法
abp.ajax({
  url: '/People/SavePerson',
  data: JSON.stringify(newPerson) //转换成json字符串
```

```
}).done(function(data) {  
    abp.notify.success('created new person with id = ' + data.personId);  
});
```

你也可以使用jquery的ajax方法调用,但是需要设置一下默认请求参数, `dataType` 设置为 'json', `type` 设置为 'POST' and `contentType` 设置为 'application/json', 在发送请求时需要将js对象转换成json字符串, 和\$.ajax一样, 你也可以传递参数覆盖 `abp.ajax` 的默认参数 `abp.ajax` 返回一个promise类型, 你可以链式编程写如下的方法:

```
.done() //成功,  
.fail() //失败,  
.then() //回调嵌套。
```

下面的一个简单的例子展示ajax请求 `PeopleController` 的 `SavePerson` 方法, 在 `.done()` 中可以获取到服务端创建记录成功后返回的记录id。

```
public class PeopleController : AbpController  
{  
    [HttpPost]  
    public JsonResult SavePerson(SavePersonModel person)  
    {  
        //TODO: save new person to database and return new person's id  
        //TODO: 创建一个新的person记录并返回此记录的id  
        return Json(new {PersonId = 42});  
    }  
}
```

`SavePersonModel` 包含name,age等属性. `SavePerson` 上标记了 `HttpPost` 特性 `abp.ajax`默认以 `POST` 方式请求. 返回值被简化成了一个匿名对象。

## 6.6.2 AJAX 返回值(AJAX return messages)

我们直接返回了一个匿名对象, ABP 通过 `MvcAjaxResponse` 类型包装了返回值. 实际的返回值类型如下:

```
{  
    "success": true, //正确处理标志  
    "result": {  
        "personId": 42 //返回的数据  
    },  
    "error": null, //如果发生错误, result为null, 此处为错误信息的对象, 包含message和details两个属性  
    "targetUrl": null, //可以提供一个url供客户端重定向, 例如自动构建下一页的url  
    "unAuthorizedRequest": false //是否通过了授权, 如果返回true, 客户端应重新登录  
}
```

如果你继承了 `AbpController`, `Json`方法返回的对象总会被这样包装, 如果未发生错误, 你在 `abp.ajax` 的 `done(function(result,data){})` 中, 第一个参数 `result` 是 `{ "personId": 42 }` 对象, `data` 是原始对象, `WebApi` 中继承 `AbpApiController` 也是同样的机制。

## 6.6.3 错误处理(Handling errors)

返回值如下:

```
{  
    "targetUrl": null,  
    "result": null,  
    "success": false, //代表出现异常  
    "error": {  
        "message": "An internal error occured during your request!", //未捕捉到的异常, 通常为系统异常, 会自动记录日志, 具体提示  
        "details": "... " //发生异常时默认会调用abp.message.error函数, 你可以在abp.jquery.js修改, 统一处理错误信息。  
    },  
    "unAuthorizedRequest": false  
}
```

## 6.6.4 WrapResult 和 DontWrapResult 特性

你可以用 `WrapResult` 和 `DontWrapResult` 特性来控制某个Action或者某个Controller的所有Action的返回结果。

## 1. ASP.NET MVC Controllers

如果返回的类型是 **JsonResult** (或者异步Action的Task)ABP默认会将Action的返回内容进行包装。你可以改变这个，只要在Controller的Action上面使用特性: **WrapResult** 或者 **DontWrapResult**; 示例如下:

```
public class PeopleController : AbpController
{
    [HttpPost]
    [WrapResult(WrapOnSuccess = false, WrapOnError = false)]
    public JsonResult SavePerson(SavePersonModel person)
    {
        //TODO: save new person to database and return new person's id
        return Json(new {PersonId = 42});
    }
}
```

为了方便开发,我们可以使用特性 **[DontWrapResult]**, 并且最终结果与上面所示相同。

## 2. ASP.NET Web API Controllers

除了 **AjaxResponse** 的返回类型, ABP 默认是没有包装 Web API的Action的返回结果。如果需要你可以给Controller的Action添加特性: **WrapResult**。

### 6.6.5 动态WebAPI (Dynamic Web API Layer)

此处会根据Services动态生成WebAPI调用函数:

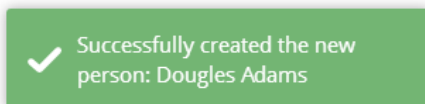
```
//通常我们使用ajax会按照如下写法, 做一个简单的封装来重用ajax, 此处框架可以帮你生成简单的调用方法
var savePerson = function(person) {
    return abp.ajax({
        url: '/People/SavePerson',
        data: JSON.stringify(person)
    });
};
//调用时你需要构建参数
var newPerson = {
    name: 'Douglas Adams',
    age: 42
};
//直接调用方法, 如何生成上面的调用方法可以参考源码中的Abp.Web.Api项目中/ WebApi/ Controllers/ Scripting/ jQuery下的实现
savePerson(newPerson).done(function(data) {
    abp.notify.success('created new person with id = ' + data.personId);
});
```

### 6.6.6 通知

通知会显示在右下角, 稍后自动消失

```
abp.notify.success('a message text', 'optional title');
abp.notify.info('a message text', 'optional title');
abp.notify.warn('a message text', 'optional title');
abp.notify.error('a message text', 'optional title');
```

通知API是依赖于toastr库, 你需要在项目中引用toastr的js和css, 然后引用ABP项目的abp.toastr.js, notify.success调用后的样子:



你可以运行样板项目, 在浏览器的控制台测试这几种提示消息, 另外, 当ajax出现异常时, 你可以修改abp.jquery.js的源文件, 来调用abp.notify.error方法实现友好的提示信息。

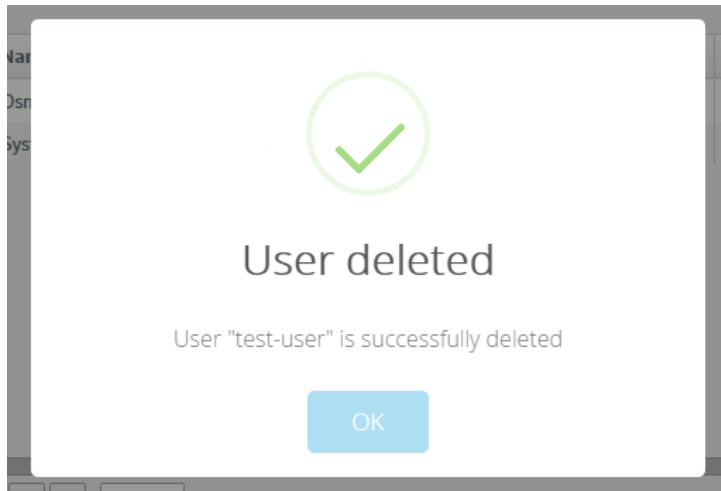
如果你有其他的的通知插件也可以使用, 引用相应的js库就可以了, 提示消息的js是可选的。

### 6.6.7 消息

用于向用户显示对话框，展示消息或者得到用户的确认,ABP默认采用的sweetalert库实现的对话框信息，使用时你需要引用sweetalert的样式和js，并且引用abp.sweet-alert.js就可以使用下列API了：

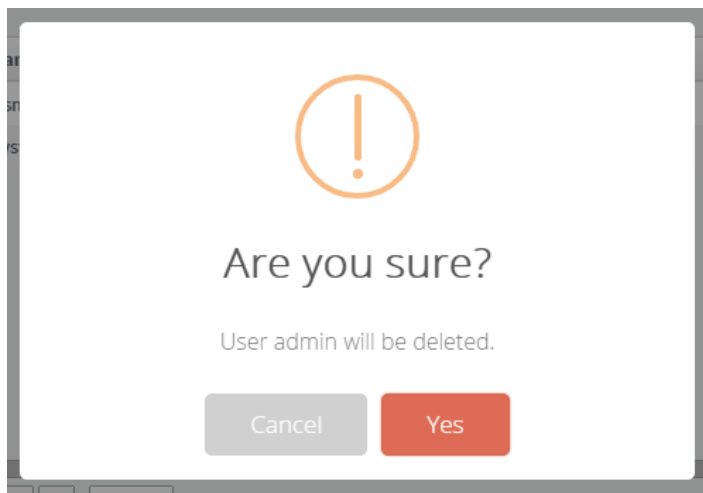
```
abp.message.info('some info message', 'some optional title');
abp.message.success('some success message', 'some optional title');
abp.message.warn('some warning message', 'some optional title');
abp.message.error('some error message', 'some optional title');
```

调用abp.message.success会展示如下的对话框:



Confirmation 确认对话框:

```
abp.message.confirm(
    'User admin will be deleted.', //确认提示
    'Are you sure?', //确认提示（可选参数）
    function (isConfirmed) {
        if (isConfirmed) {
            //...delete user 点击确认后执行
        }
    }
);
```



默认ABP的js库中可能会引用到消息API，比如ajax调用失败会调用abp.message.error。

## 6.6.8 用户界面的繁忙提示

ABP提供了设置页面的某部分繁忙的API。

### 1. UI Block API

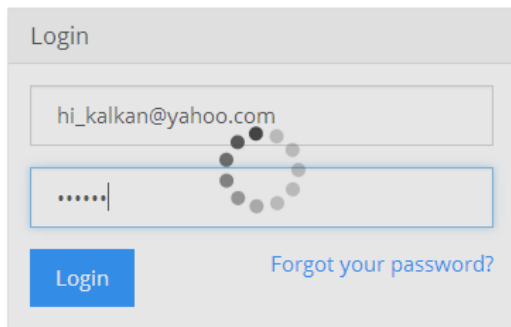
设置一个半透明层，阻止点击页面元素，可以覆盖局部或者整个页面，例子如下：

```
abp.ui.block(); //覆盖整个页面
abp.ui.block($('#MyDivElement')); //覆盖指定元素, 可以把jquery对象作为参数
abp.ui.block('#MyDivElement'); //或者直接使用选择器参数
abp.ui.unblock(); //整个页面解除覆盖
abp.ui.unblock('#MyDivElement'); //指定元素解除覆盖
```

UI Block API使用blockUI这个js库来实现效果的, 如果使用这个api需要在页面引用blockUI的js库和abp.blockUI.js文件。

UI Busy API 指示页面繁忙的API, 如ajax请求中:

```
abp.ui.setBusy('#MyLoginForm');
abp.ui.clearBusy('#MyLoginForm');
```



第一个参数可以直接使用jquery选择器如'#id'或者使用jquery对象如\$('#id'),如果传null或者'body'则标记整个页面为繁忙状态, 第二个参数可以接收一个promise, promise完成后会自动解除页面繁忙状态。

```
abp.ui.setBusy(
    $('#MyLoginForm'),
    abp.ajax({ ... }) //返回值是promise, 如果需要了解promise的更多信息, 可以参考jQuery的 Deferred
);
```

UI Busy API 使用的是spin.js, 你需要在页面中引用spin.js和abp.spin.js。

## 6.6.9 事件总线

在客户端事件总线用来触发注册过的全局事件。

### 1. 简介

**Pub/Sub** 事件模型被广泛的用在客户端。ABP包含了一个简单的全局事件总线用来注册和触发事件。

### 2. 注册事件

你可以使用 **abp.event.on** 来注册一个全局事件。示例如下:

```
abp.event.on('itemAddedToBasket', function (item) {
    console.log(item.name + ' is added to basket!');
});
```

首先, 参数: 事件名称必须是唯一的。另一个参数是 回调函数, 当指定的事件被触发后将调用该参数。

你可以使用 **abp.event.off** 方法来卸载已注册的事件。注意: 为了能够卸载指定的事件, 应该提供相同的事件函数。正如上面的示例所展示的, 你应该将回调函数设置为一个变量, 然后在 **on**和**off** 中使用它。

### 3. 触发事件

**abp.event.trigger** 被用来触发全局事件。触发一个已注册的事件的代码如下:

```
abp.event.trigger('itemAddedToBasket', {
    id: 42,
    name: 'Acme Light MousePad'
});
```

第一个参数是该事件的唯一名称。第二个是（可选的）事件参数。你可以添加任何数量的参数，并且在回调方法中获得它们。

### 6.6.10 Js日志接口

这个主要是对浏览器`console.log(...)` 进行的包装，可以支持所有浏览器，例子如下：

```
abp.log.debug('...');
abp.log.info('...');
abp.log.warn('...');
abp.log.error('...');
abp.log.fatal('...');
```

你可以通过设置`abp.log.level`来控制日志输出，和服务端一样，如设置了`abp.log.levels`为`INFO`时就不会输出`debug`日志了，你也可以根据你的需要定制重新这些API。

### 6.6.11 Javascript公共方法

ABP提供了一些常用的公共方法。

#### 1. 创建命名空间别名(`abp.utils.createNamespace`)

通过创建命名空间让js方法分类更加明确，使用更加方便，下面是例子：

```
//创建或者获取命名空间
abp.utils = abp.utils || {};
abp.utils.strings = abp.utils.strings || {};
abp.utils.strings.formatting = abp.utils.strings.formatting || {};
//在命名空间中增加一个方法
abp.utils.strings.formatting.format = function() { ... };
你可以向下面一样用
//创建命名空间别名
var formatting = abp.utils.createNamespace(abp, 'utils.strings.formatting');
//在formatting命名空间下增加/修改一个方法
formatting.format = function() { ... };
```

别名简化了以前长长的名字，需要注意的是，第一个参数是必须存在的根命名空间。

格式化字符串(`abp.utils.formatString`)

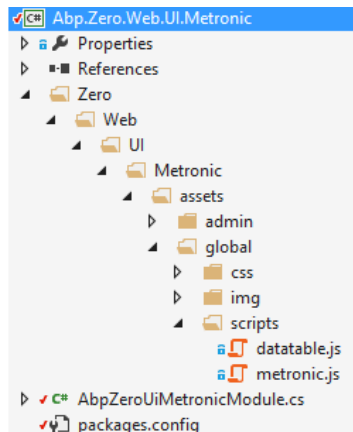
和C#的`string.Format`一样的用法

```
var str = abp.utils.formatString('Hello {0}!', 'World'); //str = 'Hello World!'
var str = abp.utils.formatString('{0} number is {1}.', 'Secret', 42); //str = 'Secret number is 42'
```

## 6.7 ABP表现层 - 嵌入资源文件

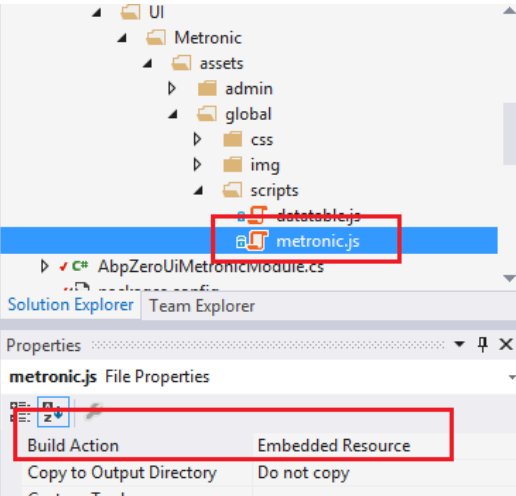
### 6.7.1 简介

在一个WEB应用程序中，有JavaScript、css、xml等客户需要使用的文件。一般添加到web项目中并作为分离的文件发布。有时候，我们需要将这些文件打包（类库、dll).ABP提供了ifrastructure使这种任务更容易完成。





我们要让这些文件在web应用程序中可用。上线，应当构建嵌入式资源文件。选择metronic.js，属性窗口（F4快捷方式）修改为资源文件



需要作为资源文件的都照此修改。

6.7.2 显示资源文件

ABP通过一行代码显示资源文件：

```
WebResourceHelper.ExposeEmbeddedResources("AbpZero/Metronic", Assembly.GetExecutingAssembly(), "Abp.Zero.Web.UI.Metrc
```

代码一般放在模块初始化方法中。参数解释：

- 第一个参数定义了文件的根文件夹。根名称空间匹配。
- 第二个参数定义了集合包含文件。在这个例子中,通过集合包含这些代码。你也可以通过任何集合包含嵌入的资源。
- 最后一个在集合中定义了文件的根名称空间。这是“默认名称空间”和“文件夹”。默认名称空间通常是组装的名字一样但可以改变组件的属性。在这里,默认名称空间是Abp(我改变它),所以Metronic文件夹的名称空间是“Abp.Zero.Web.UI.Metronic”。

6.7.3 使用嵌入的文件

使用嵌入文件很简单：

```
<script type="text/javascript" src="~/AbpZero/Metronic/assets/global/scripts/metronic.js"></script>
```

ABP能够理解这是一个嵌入文件，你也可以在Razor视图中使用ABP的HtmlHelper：

```
@Html.IncludeScript("~/AbpZero/Metronic/assets/global/scripts/metronic.js")
```

页面显示如下：

```
<script src="/AbpZero/Metronic/assets/global/scripts/metronic.js?v=635438748506909100" type="text/javascript"></scrip
```

唯一需要注意的是 v=635438748506909100 参数，这可以防止浏览器运行错误的缓存脚本。这个值的变化只有当dll重建（实际上，这是最后写文件的时间），如果改变，浏览器不缓存。因此，建议使用包含脚本的方式。这同样使用于非嵌入式的物理文件，和包含css文件的方法。

5.1 5.2 5.4 5.6 翻译 NoZero 5.3 菜刀 5.5 Carl## 6.8 ASP.NET Core

6.8.1 简介

这篇文档是对ABP中集成的ASP.NET Core的描述。ASP.NET 集成是被实现在包 Abp.AspNetCore nuget package。

6.8.2 Startup Template

你可以从[Startup Template](#)来创建你的项目，该模板是一个空置且简单的web项目，它预先集成且配置好一切可以与ABP框架一起运行。

### 6.8.3 Configuration

#### Startup Class

为了集成ABP到ASP.NET Core，我们应该在 Startup Class 中做一些配置，如下所示：

```
public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        //...

        //配置Abp和依赖注入，在最后调用
        return services.AddAbp<MyProjectWebModule>(options =>
        {
            //配置Log4Net(可选的)
            options.IocManager.IocContainer.AddFacility<LoggingFacility>(
                f => f.UseLog4Net().WithConfig("log4net.config")
            );
        });
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
    {
        //初始化ABP框架和所有其他模块，这个应该首先被调用
        app.UseAbp();

        //...
    }
}
```

#### Module Configuration

你可以使用[启动配置](#)来配置AspNet Core模块(在你模块的PreInitialize方法中使用 `Configuration.Modules.AbpAspNetCore()`)

### 6.8.4 Controllers

在ASP.NET Core中控制器可以是任何类类型。它没有限制控制器类必须派生自Controller类，以Controller结尾的类默认认为是MVC Controller，例如：`ProductController`。你也可以使用[Controller]特性添加到任何类上，使该类作为一个控制器。这就是ASP.NET Core MVC的工作方式。详细了解请查看[ASP.NET Core文档](#) 当然已经有贡献者开始翻译该文档<https://github.com/dotnetcore/aspnetcore-doc-cn>，详细可以在cnblog中搜索。

如果你要使用web层类如：`HttpContext`或者返回一个视图，最好的方式是继承 `AbpController` (该类也是派生自MVC的Controller)，如果你在创建一个和对象一起工作的API Controller，你可以考虑创建一个POCO Controller类或者可以使用应用层服务作为控制器，如下所述：

#### Application Services as Controllers

ABP提供了基础设施来创建[应用服务](#)。如果你想你的应用服务作为控制器暴露给远程客户端(正如前面文档描述的使用[动态WebAPI](#))，你可以很容易的实现，只需要在你模块的 `PreInitialize` 方法中做一个简单的配置，如：

```
Configuration.Modules.AbpAspNetCore().CreateControllersForAppServices(typeof(MyApplicationModule).Assembly, moduleNan
```

`CreateControllersForAppServices` 方法会取得指定程序集并转换该程序集下的所有的应用服务作为MVC Controller。你可以使用 `RemoteService` 特性来enable/disable 方法，类或者接口。

当应用服务被转换为MVC Controller时，默认路由模板是：`/api/services/<module-name>/<service-name>/<method-name>`。例如：如果ProductAppService中定义了一个方法，那么它的URL会是：`/api/services/app/product/create` (假设模块名称是app)。

如果 `useConventionalHttpVerbs` 被设置为 `true` (默认就是这个)。那么通过命名约定对于应用服务的方法所使用的HTTP谓词会被断定为：

- **Get**: 如果方法的名字是以 `Get` 开头

- **Put**: 如果方法的名字是以 **Put** 或者 **Update** 开头
- **Delete**: 如果方法的名字是以 **Delete** 或者 **Remove** 开头
- **Post**: 如果方法的名字是以 **Post**, **Create** 或者 **Insert** 开头
- **Path**: 如果方法的名字是以 **Path** 开头
- 否则 **Post** 被作为Http谓词的默认设置

你可以使用任何的 ASP.NET Core 特性来改变HTTP方法或者Action的路由(当然, 添加ASP.NET Core package的引用是必须的)。

注意: 首先, 动态WebAPI需要应用服务层的服务类必须实现服务层的接口。但是对于ASP.NET Core 的集成不是必须的。所以MVC attributes应该被添加到服务类上, 即使你实现了该接口。

## 6.8.4 Filters

ABP为AspNet Core预先构建了一些过滤器, 这些过滤器被默认添加到所有控制器的所有的Action上。

### Authorization Filter

**AbpAuthorizationFilter** 被集成到了 [身份授权认证](#) 和 [功能管理](#)。

- 你可以在Action或者Controller上使用 **AbpMvcAuthorize** 特性, 在Action执行之前来检查给定的权限是否开启
- 你可以在Action或者Controller上使用 **RequiresFeature** 特性, 在Action执行之前来检查给定的功能是否开启
- 你可以在Action或者Controller上使用 **AllowAnonymous**(或者在应用层使用**AbpAllowAnonymous**) 特性来进行匿名访问, 避免身份以及授权认证检测

### Audit Action Filter

**AbpAuditActionFilter** 被集成到[审计日志](#), 默认记录所有对Action的请求。你可以在Action或者Controller上使用特性: **Audited** 和**DisableAuditing** 来控制是否记录日志。

### Validation Action Filter

**AbpValidationActionFilter** 被集成到[数据传输对象验证](#) 并且自动为所有Action进行输入验证。除此之外ABP还内置了validation & normalization。它也会检查MVC的 **Model.IsValid** 属性, 如果Action的输入有无效的输入值那么它会抛出validation exception。

你可以使在Action和Controller上用特性: **EnableValidation**和**DisableValidation** 控制validation。

### Unit of Work Action Filter

**AbpUowActionFilter** 被集成到[工作单元](#), 它会在Action执行之前自动开启一个工作单元并且在Action执行完成后完成工作单元(如果没有异常抛出)。

你可以使用特性 **UnitOfWork** 来控制Action的UOW行为。你也可以使用启动配置来改变所有Action上的默认的工作单元特性设置。

### Exception Filter

**AbpExceptionHandler** 被用来处理来自Controller的Action的异常。它处理并且记录异常信息且返回包裹的响应到客户端。

- 它仅处理object result, 不包括 view result。所以, action返回的任何对象, JsonResult 或者 ObjectResult 将会被处理。Action返回的任何实现自ActionResult视图或者其他结果类型不会被处理。最好是使用定义在 Microsoft.AspNetCore.Diagnostics package中内置的 **UseExceptionHandler** 扩展方法来处理视图异常。
- 可以在方法或者类上面使用 **WrapResult**和**DontWrapResult** 特性来改变异常处理和日志行为。

### Result Filter

**AbpResultFilter** 被用来包装Action的返回结果, 如果该Action执行成功的话。

- 它仅对JsonResult, ObjectResult和其它不是实现ActionResult接口对象进行返回结果包装, 如果你的Action正在返回一个视图或者其它结果的类型, 它不会进行包装。
- 在方法或者类上面使用 **WrapResult**和**DontWrapResult** 特性可以enable/disable包装。
- 你可以在启动配置里面对包装结果的默认行为进行更改

## 6.8.5 Model Binders

**AbpDateTimeModelBinder** 被用来标准化输入时间(可空类型的时间), 实际调用 **Clock.Normalize** 方法来实现。

### 6.8.6 Views

MVC视图继承自 **AbpRazorPage**，它自动实现了大多数基础设置的注入(例如：**LocalizationManager**, **PermissionChecker**, **SettingManager**等等)。它也有一些便捷的方法如 **L()** 资源本地化。

### 6.8.7 Client Proxies

ABP能自动的为所有的MVC Controllers创建脚本代理(不仅仅是服务层应用)。它默认为控制器的应用层服务创建脚本代理。你可以在任何的MVC Controller上使用 **[RemoteService]** 特性来创建客户端代理。脚本代理在运行时被动态的创建。你只需要在页面(*Layout.cshtml*)上添加如下代码：

```
<script src="~/AbpServiceProxies/GetAll?type=jquery" type="text/javascript"></script>
```

当前仅生成了JQuery代理。我们可以使用脚本来调用MVC Action，如下所示：

```
abp.services.app.product.create({
    name: 'My test product',
    price: 99
}).done(function(result){
    //...
});
```

### 6.8.7 集成测试

对于ASP.NET Core的集成测试是相当容易的详情请查阅文档。ABP遵循这个指导并且在**Abp.AspNetCore.TestBase** package 提供了 **AbpAspNetCoreIntegratedTestBase** 类。这使集成测试变得更容易。

最好是在启动模板中通过检查集成测试来查看实际应用情况。

## 6.9 CSRF和XSRF保护

### 6.9.1 简介

**Cross-Site Request Forgery** (CSRF) 跨站请求伪造是一种攻击，发生在具有恶意的网站，email，blog，即时消息，或者程序导致用户的web浏览器在一个受信用的网站去执行了某个有害的行为来获取当前用户的认证信息，详细了解请点击[这里](#)脑补。

在ASP.NET Web API官网也有如何实现的简短描述。

ABP框架尽可能的简化且自动化CSRF保护。**启动模板** 实现了开箱即用的预配置。在这篇文档中，我们会展示它是如何集成到ASP.NET，以及它是如何工作的。

#### Http 谓词

对于 HTTP谓词是 **GET**，**HEAD**，**OPTIONS**以及**TRACE** 的Action，我们不需要进行CSRF保护，因为它们是无副作用的操作(不会更改数据库)。ABP假定(ABP仅对这些谓词实现了反伪造保护，如：**POST**，**PUT**，**PATH**以及**DELETE**)你能够使用一些已定义的特性改变这些行为(应该是说将 **[POST]** 等特性加装在以Get等开头的方法上)。

### 6.9.2 ASP.NET MVC

#### Features

正如你所了解到的ASP.NET MVC有自己内置的AntiForgery功能，但是它具有以下不完美的地方：

- 需要对所有的Actions添加 **ValidateAntiForgeryToken** 才能实现保护。我们可能会忘记添加它到需要执行保护的Action上。
- **ValidateAntiForgeryToken** 特性仅检查HTML表单中的 **\_\_RequestVerificationToken** 字段。这在 **AJAX** 请求中使用它是很难且不可能的，特别是在你发送的content-type是 **application/json** 的时候。在AJAX请求中，通常的做法是在 **request header** 中添加token。
- 在AJAX请求中我们需要取得token并且使用它，但是在脚本代码中很难取得verification token(特别是你没有在.cshtml文件中写脚本)。
- 即使我们能够在脚本中得到token，我们应该手动的添加token到每个请求头中。

ABP做了以下一些事情克服了这些困难：

- 不在需要对这些Action 如: POST, PUT, PATH以及DELETE 添加 **ValidateAntiForgeryToken** 特性, 因为它们会自动的保护起来(通过 **AbpAntiForgeryMvcFilter** 过滤器)。对于大多数情况自动保护已足够好。但是你也可以使用 **DisableAbpAntiForgeryTokenValidation** 特性来禁用对某个Action或者Controller的保护, 你也可以使用特性 **ValidateAbpAntiForgeryToken** 对任何你需要的Action/Controller来开启保护。
- 在脚本中提供了 **abp.security.antiForgery.getToken()** 函数来取得token, 即使你不需要它。
- 对所有的AJAX请求自动的添加 anti forgery token 到请求头中。

因此, 相对于ASP.NET MVC 自带AntiForgery功能, 它的几乎是无缝的工作。

## 集成

开始模板已经对CSRF进行了开箱即用的集成。如果你需要手动添加它到你的项目(你的项目先于我添加该项功能的时候就已经创建), 请跟随下面指南:

### Layout View

我们应该添加下面代码到我们的布局视图:

```
@{
    SetAntiForgeryCookie();
}
```

因此, 所有使用了该视图的页面都会包含它。这个方法被定义在ABP视图的基类中。它会创建且设置token到cookie中, 并且能在客户端脚本运行。如果你有多个布局视图, 请把该代码添加到所有的布局视图中。

这就是我们应该对ASP.NET应用所要做的。所有的AJAX请求会自动的工作。但是我们应该在那些不需要通过AJAX来提交POST请求的表单中使用 **@Html.AntiForgeryToken()** 方法(并且对相信的Action我们不需要添加 **ValidateAbpAntiForgeryToken** 特性)。

### Configuration

XSRF保护默认是开启的。你可以在模块的 **Preinitialize** 方法中禁用。如:

```
Configuration.Modules.AbpWeb().AntiForgery.IsEnabled = false;
```

你也可以使用 **Configuration.Modules.AbpWebCommon().AntiForgery** 来配置token和cookie的名称。

注意: 如果使用了伪造验证, 如果使用Swagger Ui来调试动态WebApi, 请参考 Swagger UI集成的文档。

## 6.9.3 ASP.NET Web API

### Features

ASP.NET Web API 没有提供防伪造这样的机制。当时ABP对ASP.NET Web API提供了CSRF保护的基础设施并且自动完成这个功能。

## 集成

与 **ASP.NET MVC Clients** 集成

如果你在MVC项目中使用Web API, 不需要任何额外的配置。即使你的Web API层是self-hosting在另外的进程中。不需要任何配置, 只需配置好来自MVC应用的AJAX请求。

与其他**Clients**集成

如果你的客户端是不同种类的应用(没有如前面所述: 在一个独立的angularjs中使用 **SetAntiForgeryCookie()** 方法)。那么你应该提供一个设置反伪造token的cookie。一个可能的方式是像下面一样创建一个Api Controller:

```
using System.Net.Http;
using Abp.Web.Security.AntiForgery;
using Abp.WebApi.Controllers;

namespace AngularForgeryDemo.Controllers
{
    public class AntiForgeryController : AbpApiController
    {

```

```

        private readonly IAbpAntiForgeryManager _antiForgeryManager;

        public AntiForgeryController(IAbpAntiForgeryManager antiForgeryManager)
        {
            _antiForgeryManager = antiForgeryManager;
        }

        public HttpResponseMessage GetTokenCookie()
        {
            var response = new HttpResponseMessage();

            _antiForgeryManager.SetCookie(response.Headers);

            return response;
        }
    }
}

```

那么你可以从客户端调用该action来设置cookie。

### 6.9.3 ASP.NET Core

#### Features

与之前的版本(ASP.NET MVC 5.x)相比,ASP.NET Core MVC 有一个很好的反伪造机制:

- 它有个 **AutoValidateAntiforgeryTokenAttribute** 类会自动的对所有的 **POST**, **PUT**, **PATH**以及**DELETE** Action自动实施反伪造验证。
- 它有 **ValidateAntiforgeryToken**和**IgnoreAntiforgeryToken** 特性来控制token验证。
- 如果你没有禁用反伪造功能, 它会自动的添加反伪造安全token在html表单中。所以在大多数情况下你不需要调用 **@Html.AntiForgeryToken()**。
- 他可以从HTTP头或者Form表单的字段中读取请求的token。

ABP 添加了以下功能:

- 自动的为所有的AJXA请求添加了反伪造token。
- 在脚本中提供了 **abp.security.antiForgery.getToken()** 来取得token, 即使你不使用它。

#### 集成

开始模板已经对CSRF进行了开箱即用的集成。如果你需要手动添加它到你的项目(你的项目先于我添加该项功能的时候就已经创建), 请跟随下面指南:

#### Startup Class

首先, 当我们在MVC启动类中配置ConfigureServices时, 我们应该添加 **AutoValidateAntiforgeryTokenAttribute** 特性到全局过滤器中。

```

services.AddMvc(options =>
{
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute());
});

```

因此, 所有的MVC Actions将会对反伪造token进行自动验证(除了之前所描述的 **GET**, **HEAD**, **OPTIONS**以及**TRACE** 外)。

#### Layout View

我们应该在布局视图中添加下面的代码:

```

@using Abp.Web.Security.AntiForgery
@inject IAbpAntiForgeryManager AbpAntiForgeryManager
@{
    AbpAntiForgeryManager.SetCookie(Context);
}

```

因此, 所有使用了该视图的页面都会包含它。这个方法被定义在ABP视图的基类中。它会创建且设置token到cookie中, 并且能在客户端脚本运行。如果你有多个布局视图, 请把该代码添加到所有的布局视图中。



这就是我们应该为ASP.NET Core MVC应用所要做。所有的AJAX请求会自动的工作。对于非AJAX表单提交，如果你使用在你的表单中使用了asp-\* 的标签，ASP.NET Core会自动添加反伪造token字段。所以正常情况下你不需要调用@Html.AntiForgeryToken()。

## 6.9.4 客户端库

在所有的AJAX请求头中反伪造token应该被提供，正如我们前面所描述的。在这里我们将会看到具体实现。

### JQuery

abp.jquery.js 定义了AJAX拦截器，这可以对每个AJAX请求添加反伪造token到请求头中。使用abp.security.antiForgery.getToken() 方法来取得token。

### AngularJs

AngularJs会自动的添加反伪造token到所有的AJAX请求中，详细请查阅Angularjs的跨站请求伪造保护。ABP默认使用相同的cookie和头名称。所以AngularJs集成了开箱即用的功能。

### 其他库

如果你使用了其他的脚本库来实现AJAX请求，你有三个选择可以做：

#### Intercept XMLHttpRequest

因为所有的库都会使用脚本的原生AJAX对象：XMLHttpRequest 你可以像下面一样定义一个简单的拦截器将token添加到请求头中：

```
(function (send) {  
    XMLHttpRequest.prototype.send = function (data) {  
        this.setRequestHeader(abp.security.antiForgery.tokenHeaderName, abp.security.antiForgery.getToken());  
        return send.call(this, data);  
    };  
})(XMLHttpRequest.prototype.send);
```

#### Use Library Interceptor

一个好使用的库会提供拦截点给你如jQuery和angularjs。所以，你应该学习这些库的文档来了解如何拦截请求并操作请求头。

#### 手动添加请求头

最后一种选择，你可以使用 abp.security.antiForgery.getToken() 方法取得token并且手动的添加它到每个AJAX请求中。但是你可能不需要这么做，你可以按照上面所述来解决问题。

## Internals

你可能想知道ABP是如何处理它的。事实上，在前面提到的angularjs文档中，我们使用了相同的机制来描述。ABP存储token到cookie(如前所述)，并且使用该cookie来设置请求头。为了验证防伪造token，它也很好的集成到了ASP.NET，Web API和Core框架。## 7.1 ABP后台服务 - 后台作业和后台工人

### 7.1.1 简介

ABP提供了后台作业和后台工人，来执行应用程序中的后台线程的某些任务。

### 7.1.2 后台作业

由于各种各样的原因，你需要后台作业以队列和持久化的方式来排队执行某些任务。

例如：

- 用户等待执行一个长时任务。例如：某个用户按下了报表按钮生成一个需要长时间等待的报表。你添加这个工作到队列中，当报表生成完毕后，发送报表结果到该用户的邮箱。
- 重试创建并持久化任务操作，以保证任务的成功执行。例如：在后台作业中发送一封邮件，有些问题可能会导致发送失败(网络连接异常，或者主机宕机)；由于有后台作业以及持久化机制，在问题排除后，可以重试执行失败的任务，以保证任务的成功执行。

关于作业持久化 了解更多信息可以查看源码：Background Job Store



## 1. 创建后台作业

通过继承 **BackgroundJob<TArgs>** 类或者直接实现 **IBackgroundJob<TArgs>** 接口，我们可以创建一个后台作业。

下面是一个极其简单的后台作业示例：

```
public class TestJob : BackgroundJob<int>, ITransientDependency
{
    public override void Execute(int number)
    {
        Logger.Debug(number.ToString());
    }
}
```

后台作业中定义了一个方法：**Execute** 并有一个输入参数。正如示例中所示，参数被定义为泛型参数。

后台作业必须使用依赖注入进行注册，而实现 **ITransientDependency** 是最简单的方式。

下面我们将定义一个在后台队列中发送邮件的作业：

```
public class SimpleSendEmailJob : BackgroundJob<SimpleSendEmailJobArgs>, ITransientDependency
{
    private readonly IRepository<User, long> _userRepository;
    private readonly IEmailSender _emailSender;

    public SimpleSendEmailJob(IRepository<User, long> userRepository, IEmailSender emailSender)
    {
        _userRepository = userRepository;
        _emailSender = emailSender;
    }

    public override void Execute(SimpleSendEmailJobArgs args)
    {
        var senderUser = _userRepository.Get(args.SenderUserId);
        var targetUser = _userRepository.Get(args.TargetUserId);

        _emailSender.Send(senderUser.EmailAddress, targetUser.EmailAddress, args.Subject, args.Body);
    }
}
```

我们注入了用户仓储(取得用户邮箱信息)和邮件发送服务，简单实现邮件的发送。在这里 **SimpleSendEmailJobArgs** 是作业参数，定义如下：

```
[Serializable]
public class SimpleSendEmailJobArgs
{
    public long SenderUserId { get; set; }

    public long TargetUserId { get; set; }

    public string Subject { get; set; }

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

作业参数应该是可序列化的，因为该参数需要被序列化并存储到数据库中。在ABP中作业管理默认使用的是 **JSON** 序列化的方式(所以不需要添加[Serializable]特性)。当然最好是使用 **[Serializable]** 特性，因为我们可以自由的切换到其他作业管理中，以后可能会使用**Binary**序列化等等。

参数应该做到简洁，不应该包含实体或者其他非序列化的对象。正如 **SimpleSendEmailJob** 所展示的，我们应该仅存储实体的 **Id**并从仓储内得到该实体的作业。

## 2. 添加作业到队列中

在定义后台作业后，我们可以注入并使用 **IBackgroundJobManager** 接口来添加作业到队列中。

上面我们已经定义了TestJob类，让我们来看下如何使它排队：

```
public class MyService
{
```

```
private readonly IBackgroundJobManager _backgroundJobManager;

public MyService(IBackgroundJobManager backgroundJobManager)
{
    _backgroundJobManager = backgroundJobManager;
}

public void Test()
{
    _backgroundJobManager.Enqueue<TestJob, int>(42);
}
}
```

我们以42作为参数来排队。IBackgroundJobManager将会被实例化，并使用42这个参数来执行TestJob这个作业。

让我们为SimpleSendEmailJob添加一个新作业：

```
[AbpAuthorize]
public class MyEmailAppService : ApplicationService, IMyEmailAppService
{
    private readonly IBackgroundJobManager _backgroundJobManager;

    public MyEmailAppService(IBackgroundJobManager backgroundJobManager)
    {
        _backgroundJobManager = backgroundJobManager;
    }

    public async Task SendEmail(SendEmailInput input)
    {
        await _backgroundJobManager.EnqueueAsync<SimpleSendEmailJob, SimpleSendEmailJobArgs>(
            new SimpleSendEmailJobArgs
            {
                Subject = input.Subject,
                Body = input.Body,
                SenderUserId = AbpSession.GetUserId(),
                TargetUserId = input.TargetUserId
            });
    }
}
```

Enqueue (or EnqueueAsync) 方法还有其它参数如： **priority** 和 **delay**。

### 3. 定义后台工作管理者

**BackgroundJobManager** 默认实现了IBackgroundJobManager接口。它能够被其它的后台作业提供者给替换掉(如：集成Hangfire)。在 **BackgroundJobManager** 中默认定义了一些信息：

- 在单线程中，它是一种简单的 **FIFO**(先进先出) 的作业队列。使用 **IBackgroundJobStore** 来持久化作业。
- 对作业重试执行，直到作业执行成功(不抛出任何异常)或者操作超时。默认作业超时设置是2天。
- 当作业执行成功后，作业将会从数据库中删除。如果执行超时，那么该作业会被设置为 **abandoned** 并留在数据库中。
- 作业重试执行的时间间隔会慢慢增长。第一次重试是等待1分钟，第二次是等待2分钟，第三次是等待4分钟等等。
- 以固定的时间间隔轮询存储中的作业。查询作业是通过优先级和重试次数来排序的。

#### 后台作业存储

BackgroundJobManager 默认是需要数据存储来保存和检索作业的。如果你没有实现 **IBackgroundJobStore**，那么它会使用 **InMemoryBackgroundJobStore** 来存储作业，当然作业不会被持久化到数据库中。你可以简单的实现它来存储作业到数据库中或者你可以在 **module-zero** 直接使用，因为它已被实现。

### 4. 配置

你可以在你的模块方法： **PreInitialize** 中，使用 **Configuration.BackgroundJobs** 配置后台作业系统。

#### 禁用后台作业

你可能想在你的应用中禁用后台作业：

```
public class MyProjectWebModule : AbpModule
{
    public override void PreInitialize()
    {
        Configuration.BackgroundJobs.IsJobExecutionEnabled = false;
    }

    //...
}
```

这种需求及其少见。但是想象一下，同时开启同一个应用程序的多个实例且使用的是同一个数据库。在这种情况下，每个应用程序的作业将查询相同的数据库并执行它们。这会导致相同的任务的多次执行，并还会导致其它的一些问题。为了阻止这个，你有两个选择：

- 你可以仅为你应用程序的一个实例开启后台作业
- 你可以禁用Web应用的所有实例的后台作业，并且创建一个独立的应用程序(如：Windows服务)来执行你的后台作业。

## 5. 集成Hangfire

后台作业管理者可以被其他后台作业管理者替换。详情请参考[集成Hangfire](#)

### 7.1.3 后台工人

后台工人是不同于后台作业的。在应用程序中，它们是运行在后台单个线程中。一般来说，它们周期性的执行一些任务，例如：

- 后台工人能够周期性地执行旧日志的删除
- 后台工人可以周期性地确定非活跃性用户并且发送邮件给这些用户，使这些用户返回到你的网站中。

#### 1. 创建后台工人

创建后台工人，我们应该实现 **IBackgroundWorker** 接口。根据你的需要，你可以继承 **BackgroundWorkerBase** 或者 **PeriodicBackgroundWorkerBase**。

假设有一个非活跃性的用户，该用户最近30天都没有访问我们的应用程序。

```
public class MakeInactiveUsersPassiveWorker : PeriodicBackgroundWorkerBase, ISingletonDependency
{
    private readonly IRepository<User, long> _userRepository;

    public MakeInactiveUsersPassiveWorker(AbpTimer timer, IRepository<User, long> userRepository)
        : base(timer)
    {
        _userRepository = userRepository;
        Timer.Period = 5000; //5 seconds (good for tests, but normally will be more)
    }

    [UnitOfWork]
    protected override void DoWork()
    {
        using (CurrentUnitOfWork.DisableFilter(AbpDataFilters.MayHaveTenant))
        {
            var oneMonthAgo = Clock.Now.Subtract(TimeSpan.FromDays(30));

            var inactiveUsers = _userRepository.GetAllList(u =>
                u.IsActive &&
                ((u.LastLoginTime < oneMonthAgo && u.LastLoginTime != null) || (u.CreationTime < oneMonthAgo && u.Las
                ));

            foreach (var inactiveUser in inactiveUsers)
            {
                inactiveUser.IsActive = false;
                Logger.Info(inactiveUser + " made passive since he/she did not login in last 30 days.");
            }

            CurrentUnitOfWork.SaveChanges();
        }
    }
}
```

这是一个真实性的代码，你可以在[module-zero](#)找到。

- 如果派生自 **PeriodicBackgroundWorkerBase**，你需要实现 **DoWork** 方法，在该方法中实现你周期性执行的逻辑。
- 如果派生自 **BackgroundWorkerBase** 或者直接实现 **IBackgroundWorker** 接口，你将要重写/实现 **Start**，**Stop**，**WaitStop** 方法。Start 和 Stop 是非阻塞方法，WaitStop 应该等待后台工人完成当前的重要任务。

## 2. 注册后台工人

在创建后台工人后，我们应该添加它到 **IBackgroundWorkerManager**。通常是在我们的模块的 **PostInitialize** 方法中配置：

```
public class MyProjectWebModule : AbpModule
{
    //...

    public override void PostInitialize()
    {
        var workManager = IocManager.Resolve<IBackgroundWorkerManager>();
        workManager.Add(IocManager.Resolve<MakeInactiveUsersPassiveWorker>());
    }
}
```

一般情况我们都是 **PostInitialize** 做的，这没有严格的要求。你能够在任何地方注入 **IBackgroundWorkerManager** 并添加后台工人到运行时中。当你的应用程序停止后，**IBackgroundWorkerManager** 将停止和释放所有被注册过的后台工人。

## 3. 后台工人生命周期

后台工人通常是作为单例来执行的。这里也没有严格的限制，如果你需要同一个后台工人在多个实例中，你可以使它做为瞬时对象并添加多个实例到 **IBackgroundWorkerManager**，在这种情况下，后台工人将被参数化(你有一个 **LogCleaner** 类，但是有 2 个后台 **LogCleaner** 工人实例，它们会看到和清除不同的日志文件夹)。

### 7.1.4 使你的应用程序一直运行

后台作业和后台工人仅在你的应用程序运行的时候才工作。**Asp.Net** 应用默认是关闭的，如果 web 应用长时间没有被请求执行。所以，如果你的后台作业是宿主在 **Web** 应用中执行的，你应该确保你的 web 应用是被配置为一直执行的。否则，后台作业仅在你的 web 应用在使用的时候才会执行。

有一些技术可以实现这个，最简单的方法是用一个外部程序向你的 web 应用周期性的发送请求。因此，你也可以检测你的 web 应用是否启动并运行。[Hangfire](#) 文档中描述了另外的实现方法。## 7.2 ABP 后台服务 - 集成 Hangfire

### 7.2.1 简介

**Hangfire** 是一个综合性的后台作业管理工具。你可以用 **Hangfire** 来替换 **ABP** 中默认实现的后台作业管理者。你可以对 **Hangfire** 使用相同的后台作业 API。因此，你的代码将独立于 **Hangfire**。但是，如果你喜欢，你也可以直接的使用 **Hangfire** 的 API。

### 7.2.2 安装

首先，应该使用 **Nuget** 安装 **Abp.HangFire** 到你的项目。然后你可以为 **Hangfire** 安装任何存储包。最常用的一个是 **SQL Server** 存储(请看 [Hangfire.SqlServer](#))。在你安装这些 **Nuget** 包后，你可以配置 **ABP** 来使用 **Hangfire**。如下所示：

```
[DependsOn(typeof(AbpHangfireModule))]  
public class MyProjectWebModule : AbpModule  
{  
    public override void PreInitialize()  
    {  
        Configuration.BackgroundJobs.UseHangfire(configuration =>  
        {  
            configuration.GlobalConfiguration.UseSqlServerStorage("Default");  
        });  
    }  
    //...  
}
```

上面我们添加了被依赖项 **AbpHangfireModule** 并且用 **Configuration.BackgroundJobs.UseHangfire** 方法来开启和配置了 **Hangfire** ("Default" 是配置在 **web.config** 中的连接字符串)。

注意：Hangfire 需要在数据库中有创建 Schema 的权限，因为在首次运行后，它会创建自己的架构和表。

详细请查阅[HangFire文档](#)。## 8.1 ABP实时服务 - 通知系统

### 8.1.1 简介

在系统中，通知是用来告知用户特定事件的。ABP提供了一个基于实时通知的基础设施 **pub/sub**。

### 8.1.2 发送模式

有两种方法来发送通知给用户：

- 用户 订阅 一个特定的通知类型。当我们发布这个类型的通知时，该通知会被投递给所有的订阅用户。这就是 **pub/sub** 模式。
- 我们能直接的发送通知给目标用户。

### 8.1.3 通知类型

通知类型也有两种：

- 常规通知 是任意类型的通知。例如："如果某个用户发送给我一个交友请求，那么通知我。"
- 实体通知 是被关联到一个指定的实体。"如果某个用户评论了这个照片，那么通知我。"，这是一个基于实体的通知，因为它被关联到一个特定的照片实体。用户可以得到对某些照片评论的通知，而不是所有照片的。

### 8.1.4 通知数据

通常来说通知中包含了通知数据。例如："如果某个用户发送给我一个友好的请求，那么通知我。"，该通知可以有两个数据属性：发送者的用户名(那个用户发送了这个交友请求)和请求信息(用户发送请求的具体信息)。很显然，这个通知的数据类型是和通知类型紧密耦合的。不同的通知类型有不同的数据类型。

通知数据是可选的。有些通知不需要数据。ABP预定义了一些通知数据类型，这些类型适合于大多数情况下。**MessageNotificationData** 能够用于一些简单的消息，而 **LocalizableMessageNotificationData** 可以用于本地化和参数化的通知消息。在后面的部分我们会看到一些有用的例子。

### 8.1.5 通知的程度

ABP有5种程度的通知等级，它被定义在枚举类型 **NotificationSeverity** 中: **Info**, **Success**, **Warn**, **Error** 以及 **Fatal**。默认是 **Info**。很显然，这个通知的数据类型是和通知类型紧密耦合的。不同的通知类型有不同的数据类型。

关于通知持久化

了解更多请看 **Notification Store**

### 8.1.6 订阅通知

**INotificationSubscriptionManager** 接口中提供了订阅通知的API，例如：

```
public class MyService : ITransientDependency
{
    private readonly INotificationSubscriptionManager _notificationSubscriptionManager;

    public MyService(INotificationSubscriptionManager notificationSubscriptionManager)
    {
        _notificationSubscriptionManager = notificationSubscriptionManager;
    }

    //订阅常规通知
    public async Task Subscribe_SentFrendshipRequest(int? tenantId, long userId)
    {
        await _notificationSubscriptionManager.SubscribeAsync(tenantId, userId, "SentFrendshipRequest");
    }

    //订阅实体通知
    public async Task Subscribe_CommentPhoto(int? tenantId, long userId, Guid photoId)
    {
        await _notificationSubscriptionManager.SubscribeAsync(tenantId, userId, "CommentPhoto", new EntityIdentifier(
    }
}
```

首先，我们注入 `INotificationSubscriptionManager` 接口。第一个方法是定义 常规通知，用户想得到某些用户发来的交友请求的通知。第二个方法是订阅与 特定实体(Photo) 有关的通知。用户想得到某些用户对特定的照片写了评论的通知。

每个通知的类型应该有唯一的名字(就像例子中的 `SentFrendshipRequest` 和 `CommentPhoto` 一样)。

`INotificationSubscriptionManager` 还有其它的方法来管理订阅，如：`UnsubscribeAsync`, `IsSubscribedAsync`, `GetSubscriptionsAsync`等等。

### 8.1.7 发布通知

`INotificationPublisher`被用来发布通知，例如：

```
public class MyService : ITransientDependency
{
    private readonly INotificationPublisher _notiticationPublisher;

    public MyService(INotificationPublisher notiticationPublisher)
    {
        _notiticationPublisher = notiticationPublisher;
    }

    //发送常规通知给特定用户
    public async Task Publish_SentFrendshipRequest(string senderUserName, string friendshipMessage, long targetUserId)
    {
        await _notiticationPublisher.PublishAsync("SentFrendshipRequest", new SentFrendshipRequestNotificationData(se

    }

    //发送实体通知给特定用户
    public async Task Publish_CommentPhoto(string commenterUserName, string comment, Guid photoId, long photoOwnerId)
    {
        await _notiticationPublisher.PublishAsync("CommentPhoto", new CommentPhotoNotificationData(commenterUserName,

    }

    //发送常规通知给所有的订阅用户并制定通知的严重等级
    public async Task Publish_LowDisk(int remainingDiskInMb)
    {
        //例如: "LowDiskWarningMessage" 英文内容是 -> "Attention! Only {remainingDiskInMb} MBs left on the disk!"
        //注意, 磁盘空间仅剩下{remainingDiskInMb} MBs
        var data = new LocalizableMessageNotificationData(new LocalizableString("LowDiskWarningMessage", "MyLocalizat
        data["remainingDiskInMb"] = remainingDiskInMb;

        await _notiticationPublisher.PublishAsync("System.LowDisk", data, severity: NotificationSeverity.Warn);
    }
}
```

在第一个例子中，我们对一个用户发布了一条通知。`SentFrendshipRequestNotificationData` 应该派生自 `NotificationData` 如下所示：

```
[Serializable]
public class SentFrendshipRequestNotificationData : NotificationData
{
    public string SenderUserName { get; set; }

    public string FriendshipMessage { get; set; }

    public SentFrendshipRequestNotificationData(string senderUserName, string friendshipMessage)
    {
        SenderUserName = senderUserName;
        FriendshipMessage = friendshipMessage;
    }
}
```

在第二个例子中，我们发送了一个特定实体的通知给了特定的用户。通知数据类不需要被 序列化 (因为默认被序列化为 JSON)。但是还是建议你特性：`Serializable` 加在数据类上，因为你可能需要在应用之间移动通知，也可能在将来使用二进制序列化。此外，正如之前声明的那样，通知数据是可选的，而且对于所有的通知可能不是必须的。

注意：如果我们发布通知给特定的用户，那么他们不需要订阅那些通知。

在第三个例子中，我们没有定义一个专门的通知数据类。相反，直接使用了内置的基于字典类型的

`LocalizableMessageNotificationData` 类，并以 `Warn` 等级来发布通知。`LocalizableMessageNotificationData` 能存储基于字

典的任意数据(这是由于自定义通知数据类型是继承 **NotificationData** 类)。我们在本地化时使用 **remainingDiskInMb** 作为参数。本地化消息可以包含这些参数(就像例子中的"Attention! Only {remainingDiskInMb} MBs left on the disk!").我们将会客户端看到如何本地化。

### 8.1.8 用户通知管理

**IUserNotificationManager** 被用来管理用户通知。有这些方法来为用户 **get**, **update** 或 **delete** 通知。你可以使用它们为你的应用程序来准备一个通知列表页面。

### 8.1.9 实时通知

当你使用 **IUserNotificationManager** 来查询通知时, 通常我们是想实时推送通知到客户端。

通知系统使用 **IRealTimeNotifier** 接口发送实时通知给用户。任何类型的实时通信系统都能实现它。它已经被实现在一个独立的 **SignalR** 包中。在启动模板中已经安装了 **SignalR**。详情请参考集成 **SignalR**。

注意: 在后台作业中, 通信系统异步调用 **IRealTimeNotifier**。所以, 通知可能会延迟一小会才会被发送。

### 8.1.10 客户端

当实时通知被接受的时候, **ABP**在客户端触发了一个 全局事件。你可以像下面一样注册它来获取通知:

```
abp.event.on('abp.notifications.received', function (userNotification) {
    console.log(userNotification);
});
```

每次接受到实时通知 **abp.notifications.received** 事件会被触发。你能够想上面展示的那样注册这个事件来获取通知。详情请参考事件总线。下面是一个传入的JSON格式的"System.LowDisk"示例:

```
{
  "userId": 2,
  "state": 0,
  "notification": {
    "notificationName": "System.LowDisk",
    "data": {
      "message": {
        "sourceName": "MyLocalizationSourceName",
        "name": "LowDiskWarningMessage"
      },
      "type": "Abp.Notifications.LocalizableMessageNotificationData",
      "properties": {
        "remainingDiskInMb": "42"
      }
    },
    "entityType": null,
    "entityTypeName": null,
    "entityId": null,
    "severity": 0,
    "creationTime": "2016-02-09T17:03:32.13",
    "id": "0263d581-3d8a-476b-8e16-4f6a6f10a632"
  },
  "id": "4a546baf-bf17-4924-b993-32e420a8d468"
}
```

在这个对象中:

- **userId**: 当前userId。通常你不需要知道这个, 因为你那个是当前用户。
- **state**: 枚举类型 **UserNotificationState** 的值。0: **Unread**, 1: **Read**。
- **notification**: 通知详细信息:
  - i. **notificationName**: 通知的唯一名字(发布通知时使用相同的值)。
  - ii. **data**: 通知数据。在上面例子中, 我们使用了 **LocalizableMessageNotificationData** (在之前的例子中我们使用它来发布的)。
    - **message**: 本地化信息。在UI端, 我们可以使用 **sourceName** 和 **name** 来本地化信息。
    - **type**: 通知数据类型。类型的全名称, 包含名称空间。当处理这个通知数据的时候, 我们可以检查这个类型。
    - **properties**: 自定义属的基于字典类型的属性。
  - iii. **entityType**, **entityTypeName** 和 **entityId**: 实体信息, 如果这是一个与实体相关的通知。



iv. **severity**: 枚举类型 **NotificationSeverity** 的值。0: Info, 1: Success, 2: Warn, 3: Error, 4: Fatal。

v. **creationTime**: 表示通知被创建的时间。

vi. **id**: 通知的id。

- **id**: 用户通知id。

当然你不会去记录这个通知。你可以使用通知数据来显示通知信息给用户。例如：

```
abp.event.on('abp.notifications.received', function (userNotification) {
    if (userNotification.notification.data.type === 'Abp.Notifications.LocalizableMessageNotificationData') {
        var localizedText = abp.localization.localize(
            userNotification.notification.data.message.name,
            userNotification.notification.data.message.sourceName
        );

        $.each(userNotification.notification.data.properties, function (key, value) {
            localizedText = localizedText.replace('{ ' + key + '}', value);
        });

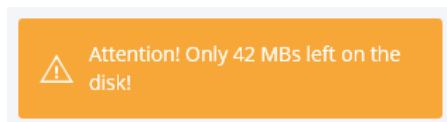
        alert('New localized notification: ' + localizedText);
    } else if (userNotification.notification.data.type === 'Abp.Notifications.MessageNotificationData') {
        alert('New simple notification: ' + userNotification.notification.data.message);
    }
});
```

为了能够处理通知数据，我们应该检查数据类型。这个简单的例子展示的是从通知数据中取得消息。对于本地化消息 (LocalizableMessageNotificationData)，我们要本地化该消息并替换掉参数。对于简单消息(MessageNotificationData)，我们直接的取得该消息。当然，在一个真实的项目中，我们不会使用alert函数。我们会使用 **abp.notify** API来展示漂亮的UI通知。

如果你需要实现上面所展示的逻辑，这里有一个更简单且可伸缩性的方法。当推送通知被接收到的时候，你可以仅使用一行代码来显示 **UI**通知。

```
abp.event.on('abp.notifications.received', function (userNotification) {
    abp.notifications.showUiNotifyForUserNotification(userNotification);
});
```

下面展示的是一个 **UI** 通知 (上面System.LowDisk示例)。



它对于内置的通知数据类型(LocalizableMessageNotificationData 和 MessageNotificationData)可以很好的工作。如果你有自定义的通知数据类型，那么你应该像下面一样使用格式化器来注册这个数据：

```
abp.notifications.messageFormatters['MyProject.MyNotificationDataType'] = function(userNotification) {
    return ...; //在这是实现格式化数据的逻辑并返回消息
};
```

因此，**showUiNotifyForUserNotification** 能对你的数据类型来创建并显示该消息。如果你仅需要格式化消息，你可以直接的使用 **abp.notifications.getFormattedMessageFromUserNotification(userNotification)**，这是由 **showUiNotifyForUserNotification**内部使用的。

当你接受到推送通知的时候，启动模板已经包含了显示UI通知的代码。

### 8.1.11 存储通知

通知系统使用 **INotificationStore** 来持久化通知。为了使通知系统正确的工资，我们应该实现这个接口。你可以自己实现它，或者使用已经实现了该接口的**module-zero**。

### 8.1.12 通知定义

使用之前，你不需要定义一个通知。你可以使用任何的没有被定义过的通知名字。但是，定义通知名字可以给你带来额外的好处。例如：你可能需要在你的应用程序中调研所有的通知。在这种情况下，我们需要为我们模块 通知提供者(notification provider)，如下所示：

```
public class MyAppNotificationProvider : NotificationProvider
{
    public override void SetNotifications(INotificationDefinitionContext context)
    {
        context.Manager.Add(
            new NotificationDefinition(
                "App.NewUserRegistered",
                displayName: new LocalizableString("NewUserRegisteredNotificationDefinition", "MyLocalizationSourceName"),
                permissionDependency: new SimplePermissionDependency("App.Pages.UserManagement")
            )
        );
    }
}
```

**App.NewUserRegistered** 是通知的唯一名字。我们定义了一个本地化的 **displayName**(当我们订阅了该通知时, 我们可以在UI上显示它)。最后, 我们声明了 **App.Pages.UserManagement** 权限, 只有当用户具有该权限的时候, 该通知才会显示给用户。

当然这里还有其它一些参数。你可以在代码中研究它们。对于通知定义只有通知的名字是必须的。

在你定义了如上所述的通知提供器后, 我们应该在模块的 **PreInitialize** 方法中注册它。如下所示:

```
public class AbpZeroTemplateCoreModule : AbpModule
{
    public override void PreInitialize()
    {
        Configuration.Notifications.Providers.Add<MyAppNotificationProvider>();
    }

    //...
}
```

最后, 你可以在你的应用程序中注入并使用 **INotificationDefinitionManager** 接口来获取通知定义。

## 8.2 ABP实时服务 - 集成SignalR

### 8.2.1 简介

在基于ABP创建的项目中, 有一个很容易的方式使用 **SignalR**, 那就是使用 **Abp.Web.SignalR**。详情请参考SignalR文档。

### 8.2.2 安装

#### 1. 服务器端

使用Nuget安装**Abp.Web.SignalR**到你的项目中(通常是你的Web项目)并且在模块中添加被依赖的模块:

```
[DependsOn(typeof(AbpWebSignalRModule))]
public class YourProjectWebModule : AbpModule
{
    //...
}
```

然后, 在你的OWIN Startup类中使用 **MapSignalR** 方法, 正如你往常那样做的:

```
[assembly: OwinStartup(typeof(Startup))]
namespace MyProject.Web
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.MapSignalR();

            //...
        }
    }
}
```

注意: **Abp.Web.SignalR** 依赖于 **Microsoft.AspNet.SignalR.Core**。所以, 你需要安装 **Microsoft.AspNet.SignalR**到你的 Web项目中。详情请参考[SignalR文档](#)。

## 2. 客户端

脚本 **abp.signalr.js** 应该被引用到页面中。它位于 **Abp.Web.Resources** 包中(它已经被安装到启动模板中)。我们应该在 **signalr hubs** 后引用它:

```
<script src="../../signalr/hubs"></script>
<script src="../../Abp/Framework/scripts/libs/abp.signalr.js"></script>
```

这么做了以后, **SignalR**就已经恰当的配置和集成到你的项目中了。

## 3. 建立连接

当 **abp.signalr.js** 被引用到页面后, **ABP**会自动的连接到你的服务器。一般我们都会这么做, 但是在某些情况下你不想这样做。你可以像下面代码所示禁用自动连接:

```
<script>
    abp.signalr = abp.signalr || {};
    abp.signalr.autoConnect = false;
</script>
```

在这种情况下, 你可以手动调用 **abp.signalr.connect()** 函数来连接服务器。

当客户端连接到服务器时, 全局事件 "**abp.signalr.connected**" 会被触发。当连接建立成功的时候, 你可以注册这个事件来采取相应的行动。详情请参考[Javascript函数库](#)

### 8.2.3 内置功能

你可以在应用程序中使用所有的**SignalR**的功能。还有, 在 **Abp.Web.SignalR** 中实现了一些内置功能。

#### 1. 通知

**Abp.Web.SignalR** 实现了 **IRealTimeNotifier** 接口来发送实时时间到客户端。因此, 你的用户可以获得实时的推送通知。

#### 2. 在线客户端

**ABP**提供了 **IOneClientManager** 来取得在线用户的信息(如: 注入**IOneClientManager**以及使用**GetByUserIdOrNull**, **GetAllClients**, **IsOnline** 方法)。为了能够正常工作, **IOneClientManager**需要一个通信基础设施。**Abp.Web.SignalR** 提供了这个基础设施。如果安装了**SignalR**, 那么在应用的任何层都可以注入并使用**IOneClientManager**。

#### 3. PascalCase vs. camelCase

**Abp.Web.SignalR** 使用 **CamelCasePropertyNamesContractResolver** 重写了 **SignalR's** 默认的序列化类 **ContractResolver**。因此, 在服务器端类具有 **PascalCase** 属性, 而在客户端为了发送/接受对象, 我们使用 **camelCase** (因为**camelCase**在**JavaScript**中更受欢迎)。如果你想在某些程序集中忽略这个, 那么你可以将那些程序集添加**AbpSignalRContractResolver.IgnoredAssemblies** 列表中。

### 8.2.3 你的SignalR代码

使用 **Abp.Web.SignalR** 包也会简化你的 **SignalR**代码。假设我们想要添加一个**Hub**到你的应用程序中:

```
public class MyChatHub : Hub, ITransientDependency
{
    public IAbpSession AbpSession { get; set; }

    public ILogger Logger { get; set; }

    public MyChatHub()
    {
        AbpSession = NullAbpSession.Instance;
        Logger = NullLogger.Instance;
    }

    public void SendMessage(string message)
    {
        Clients.All.getMessage(string.Format("User {0}: {1}", AbpSession.UserId, message));
    }
}
```

```
}

public async override Task OnConnected()
{
    await base.OnConnected();
    Logger.Debug("A client connected to MyChatHub: " + Context.ConnectionId);
}

public async override Task OnDisconnected(bool stopCalled)
{
    await base.OnDisconnected(stopCalled);
    Logger.Debug("A client disconnected from MyChatHub: " + Context.ConnectionId);
}
}
```

为了使我们的Hub可以简单的注册到依赖注入系统中，我们可以实现 **ITransientDependency** 接口。当然你可以根据你的需求，注册它为单例模式。我们也使用属性注入了Session和Logger。

SendMessage是hub的一个方法，它可以被客户端使用。在这个方法中，我们可以调用所有客户端的 getMessage函数。正如你看到的那样，我们可以使用AbpSession来获得当前的用户id(如果用户已经登录)。为了演示，我们也重写了 OnConnected 和 OnDisconnected，实际这里是不需要的。

下面是用在Hub中，用来发送/接受信息的客户端脚本：

```
var chatHub = $.connection.myChatHub; //get a reference to the hub

chatHub.client.getMessage = function (message) { //register for incoming messages
    console.log('received message: ' + message);
};

abp.event.on('abp.signalr.connected', function() { //register for connect event
    chatHub.server.sendMessage("Hi everybody, I'm connected to the chat!"); //send a message to the server
});
```

然后，在我们需要发送信息到服务器时，我们就可以使用 chatHub。详情请参考SignalR文档。

## 9.1 ABP基础设施层 - 集成Entity Framework

ABP可以与任何ORM框架协同工作，它内置了对EntityFramework的集成支持。本文将介绍如何在ABP中使用EntityFramework。本文假定你已经初步掌握了EntityFramework。

译者注：怎么才算初步掌握了EntityFramework呢？译者认为应当懂得使用Code First模式进行CRUD。

### 9.1.1 Nuget包

在ABP中要使用EntityFramework作为ORM框架的话，需要到Nuget上下载一个名为Abp.EntityFramework的包。比较好的做法是：新建一个独立的程序集(dll)，然后在这个程序集中调用这个包和EntityFramework。

ABP官方提供的模板程序就是这样做的。模板程序的下载方法详见《ABP系列之2、ABP入门教程》

### 9.1.2 创建DbContext

要使用EntityFramework，首先需要定义一个DbContext类。下面是一个DbContext类的示例：

```
public class SimpleTaskSystemDbContext : AbpDbContext
{
    public virtual IDbSet<Person> People { get; set; }
    public virtual IDbSet<Task> Tasks { get; set; }

    public SimpleTaskSystemDbContext()
        : base("MyConnectionString")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<Person>().ToTable("StsPeople");
    }
}
```

```

        modelBuilder.Entity<Task>().ToTable("StsTasks").HasOptional(t => t.AssignedPerson);
    }
}

```

上面的SimpleTaskSystemDbContext本质上是一个DbContext类，它派生自AbpDbContext，而不是DbContext。AbpDbContext提供了很多重载的构造函数，如果需要的话，我们可以使用它。

EntityFramework可以使用约定的方式来映射实体和数据表。除非你想进行自定义映射，否则你甚至不需要做任何配置。在上例中，我们将实体映射到了不同的表中。默认情况下（按照约定优先于配置的原则，会默认采用约定的方式），Task实体会映射到Tasks表，但在这里我们将它映射到了StsTasks表。相比起使用Data Annotation模式来进行自定义映射，我更喜欢使用Fluent API模式。当然，你可以选择你所喜欢的模式，这里没有特别的限制。

### 9.1.3 仓储

ABP提供了一个名为EfRepositoryBase的基类，这使得实现仓储变得简单快捷。要实现IRepository接口，你只需要从这个基类进行派生即可。但是更好的做法是，自定义一个派生自EfRepositoryBase的基类，然后在这个基类中添加一些通用的方法。这样做的好处是，所有派生自这个基类的仓储都继承了这些通用方法。

#### 1. 应用程序专用的仓储基类 (Application specific base repository class)

在下面的例子中，我们定义了一个名为SimpleTaskSystem仓储基类，这个类是此应用程序所专用的。

```

// Base class for all repositories in my application
// 应用程序中的所有仓储的基类
public class SimpleTaskSystemRepositoryBase<TEntity, TPrimaryKey> : EfRepositoryBase<SimpleTaskSystemDbContext, TEntity>
    where TEntity : class, IEntity<TPrimaryKey>
{
    public SimpleTaskSystemRepositoryBase(IDbContextProvider<SimpleTaskSystemDbContext> dbContextProvider)
        : base(dbContextProvider)
    {
    }
}
// add common methods for all repositories
// 添加仓储基类的通用方法
}
// A shortcut for entities those have integer Id
// 为所有拥有整型Id的实体添加一个快捷方式。
public class SimpleTaskSystemRepositoryBase<TEntity> : SimpleTaskSystemRepositoryBase<TEntity, int>
    where TEntity : class, IEntity<int>
{
    public SimpleTaskSystemRepositoryBase(IDbContextProvider<SimpleTaskSystemDbContext> dbContextProvider)
        : base(dbContextProvider)
    {
    }
}
// do not add any method here, add to the class above (because this class inherits it)
// 不要在这里添加任何通用方法，通用方法应当被添加到上面的基类中 (MyRepositoryBase<TEntity, TPrimaryKey> (SimpleTaskSystemRepositoryBase))。
}

```

需要注意的是，我们继承了基类EfRepositoryBase。这相当于做了一个声明，它会让ABP在仓储中调用SimpleTaskSystemDbContext。

#### 2. 仓储实现 (Implementing a repository)

如果你只想使用预定义的仓储方法的话，你甚至不需要为实体创建仓储类。如下所示：

```

public class PersonAppService : IPersonAppService
{
    private readonly IRepository<Person> _personRepository;
    public PersonAppService(IRepository<Person> personRepository)
    {
        _personRepository = personRepository;
    }
    public void CreatePerson(CreatePersonInput input)
    {
        person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };
        _personRepository.Insert(person);
    }
}

```

PersonAppService类采用构造函数注入的方式注入了一个IRepository对象，然后调用了预定义的Insert方法。采用这种方式，你可以方便地注入IRepository（或者IRepository）对象，然后使用预定义的方法。查看仓储文档以获得预定义方法的列表。

### 3. 自定义仓储(Custom repositories)

要实现自定义仓储，只要从上面所创建的应用程序专用的仓储基类（SimpleTaskSystemRepositoryBase）派生即可。

假定我们有一个名为Task的实体，它可以被赋予Person实体。Task有一个状态属性（值为新建、已分配、已完成等）。我们需要编写一个自定义方法，这个方法能根据某些条件获取Task列表，并且使得Task的AssignedPerson属性可以在一次数据库查询中被加载（使用EntityFramework的贪婪加载方法Include）。如下所示：

```
public interface ITaskRepository : IRepository<Task, long>
{
    List<Task> GetAllWithPeople(int? assignedPersonId, TaskState? state);
}

public class TaskRepository : SimpleTaskSystemRepositoryBase<Task, long>, ITaskRepository
{
    public TaskRepository(IDbContextProvider<SimpleTaskSystemDbContext> dbContextProvider)
        : base(dbContextProvider)
    {
    }

    public List<Task> GetAllWithPeople(int? assignedPersonId, TaskState? state)
    {
        var query = GetAll();

        if (assignedPersonId.HasValue)
        {
            query = query.Where(task => task.AssignedPerson.Id == assignedPersonId.Value);
        }

        if (state.HasValue)
        {
            query = query.Where(task => task.State == state);
        }

        return query
            .OrderByDescending(task => task.CreationTime)
            .Include(task => task.AssignedPerson)
            .ToList();
    }
}
```

我们首先定义了一个名为ITaskRepository的接口，紧接着定义了一个名为TaskRepository的类来实现它。预定义方法GetAll()返回了一个IQueryable对象，接着我们将参数放入到Where筛选器中，最后调用ToList()来获得一个Task列表。

此外，我们可以通过调用base.Context属性来获得一个DbContext对象，这样一来，你就可以直接使用Entity Framework的所有功能。

仓储类应当在构造函数中获取ISessionProvider对象。通过这种方式，在单元测试的时候，我们可以很容易地注入一个虚拟的DbContext Provider对象；而在运行的时候，ABP会根据配置注入相应的DbContext Provider对象。

## 9.2 ABP基础设施层 - 集成NHibernate

ABP可以与任何ORM框架协同工作，它内置了对NHibernate的集成支持。本文将介绍如何在ABP中使用NHibernate。本文假定你已经初步掌握了NHibernate。

译者注：怎么才算初步掌握了NHibernate呢？译者认为应当懂得使用NHibernate进行CRUD，懂得使用Fluent模式进行映射。

### 9.2.1 Nuget包

要在ABP中使用NHibernate作为ORM框架的话，需要到Nuget上下载一个名为Abp.NHibernate的包。比较好的做法是：新建一个独立的程序集(dll)，然后在这个程序集中调用这个包和NHibernate。

译者注：ABP官方提供的模板程序就是这样做的。模板程序的下载方法详见《ABP系列之2、ABP入门教程》。

### 9.2.2 配置

要使用Nhibernate，首先要对它进行配置，配置的方法是在模块的PreInitialize方法中编写配置代码，如下所示：

```
[DependsOn(typeof(AbpNHibernateModule))]  
public class SimpleTaskSystemDataModule : AbpModule  
{  
    public override void PreInitialize()  
    {  
        var connStr = ConfigurationManager.ConnectionStrings["Default"].ConnectionString;  
  
        Configuration.Modules.AbpNHibernate().FluentConfiguration  
            .Database(MsSqlConfiguration.MsSql2008.ConnectionString(connStr))  
            .Mappings(m => m.FluentMappings.AddFromAssembly(Assembly.GetExecutingAssembly()));  
    }  
  
    public override void Initialize()  
    {  
        IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());  
    }  
}
```

ABP之所以能与Nhibernate协同工作，是因为内置的AbpNHibernateModule模块提供了必要的适配器。

### 1. 实体映射（Entity mapping）

在下面的示例中，我们使用Fluent映射模式来对所有的实体类进行映射：

```
public class TaskMap : EntityMap<Task>  
{  
    public TaskMap()  
        : base("TeTasks")  
    {  
        References(x => x.AssignedUser).Column("AssignedUserId").LazyLoad();  
        Map(x => x.Title).Not.Nullable();  
        Map(x => x.Description).Nullable();  
        Map(x => x.Priority).CustomType<TaskPriority>().Not.Nullable();  
        Map(x => x.Privacy).CustomType<TaskPrivacy>().Not.Nullable();  
        Map(x => x.State).CustomType<TaskState>().Not.Nullable();  
    }  
}
```

EntityMap类是ABP中的一个内置类，它派生自ClassMap类，可以自动地对Id属性进行映射，并且在构造函数中获取表名。在上例中我们从EntityMap中派生，然后使用FluentNHibernate来映射其它属性。当然，你也可以直接从ClassMap中派生。在ABP中，你可以使用FluentNHibernate中定义的所有API。

在上例中我们使用了Fluent映射模式，你也可以使用其它映射模式，比如基于xml文件的映射模式

译者注：Nhibernate有四种映射模式：（1）基于XML的映射。（2）基于特性的映射。（3）Fluent映射。（4）基于约定的映射，也称为自动映射。

## 9.2.3 仓储实现

如果需求比较简单的话，你甚至不需要创建单独的仓储类，直接使用仓储的默认实现即可。如果实在需要创建单独的仓储类的话，建议从NhRepositoryBase中派生。

### 1. 默认实现（Default implementation）

如果你只想使用预定义的仓储方法的话，你甚至不需要为实体创建单独的仓储类。如下所示：

```
public class PersonAppService : IPersonAppService  
{  
    private readonly IRepository<Person> _personRepository;  
  
    public PersonAppService(IRepository<Person> personRepository)  
    {  
        _personRepository = personRepository;  
    }  
  
    public void CreatePerson(CreatePersonInput input)  
    {  
        person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };  
    }  
}
```



```

        _personRepository.Insert(person);
    }
}

```

PersonAppService类采用构造函数注入的方式注入了一个IRepository对象，然后调用了预定义的Insert方法。采用这种方式，你可以方便地注入IRepository（或者IRepository）对象，然后调用预定义的方法。请查看仓储文档以获得预定义方法的列表。

## 2. 自定义仓储（Custom repositories）

如果你想添加自定义方法，最佳的作法是，首先将它添加到仓储接口中，然后在仓储类中实现。ABP提供了一个名为NhRepositoryBase的基类，这使得实现仓储变得简单快捷，要实现IRepository接口，只需要从这个基类中派生。

假定我们有一个名为Task的实体，它可以被赋予Person实体。Task有一个State属性（值为新建、已分配、已完成等）。我们需要编写一个自定义方法，这个方法能根据某些条件获取Task列表，并且使得Task的AssignedPerson属性可以在一次数据库查询中被加载（使用NHibernate的立即加载方法Fetch）。如下所示：

```

public interface ITaskRepository : IRepository<Task, long>
{
    List<Task> GetAllWithPeople(int? assignedPersonId, TaskState? state);
}

public class TaskRepository : NhRepositoryBase<Task, long>, ITaskRepository
{
    public TaskRepository(ISessionProvider sessionProvider)
        : base(sessionProvider)
    {
    }
    public List<Task> GetAllWithPeople(int? assignedPersonId, TaskState? state)
    {
        var query = GetAll();
        if (assignedPersonId.HasValue)
        {
            query = query.Where(task => task.AssignedPerson.Id == assignedPersonId.Value);
        }
        if (state.HasValue)
        {
            query = query.Where(task => task.State == state);
        }
        return query
            .OrderByDescending(task => task.CreationTime)
            .Fetch(task => task.AssignedPerson)
            .ToList();
    }
}

```

预定义方法GetAll()返回了一个IQueryable对象，接着我们将参数放入到Where筛选器中，最后调用ToList()来获得一个Task列表。

此外，你可以通过调用base.Session属性来获得一个Session对象，这样一来，你就可以直接使用NHibernate的所有功能。

仓储类应当在构造函数中获取ISessionProvider对象。通过这种方式，在单元测试的时候，我们可以很容易地注入一个虚拟的Session Provider 对象；而在运行的时候，ABP会根据配置注入相应的Session Provider对象。

## 3. 应用程序专用的仓储基类（Application specific base repository class）

要实现仓储类，只需要从NhRepositoryBase中派生即可。但是更好的做法是，自定义一个派生自NhRepositoryBase的基类，然后在这个基类中添加一些通用的方法。这样做的好处是，所有派生自这个基类的仓储都继承了这些通用方法。

```

// 应用程序中的所有仓储的基类
public abstract class MyRepositoryBase<TEntity, TPrimaryKey> : NhRepositoryBase<TEntity, TPrimaryKey>
    where TEntity : class, IEntity<TPrimaryKey>
{
    protected MyRepositoryBase(ISessionProvider sessionProvider)
        : base(sessionProvider)
    {
    }
    // 添加仓储基类的通用方法
}
//为所有拥有整型 Id 的实体添加一个快捷方式
public abstract class MyRepositoryBase<TEntity> : MyRepositoryBase<TEntity, int>
    where TEntity : class, IEntity<int>

```

```

{
    protected MyRepositoryBase(ISessionProvider sessionProvider)
        : base(sessionProvider)
    {
    }
}
// 不要在这里添加任何通用方法，通用方法应当被添加到上面的基类中(MyRepositoryBase<TEntity, TPrimary>)
public class TaskRepository : MyRepositoryBase<Task>, ITaskRepository
{
    public TaskRepository(ISessionProvider sessionProvider)
        : base(sessionProvider)
    {
    }
}
// 添加本仓储类的专用方法
}

```

(6.1、6.2由深圳-Leo翻译) ## ABP 基础设施层 - 集成 Entity Framework Core

### 9.3.1 简介

**Abp.EntityFrameworkCore** nuget package 被用来集成到EF Core ORM框架. 在安装这个包以后，我们应该在模块类 **AbpEntityFrameworkCoreModule** 上添加 **DependsOn** 特性。

### 9.3.2 DbContext

EF Core要求有个派生自DbContext的类。在ABP中，我们应该使其派生自 **AbpDbContext**，如下所示：

```

public class MyDbContext : AbpDbContext
{
    public DbSet<Product> Products { get; set; }

    public MyDbContext(DbContextOptions<MyDbContext> options)
        : base(options)
    {
    }
}

```

如上所示，构造函数应该取得 **DbContextOptions<T>**。

### 9.3.3 Configuration

#### In Startup Class

对于ASP.NET Core项目和往常一样，你会有一个 **Startup** 文件和一个你能使用的EF Core的 **AddDbContext** 扩展方法，如下所示：

```

public IServiceCollection ConfigureServices(IServiceCollection services)
{
    ...

    services.AddDbContext<MyDbContext>(options =>
    {
        options.UseSqlServer(Configuration.GetConnectionString("Default"));
    });

    ...
}

```

这对ABP来说是一个简单且标准的方式。在这种方式中，连接字符串是静态的(因为我们一直使用"Default"连接字符串)。如果你只有一个数据库并有与之对应的DbContext，那就足够了。但是，如果你想要动态 切换连接字符串且更好的集成ABP框架的连接字符串，那么请使用 **AddAbpDbContext** 方法：

```

services.AddAbpDbContext<MyDbContext>(options =>
{
    options.DbContextOptions.UseSqlServer(options.ConnectionString);
});

```

我们使用给定的连接字符串并使用Sql Server作为数据库提供者。通常 `options.ConnectionString` 的值就是 `default` 连接字符串。但是ABP使用 `IConnectionStringResolver` 来确定。所以，这个行为方式是可以改变的并且连接字符串可以动态的切换。每当DbContext被实例化的时候，这个动作会传递给 `AddAbpDbContext` 方法。所以，你有机会可以返还不同条件下的连接字符串。

那么，哪里可以设置默认的连接字符串呢？

### In Module PreInitialize

你可以在模块的 `PreInitialize` 方法中来设置它，如下所示：

```
public class MyEfCoreAppModule : AbpModule
{
    public override void PreInitialize()
    {
        Configuration.DefaultNameOrConnectionString = GetConnectionString("Default");
        ...
    }
}
```

对于非web项目，我们不会有启动类，在这种情况下，我们可以使用 `Configuration.Modules.AbpEfCore().AddDbContext` 方法来配置DbContext，如下所示：

```
Configuration.Modules.AbpEfCore().AddDbContext<MyDbContext>(options =>
{
    options.DbContextOptions.UseSqlServer(options.ConnectionString);
});
```

## 9.3.4 Repositories

仓储被用来从更改层的抽象数据访问。详细请看[仓储文档](#)。

### Default Repositories

`Abp.EntityFrameworkCore` 为所有在DbContext中已定义的实体默认实现了仓储。你没必要创建仓储类来使用预定义的仓储方法。例如：

```
public class PersonAppService : IPersonAppService
{
    private readonly IRepository<Person> _personRepository;

    public PersonAppService(IRepository<Person> personRepository)
    {
        _personRepository = personRepository;
    }

    public void CreatePerson(CreatePersonInput input)
    {
        person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };
        _personRepository.Insert(person);
    }
}
```

`IRepository<Person>` 在 `PersonAppService` 的构造函数中被注入，并且使用了 `Insert` 方法。使用这种方式，你可以很容易的注入 `IRepository<TEntity>` 或者 `IRepository<TEntity, TPrimaryKey>` 并且使用其中预定义的方法。查看所有的预定义方法请查询[仓储文档](#)。

自定义仓储

如果你对标准的仓储方法不满意，你可以对你的实体创建自定义仓储。

特定于应用的基础仓储类

ABP提供了一个基础类：`EfCoreRepositoryBase` 使用它你可以很容易实现仓储类。为了实现 `IRepository` 接口，你可以使你的仓储类派生自该类。但是最好的方式是创建你自己的基类来扩展 `EfRepositoryBase` 类。因此，你可以轻松的添加共享/通用方法给你的仓储类。下面是SimpleTaskSystem应用对所有仓储的基类实现示例：

```
//应用程序中的所有仓储的基类
public class SimpleTaskSystemRepositoryBase : EfCoreRepositoryBase<SimpleTaskSystemDbContext, TEntity, TPrimaryKey>
{
    public SimpleTaskSystemRepositoryBase(IDbContextProvider<SimpleTaskSystemDbContext> dbContextProvider)
        : base(dbContextProvider)
    {
    }

    //为所有仓储添加通用方法
}

//对于那些有int类型Id的实体的仓储的快速实现方式
public class SimpleTaskSystemRepositoryBase<TEntity> : SimpleTaskSystemRepositoryBase<TEntity, int>
    where TEntity : class, IEntity<int>
{
    public SimpleTaskSystemRepositoryBase(IDbContextProvider<SimpleTaskSystemDbContext> dbContextProvider)
        : base(dbContextProvider)
    {
    }

    //别在这里添加任何方法，请将方法添加到上面那个类，因为该类被上面类继承
}
```

注意：我们的仓储类都是继承自 `EfCoreRepositoryBase<SimpleTaskSystemDbContext, TEntity, TPrimaryKey>`。这说明 ABP在仓储中使用了 `SimpleTaskSystemDbContext`。

#### 自定义仓储示例

为了实现一个自定义的仓储，你应该使你创建的应用的仓储派生自上面所说的特定于应用的基础仓储类。

假设我们有一个任务实体且任务能够被分配给某个人并且任务有自己的状态(如: new, assigned, completed 等等)。我们可能需要写一个自定义的方法来取得任务列表并且需要使用某些条件来执行数据过滤，如：分配人，任务状态 来过滤。如下所示：

```
public interface ITaskRepository : IRepository<Task, long>
{
    List<Task> GetAllWithPeople(int? assignedPersonId, TaskState? state);
}

public class TaskRepository : SimpleTaskSystemRepositoryBase<Task, long>, ITaskRepository
{
    public TaskRepository(IDbContextProvider<SimpleTaskSystemDbContext> dbContextProvider)
        : base(dbContextProvider)
    {
    }

    public List<Task> GetAllWithPeople(int? assignedPersonId, TaskState? state)
    {
        var query = GetAll();

        if (assignedPersonId.HasValue)
        {
            query = query.Where(task => task.AssignedPerson.Id == assignedPersonId.Value);
        }

        if (state.HasValue)
        {
            query = query.Where(task => task.State == state);
        }

        return query
            .OrderByDescending(task => task.CreationTime)
            .Include(task => task.AssignedPerson)
            .ToList();
    }
}
```

首先我们定义了 `ITaskRepository` 接口然后实现它。`GetAll()` 方法返回 `IQueryable<Task>`，然后我们可以使用给定的参数添加一些 `Where` 来过滤数据。最后我们可以调用 `ToList()` 方法来获取任务列表。

你也可以在仓储的方法中使用 **Context** 对象取得DbContext然后直接使用EF的API。

注意：在领域层定义自定义仓储接口，然后在 **EntityFrameworkCore** 项目中实现该接口。因此，你可以从任何项目中注入该接口并不需要引用EF Core。

#### Repository 最佳实践

- 在任何可能的地方使用默认仓储。即使你的实体有自定义仓储，你也能使用默认仓储(如果你使用了标准的仓储方法)。
- 如同上面所述：对自定义的仓储创建 仓储基类。
- 在领域层中定义自定义仓储接口(.Core 项目是在启动模板)，如果你从你的领域层抽象出EF Core, 那么请在 **.EntityFrameworkCore** 项目中实现自定义仓储类。

### 9.3.5 未实现的功能

Abp.EntityFrameworkCore 模块当前没有提供如下功能：

- **Data Filters**: Abp.EntityFramework 模块使用 EntityFramework.DynamicFilters 来实现自动数据过滤。当前它是不支持 EF Core。然而另一些库可以做到，如EF Plus，正如我们对它调研后所知：它可以用一种有限的方式来实现。例如：EF Plus 不支持导航属性的过滤。
- **Date Time normalization**: Abp.EntityFramework 模块对从数据库取得的实体的DateTime字段属性使其规范化。但是，对于 Abp.EntityFrameworkCore模块实现这个功能是不可能的。为了使得文档更美观、便于阅读，现统一MD文档排版规范，请大家自觉遵守。谢谢。
- 保持各级标题的数字序号，如：1.1.3 ABP适用的场景。
- 文件格式为md，如：1.1 ABP领域层—实体.md。
- 文件名格式：数字序号 章标题—节标题.md，如：1.1 ABP领域层—实体.md。
- 内容首行为节标题，即文件名，如：3.1 ABP 领域层—实体。格式为二级标题 **##**，如：**## 3.1 ABP 领域层—实体**。
- 标题格式：

如：

节标题 ---> 二级标题 **##** 如：**## 3.1 ABP 领域层—实体**

1级标题 ---> 三级标题 **###** 如：**### 3.1.1 实体类**

2级标题 ---> 四级标题 **####** 如：**#### 1. 审计（Auditing）**

若有更低级标题，请依次类推。

- 正文：

段落开始请顶格，无需留空。

正文部分除注意、编者注、译者注等特别说明外，不要使用特殊格式

注意、编者注、译者注等特殊说明，请使用引用格式 > 。

- 代码段请使用“`”格式，如：`

```
//这是一些代码
```

