

# ABP Framework 入门开发指南

---

## 领域驱动设计(DDD)红宝书

修订版本号：1.0.0.0

## 序 言

天道

该开发指南基于 ABP Framework 官方文档,由 **ABP 框架中国小组(ABPFrameWorkGroup)** 翻译。ABP 框架中国小组在 Github 的地址为: <https://github.com/ABPFrameWorkGroup>, 大家可以在该地址下载最新的翻译文档并加入讨论。

以后,最新的 ABPFrameWork 档,ABPFrameWork 实例教程,以及代码都通过 Github 的 **ABP 框架中国小组(ABPFrameWorkGroup)**发布,请大家关注。

ABP 官方文档翻译征集启动以来,得到了大家的热烈响应,先后共有 10 多名群友参与了翻译,目前翻译工作已经安排完毕,再次感谢广大群友的热烈支持。毫无疑问,该开发指南的发布是一次团体智慧和协作的结晶,是**ABP 框架中国小组**共同努力的成果。我们应当感谢他们的默默付出,感谢他们在这个酷热的夏天依旧能够挥洒着汗水辛苦的工作。

此次的翻译工作完全是个人自主行为,旨在帮助那些学习 .NET 架构和研究领域驱动设计(DDD)的开发者们。我们有理由相信,通过学习官方文档并结合 ABP 的源码,大家的 .NET 水平必定有所提高。

该文档不仅是 ABP 的开发指南,也是 .NET 架构设计的一个优秀参考范本,指南中提及的各种封装和技术实现,也可以很方便的集成到自己的项目中去。因此,这里推荐大家在研究源码的时候,遇到问题可以首先参考一下这本指南。

- 关于此次翻译的标准如下:

- (1) 保留了原文的大部分关键词,这样方便大家以后学习这些常见词汇。
- (2) 翻译的人称有第一人称和第二人称,由于时间关系,来不及统一。我个人倾向于第一人称,因此对部分群友的翻译做了调整。
- (3) 对部分翻译的语法做了调整,并且更加符合中国人的习惯。
- (4) 翻译总体上保留了原文的风格,大家可以对比一下原文,这不失为一个学习英语的最佳实践。
- (5) 对于常见词汇,翻译统一了标准:如, **derived** 翻译为派生, **be used to** 翻译为“调用”而不是“被用来”, **implement** 翻译为实现,等等。这里不在叙述。

- 关于 **ABP** 快速开发框架及实战:

(1) 接下来 **ABP** 群会推出实例演示视频以及在线培训等。

(2)此外会根据大家的使用情况推出集成权限、用户、角色的基本开发框架。

(3)针对企业的开发者,我们将会推出高级版开发框架。

欢迎大家关注群主(上海-阳铭)的博客: <http://www.cnblogs.com/mienreal>

**ABP 架构设计交流群: 134710707**

**ABP 框架中国小组(ABPFrameWorkGroup)成员如下** (排名不分先后, 随机排列):

- 山东-李伟 303283209
- 台湾-小張 2987853943
- 厦门-浩歌 385650059
- 深圳-Carl 280141563
- 冰封 121087772
- NoZero 3921342
- 南京-菜刀 252900664
- 深圳-Leo 254213048
- 成都-乐忧 68336713
- 上海-半冷 562758404
- 上海-阳铭 614573519
- 东莞-天道 1832339824

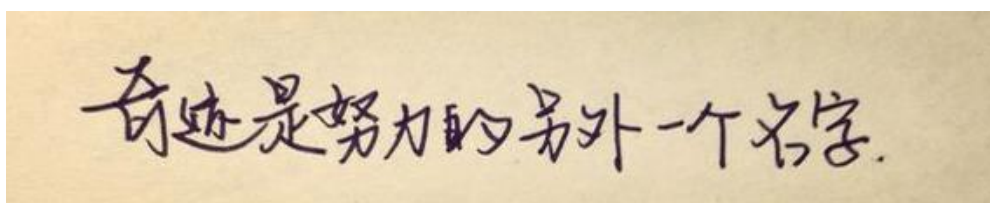
总排版: 北京-北北 34665992

总审核: 上海-阳铭 614573519

总编辑: 东莞-天道 1832339824

由于个人技术水平和英文水平也是有限的,因此错误在所难免。希望大家多多指正。有错误或其他疑问的, 请联系天道: [1832339824@qq.com](mailto:1832339824@qq.com)

东莞-天道 于 2015 年 6 月 13 日星期六晚 20 点



## 目 录

<b>1 ABP 总体介绍</b>	<b>1</b>
1.1 入门介绍	1
1.1.1 ABP 采用了以下技术	2
1.1.2 ABP 框架已实现了以下特性	2
1.1.3 ABP 适用的场景	4
1.2 多层架构体系	4
1.2.1 前言	4
1.2.2 ABP 的体系结构	5
1.2.3 领域层	5
1.2.4 应用层	6
1.2.5 基础设施层	6
1.2.6 WEB 与展现层	6
1.2.7 其它	7
1.3 模块系统	7
1.3.1 ABP 模块系统简介	7
1.3.2 生命期事件	8
1.3.3 模块依赖	9
1.3.4 自定义的模块方法	10
1.4 启动配置	11
1.4.1 配置 ABP	11

1.4.2 配置模块 .....	13
1.4.3 为一个模块创建配置 .....	13
<b>2 ABP 公共结构 .....</b>	<b>16</b>
2.1 ABP 依赖注入 .....	16
2.1.1 传统方式的问题 .....	16
2.1.2 解决方案 .....	18
2.1.3 依赖注入框架 .....	20
2.1.4 ABP 依赖注入的基础结构 .....	21
2.1.5 附件 .....	26
2.2 ABP 会话管理 .....	26
2.2.1 简介 .....	26
2.2.2 注入会话 .....	27
2.2.3 使用会话属性 .....	27
2.3 ABP 日志管理 .....	28
2.3.1 服务器端 .....	28
2.3.2 客户端 .....	32
2.4 ABP 设置管理 .....	32
2.4.1 介绍 .....	32
2.4.2 定义设置 .....	33
2.4.3 设置范围 .....	34
2.4.4 获取设置值 .....	35

2.4.5 更改设置 ..... 36

2.4.6 关于缓存 ..... 36

**3 ABP 领域层..... 37**

3.1 ABP 领域层—实体 ..... 37

3.1.1 实体类 ..... 37

3.1.2 接口约定 ..... 38

3.1.3 IEntity 接口 ..... 41

3.2 ABP 领域层—仓储 ..... 42

3.2.1 IRepository 接口 ..... 42

3.2.2 仓储的实现..... 47

3.2.3 管理数据库连接 ..... 48

3.2.4 仓储的生命周期 ..... 48

3.2.5 仓储的最佳实践 ..... 48

3.3 ABP 领域层—工作单元 ..... 49

3.3.1 通用连接和事务管理方法 ..... 49

3.3.2 ABP 的连接和事务管理..... 50

3.3.3 工作单元 ..... 53

3.3.4 选项 ..... 56

3.3.5 方法 ..... 57

3.3.6 事件 ..... 57

3.4 ABP 领域层—数据过滤器 ..... 58

3.4.1 介绍 .....58

3.4.2 预定义过滤器 .....58

3.4.3 禁用过滤器.....60

3.4.4 启用过滤器.....61

3.4.5 设定过滤器参数 .....62

3.4.6 自定义过滤器 .....62

3.4.7 其它对象关系映射工具.....64

3.5 ABP 领域层—领域事件 .....64

3.5.1 事件总线 .....64

3.5.2 定义事件 .....65

3.5.3 触发事件 .....65

3.5.4 事件处理 .....66

3.5.5 注册处理器.....68

3.5.6 取消注册事件 .....69

**4 ABP 应用层.....71**

4.1 ABP 应用层—应用服务 .....71

4.1.1 IApplicationService 接口.....71

4.1.2 应用服务类型 .....73

4.1.3 工作单元 .....74

4.1.4 应用服务的生命周期 .....76

4.2 ABP 应用层—数据传输对象 .....76

4.2.1 数据传输对象的作用 .....	76
4.2.2 DTO 约定 & 验证.....	78
4.2.3 DTO 和实体间的自动映射 .....	80
4.2.4 辅助接口和类型 .....	82
4.3 ABP 应用层—DTO 有效性验证 .....	82
4.3.1 使用数据注解 .....	83
4.3.2 自定义检验.....	84
4.3.3 设置缺省值.....	85
4.4 ABP 应用层—权限认证 .....	86
4.4.1 定义权限 .....	86
4.4.2 检查权限 .....	87
4.5 ABP 应用层—审计日志 .....	90
4.5.1 配置 .....	91
4.5.2 通过属性来启用和禁用审计日志 .....	92
4.5.3 说明 .....	93
<b>5 ABP 表现层.....</b>	<b>94</b>
5.1 ABP 展现层—动态 WebApi 层 .....	94
5.1.1 建立动态 web api 控制器 .....	94
5.1.2 使用动态 js 代理.....	95
5.2 ABP 展现层—本地化.....	97
5.2.1 程序语言 .....	97



5.2.2 本地化源文件 ..... 98

5.2.3 获得一个本地化配置文件 ..... 100

5.2.4 总结 ..... 103

5.3 ABP 展现层—Javascript 函数库 ..... 103

5.3.1 AJAX ..... 103

5.3.2 通知 ..... 107

5.3.3 消息 ..... 107

5.3.4 用户界面的繁忙提示 ..... 109

5.3.5 Js 日志接口 ..... 110

5.3.6 Javascript 公共方法 ..... 111

5.4 ABP 展现层—导航栏 ..... 111

5.4.1 创建菜单 ..... 112

5.4.2 显示菜单 ..... 114

5.5 ABP 展现层—异常处理 ..... 114

5.5.1 开启错误处理 ..... 115

5.5.2 非 Ajax 请求 ..... 115

5.5.3 AJAX 请求 ..... 117

5.5.4 异常事件 ..... 118

5.6 ABP 展现层—嵌入资源文件 ..... 118

**6 ABP 基础设施层 ..... 119**

6.1 ABP 基础设施层—集成 Entity Framework ..... 119

6.1.1 Nuget 包 ..... 119

6.1.2 创建 DbContext .....	119
6.1.3 仓储 .....	120
6.2 ABP 基础设施层—集成 NHibernate.....	124
6.2.1 Nuget 包 .....	125
6.2.2 配置 .....	125
6.2.3 仓储实现 .....	127
<b>7 ABP 实例一：ASP.NET Boilerplate .....</b>	<b>131</b>
7.1 引子 .....	131
7.2 什么是 ASP.Net Boilerplate? .....	132
7.3 ABP 不适用于那些场合? .....	133
7.4 开始 .....	134
7.5 使用模板创建空的网站应用程序.....	134
7.6 领域层.....	136
7.6.1 实体-Entities .....	137
7.6.2 仓储-Repository .....	138
7.6.3 关于命名空间 .....	139
7.7 基础设施层 .....	139
7.7.1 数据库迁移.....	139
7.7.2 实体映射 .....	143
7.7.3 仓储实现 .....	143
7.8 应用层.....	145

7.8.1 应用服务及数据传输对象 .....	145
7.8.2 DTO 验证 .....	149
7.8.3 动态 Web API 控制器 .....	150
7.9 表现层 .....	151
7.9.1 单页应用 .....	152
7.9.2 视图和视图模型 .....	152
7.9.3 本地化 .....	160
7.9.4 JavaScript API .....	162
7.10 更多 .....	164
7.10.1 模块系统 .....	164
7.10.2 依赖注入和约定 .....	165
7.11 结论 .....	165
<b>8 ABP 实例二：单页面网站应用程序 .....</b>	<b>167</b>
8.1 简介 .....	167
8.2 基于 Abp 创建应用程序 .....	168
8.3 创建实体 .....	169
8.4 创建 DbContext .....	170
8.5 创建 Database Migrations .....	171
8.6 定义仓储 .....	173
8.7 实现仓储 .....	174
8.8 构建应用程序服务 .....	176
8.9 验证 .....	179

8.10 异常处理 .....	181
8.11 构建 Web API 服务 .....	181
8.12 开发单页面应用 ( SPA ) .....	181
8.13 本地化.....	187

# 1 ABP 总体介绍

## 1.1 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。

ASP.NET Boilerplate

A starting point for new web applications using best practices and most popular tools.

<http://www.aspnetboilerplate.com>

Filters Find a repository...

**aspnetboilerplate** ← 这是ABP框架项目 C# ★ 378 🍴 175

ASP.NET Boilerplate

Updated a day ago

**module-zero** ← 这是一个用户身份及权限管理的模块 JavaScript ★ 64 🍴 45

ASP.NET Boilerplate - Module Zero

Updated 3 days ago

**aspnetboilerplate-samples** ← 这是示例项目 JavaScript ★ 47 🍴 48

Sample projects using ASP.NET Boilerplate

Updated 7 days ago

**People**

**hikalkan**  
Halil Ibrahim Kalkan

作者是一位土耳其的架构师

### 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）

我在其他项目中看到的很多优秀设计，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。

（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 多层架构体系

### 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：

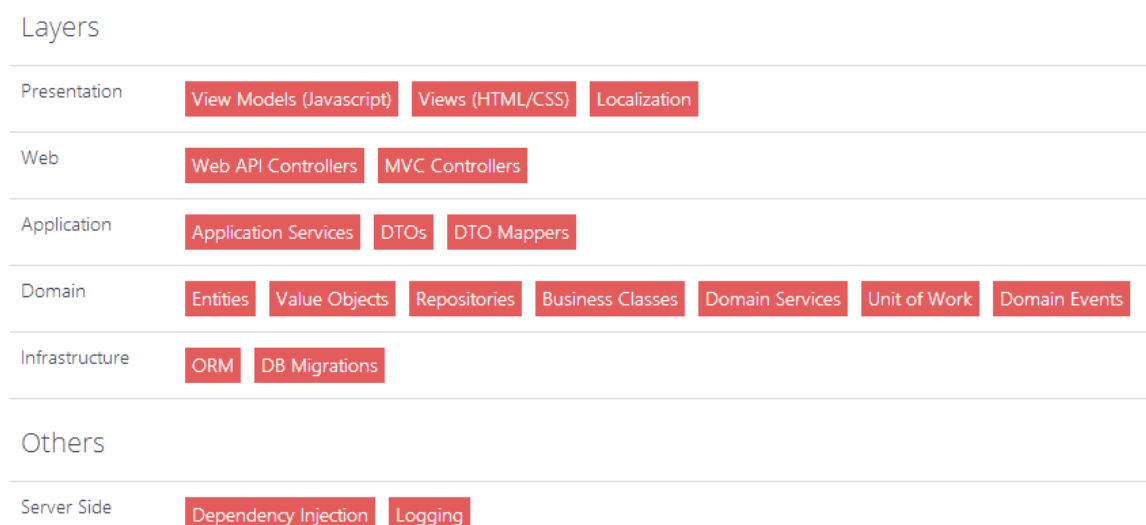
- **展现层 (Presentation)**：提供一个用户界面，实现用户交互操作。
- **应用层 (Application)**：进行展现层与领域层之间的协调，协调业务对象来执行特定的应用程序的任务。它不包含业务逻辑。
- **领域层 (Domain)**：包括业务对象和业务规则，这是应用程序的核心层。
- **基础设施层 (Infrastructure)**：提供通用技术来支持更高的层。例如基础设施层的仓储(Repository)可通过 ORM 来实现数据库交互。

根据实际需要，可能会有额外添加的层。例如：

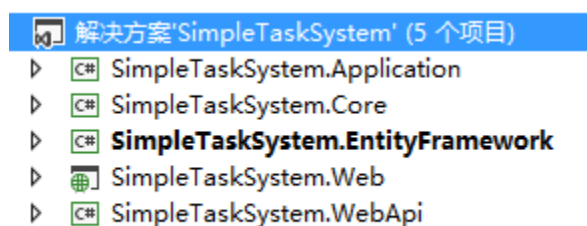
- **分布式服务层 (Distributed Service)**：用于公开应用程序接口供远程客户端调用。比如通过 ASP.NET Web API 或 WCF 来实现。这些都是常见的以领域为中心的分层体系结构。不同的项目在实现上可能会有细微的差别。



## 1.2.2 ABP 的体系结构



一个简单的解决方案，大致包含 5 个项目：



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 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 是一个包含来程序的名称，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，第一次会调用这个方法。在依赖注入注册之前，你可以在这个方法中指定自己的特别代码。举个例子吧：假如你创建了一个传统的登记类，那么你要先注册这个类（使用 **IocManager** 对登记类进行注册），你可以注册事件到 **IOC** 容器。等。

- **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 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。

#### △ 阳铭注：

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 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。配置在每一个应用中都可能会有，比如你有一个网站，你要获取网站的一些自定义基本参数，比如 logo 位置，网站名称，上传文件大小等等。模块化的配置方式和我们之前的做法肯定是不一样的，大家要注意。之前无非就是一个方法 `getConfig` 从对应的表取数据，然后使用。

---

### 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）代码示例如下：

```
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： <http://www.aspnetboilerplate.com/Pages/Documents/Localization>

导航： <http://www.aspnetboilerplate.com/Pages/Documents/Navigation>



### 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<MyModuleConfig>();
```

---

#### △ 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，通过注射 `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。

（1.1、1.2 由阳铭翻译，1.3、1.4 由天道翻译）

## 2 ABP 公共结构

### 2.1 ABP 依赖注入

如果你已经知道依赖注入的概念，构造函数和属性注入模式，你可以跳过这一节。

维基百科说：“依赖注入是一种软件设计模式的一个或多个依赖项注入(或服务)，或通过引用传递，为依赖对象(或客户)和客户端状态的一部分。模式之间建立一个客户的依赖关系的行为，它允许程序设计是松散耦合的，依赖倒置和单一职责原则。它直接对比 **service locator** 模式，它允许客户了解他们所使用的系统找到依赖。”。

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，应用程序服务类依赖于仓储(**repository**)类。看下例子：

```
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 创建 PersonRepository 本身，它成为依赖 IPersonRepository 接口的具体实现，不能使用另一个实现。因此，此方式的将接口与实现分离变得毫无意义。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。

### 2.1.3 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)。

#### 2.1.4 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。

##### (1) 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。

##### (2) 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 `PersonAppService` 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。

### (3) 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。

#### (4) 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>().Li
festylePerThread().WithServiceSelf());
    }
}
```

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

#### (5) 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。

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

作为最佳实践,你可以使用构造函数和属性注入去获取你的类的依赖。任何可能的地方,你都应该这样做。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。

### (7) `IIocResolver` 和 `IIocManager` 接口

有时你可能需要直接创建你的依赖项,而不是构造函数和属性注入。应该尽可能避免这种情况,但它可能无法避免。**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 **IlocManager** 并使用它 **IlocManager.IocContainer** 属性。如果你是在一个静态上下文或不能注入 **IlocManager**，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易容测试。

### 2.1.5 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 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。

---

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。

### 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.如果需要授权访问则它不

可能为空。

- **TenantId**: 当前租户的标识 ID，如果没有当前租户则为 null。
- **MultiTenancySide**: 可能是 Host 或 Tenant。

**UserId** 和 **TenantId** 是可以为 null 的。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。

## 2.3 ABP 日志管理

### 2.3.1 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:

```
using Castle.Core.Logging; //1: 导入日志的命名空间, 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 (就是我们应用中定义的任务) 以后, 通过属性注入的方式来实现。

    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:Remember to drink milk before sleeping!
```

## (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。

## 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。

### 2.3.2 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('...');`

## 2.4 ABP 设置管理

### 2.4.1 介绍

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。

### 2.4.2 定义设置

使用设置之前必须要先定义。**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<MySettingProvider>();` 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。

### 2.4.3 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来为每个用户存储/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 (重写) **User**。

如果没有，我们获取特定的租户值，如果它定义 (重写) **Tenant**。

如果没有，我们获取应用的值，如果它定义 **Application**。

如果没有，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。

#### 2.4.4 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。

#### 2.4.5 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。

#### 2.4.6 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)



## 3 ABP 领域层

### 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，而是通过一连串的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;
    }
}
```

Person 类被定义为一个实体。它具有两个属性，它的父类中有 Id 属性。Id 是该实体的主键。所以，Id 是所有继承自 Entity 类的实体的主键（所有实体的主键都是 Id 字段）。

Id(主键)数据类型可以被更改。默认是 int（int32）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声明 **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（两个实体的 **Id** 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

在很多应用程序中，很多实体具有像 **CreationTime** 的属性（数据库表也有该字段）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。也就是说，为实现这些接口的实体提供了一个通用的编码方式（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）

实体类实现 **IHasCreationTime** 接口就可以具有 **CreationTime** 的属性。当该实体被插入到数据库时，**ABP** 会自动设置该属性的值为当前时间。

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

**Person** 类可以被重写像下面示例一样实现 **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，ABP 会自动设置 **CreatorUserId** 的属性值为当前用户的 **Id**

你可以轻松的实现 **ICreationAudited** 接口，通过派生自实体类 **CreationAuditedEntity**（因为该类已经实现了 **ICreationAudited** 接口，我们可以直接继承 **CreationAuditedEntity** 类就实现了上述功能）。它有一个实现不同 ID 数据类型的泛型版本(默认是 **int**)，可以为 ID（**Entity** 类中的 ID）赋予不同的数据类型。

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，ABP 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** 类已经实现了该功能，直接继承该类就可以实现上述功能），**AuditedEntity** 类有一个实现不同 ID 数据类型的泛型版本(默认是 int)，可以为 ID（**Entity** 类中的 ID）赋予不同的数据类型。

## （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。

---

#### △ 注意：

所有的审计接口和类都有一个泛型模板为了导航定义属性到你的 **User** 实体（例如：**ICreationAudited<TUser>**和 **FullAuditedEntity<TPriamaryKey, TUser>**），这里的 **TUser** 指的进行创建，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<TPriamaryKey, TUser>**类），**TPriamaryKey** 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（和 **Entity<TPriamaryKey>** 实现了 **IEntity<TPriamaryKey>**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<TEntity>, 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<TEntity, TPrimaryKey> 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 = _personRespository.GetAllList();
```

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

**GetAll** 返回 `IQueryable<T>` 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).Take(20).ToList();
```

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<T>`

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<T>` 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<T>` 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<T>` 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<T>` 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);  
TPriamryKey InsertAndGetId(TEntity entity);  
Task<TPriamryKey> InsertAndGetIdAsync(TEntity entity);  
TEntity InsertOrUpdate(TEntity entity);  
Task<TEntity> InsertOrUpdateAsync(TEntity entity);  
TPriamryKey InsertOrUpdateAndGetId(TEntity entity);  
Task<TPriamryKey> 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。**InsertOfUpdate** 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,**InsertOrUpdatedAndGetId** 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(TPriamryKey id);  
Task DeleteAsync(TPriamryKey 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(About 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<TEntity>`(或 `IRepository<TEntity, TPrimaryKey>`)。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<Person>` 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<T>`。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<TEntity>`)

- 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，尽可较少或是不相根据于其它服务。

## 3.3 ABP 领域层—工作单元

### 3.3.1 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,为啥？

- 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)。

### 3.3.2 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 = personRepository;
        _statisticsRepository = statisticsRepository;
    }
}
```

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。见工作单元方法限制章节。

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
statisticsRepository) {

        _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。

### 3.3.3 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);  
}
```

平时时,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)

    };

}
```

我建议可以这么做[UnitOfWork(isTransaction:false)]。我认为这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。

考虑以下示例:

```
[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。

### (6) 工作单员属性的限制(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。

### 3.3.4 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  
}
```

### 3.3.5 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`。

### 3.3.6 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);  
}
```

## 3.4 ABP 领域层—数据过滤器

### 3.4.1 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。

---

### 3.4.2 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。

---

一个小提醒: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 界面默认是启用的。

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 会自动地开启 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。

---

### 3.4.3 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。若你不使用 `using` 语法,过滤器只有在当前的工作单元(unit of work)结尾或你再度启用时才会禁用过滤器。

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事,并且过滤器会维持禁用状态即便是 `using` 语法的结束后。

---

### 3.4.4 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。

### 3.4.5 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);
```

### 3.4.6 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。

### 3.4.7 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。

## 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:

- 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
```

- 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 已自动地记录所有的异常)。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: **EntityCreatedEventData<TEntity>**, **EntityUpdatedEventData<TEntity>** 和 **EntityDeletedEventData<TEntity>**。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<Person>**,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<Person>**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<T>` 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 `EventHandler<TaskEventData>` 来处理这两个事件:

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

当然,你也可以实现 `EventHandler<EventData>` 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 `EventHandler<T>`。范例如下:

```
public class ActivityWriter :
    EventHandler<TaskCompletedEventData>,
    EventHandler<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<T>`:

```
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, `locHandlerFactory`, 通过依赖注入系统, `locHandlerFactory` 可以用来创建或者释放(dispose)处理器。ABP 可以自动化注册 `locHandlerFactory`。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,你或许想要在之后取消注册。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.TaskId));  
  
//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<T>()**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)

## 4 ABP 应用层

### 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<Person>** 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<T>**。数据库连接应会在调用仓储方法后打开。这是因为 **IQueryable<T>** 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<Person>转换成 List<PersonDto>。可以查看 DTO 文档获取相关细节。

---

#### △ 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<User>。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<Permission>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, 那么你不需要定义一个 **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。注意我们没有手动验证 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。

但是你很可能不喜欢手动将 Person 实体转换成 PersonDto。这真的是个乏味的工作。Person 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—DTO 有效性验证

应用程序的输入数据首先应该被检验是否有效。输入的数据能被用户或其他应用程序提交。在 Web 应用中，通常进行 2 次数据有效性检验：包括客户端检验和服务端检验。客户端的检验主要是使用户有一个好的用户体验。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的那些字段输入是无效的。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被应用服务（层）执行，应用服务(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。

应用服务（层）方法得到一个数据传输对象（DTO）作为输入。ABP 有一个 `IValidate` 的接口，DTO 通过实现这个接口能够检验数据的有效性。由于 `IInputDto` 扩展自 `IValidate`，所以你可以直接实现 `IInputDto` 接口来对数据传输对象（DTO）检验其有效性。

#### 4.3.1 使用数据注解

ABP 提供数据注解的特性。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）。

在 `System.ComponentModel.DataAnnotations` 命名空间中，请看 `Task application service` 的实现

```
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性代码（指对 **Description** 属性的验证）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，我们必须添加把这些验证结果添加到结果集合中。（就是将 **ValidationResult** 添加到 **results**）

### 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数据。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，NullIPermissionChecker 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，**authorizationprovider** 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" permission.
}
```

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 特性的方法会有一些限制。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—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)  
{  
    //...  
}  
}
```

上述例子中，**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 ABP 表现层

### 5.1 ABP 展现层—动态 WebApi 层

#### 5.1.1 建立动态 web api 控制器

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);
}
```

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，这里也不异常。

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：

```
DynamicApiControllerBuilder

    .ForAll<IApplicationService>(Assembly.GetAssembly(typeof(SimpleTaskS
ystemApplicationModule)), "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。

### 5.1.2 使用动态 js 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

你可能希望自定义 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。

### （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：

```
(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>
```

#### (4) Durandal 支持

**ABP** 框架可以注入服务到 **Durandal** 框架，如下：

```
define(['service!tasksystem/task'],
    function (taskService) {
        //taskService can be used here
    });
```

**ABP** 框架配置 **Durandal**（实际上是 **Require.js**？）来解析服务代理并注入合适的 **js** 到服务代理。

## 5.2 ABP 展现层—本地化

### 5.2.1 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。

### 5.2.2 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：

```
<?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>
```

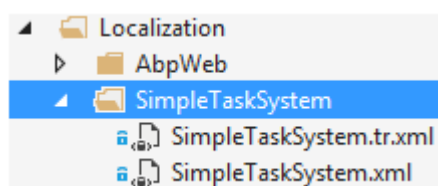
`XML` 文件必须使用 `UTF-8` 编码，文件名确保唯一性。你可以使用属性-值或内部文本（如最后一个）设置本地化文本。使用下面的方式注册来源：

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

```
    )  
    )  
);
```

这样就完成了模块的预初始化事件。

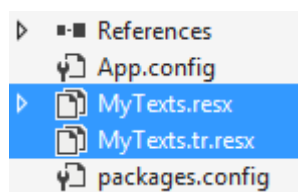
本地化配置文件必须为每个语言设置一个唯一的名字(单任务系统)和一个包含 xml 文件目录路径。在本例中, 它们将放置在 **Localization/SimpleTaskSystem** 文件夹中。



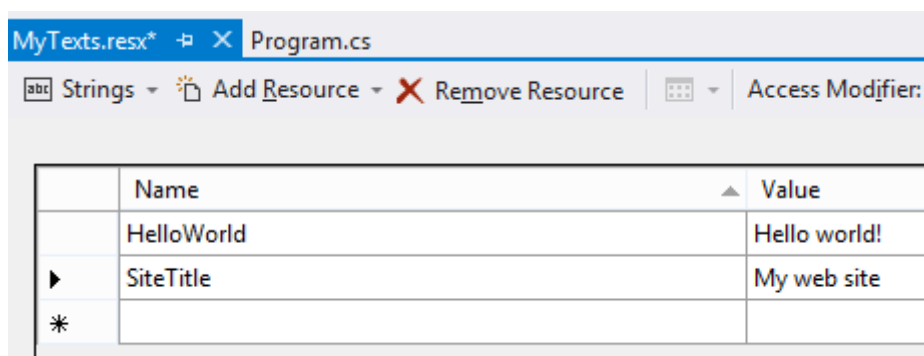
**SimpleTaskSystem.xml** 是默认的本地化语言配置文件(它没有包含语言代码)  
**SimpleTaskSystem.tr.xml** 包含了土耳其语言。当请求一个文本时, **ABP** 获取当前语言 xml 文件, 使用 **Thread.CurrentThread.CurrentUICulture**。如果不存在当前的语言。它会从默认的语言 xml 中获取。使用 xml 增加了灵活性, 即使没有经验的用户也很容易修改和发布。

## (2) 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。

### 5.2.3 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();
    }
}
```

L 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="@L("EnterDescriptionHere")" required></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,
```

```
optionsCaption: '@L("SelectPerson")' class="form-control"></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.SimpleTaskSystemWebViewPageBas
e">
```

#### (4) 在 javascript(In javascript)

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 OK。

```
<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'`。

## 5.2.4 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。

## 5.3 ABP 展现层—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。

### 5.3.1 AJAX

现代的应用经常会使用 AJAX，尤其是单页应用，几乎是和服务器通信的唯一手段，执行 AJAX 通常会有以下步骤：

在客户端，你需要提供一个 URL，选择一个和服务器通信的方法

(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，成功时你需要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: 'Dougles 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.

例子很简单, 但是有些事情是 `ABP` 框架帮你做的, 我们可以看一下返回值

### (1) 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。

## (2) 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 occurred during your request!", //未捕捉到的异常，通常为系统异常，会自动记录日志，具体提示信息在配置文件配置，可以搜索一下，如果是业务抛出的 UserFriendlyException 异常，message 为具体的错误信息
    "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
}
```

## (3) 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);

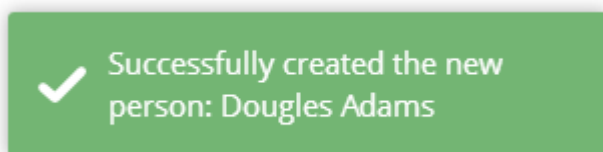
});
```

### 5.3.2 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。

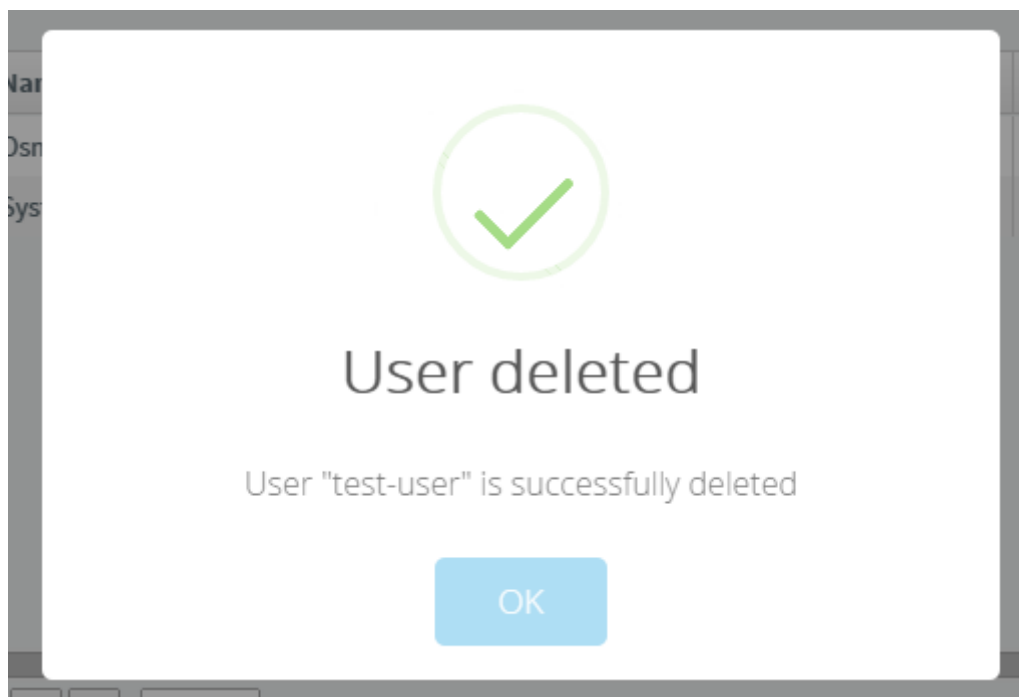
### 5.3.3 消息

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

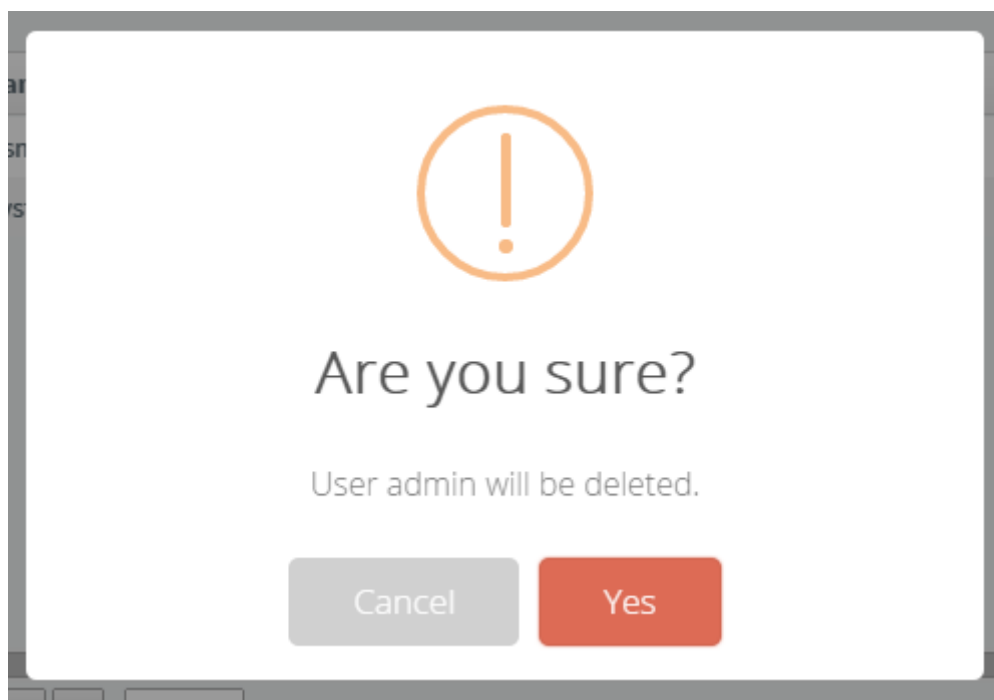


**Confirmation 确认对话框**

```
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  
        }  
    }  
);
```

样子





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`。

### 5.3.4 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。

#### 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
```

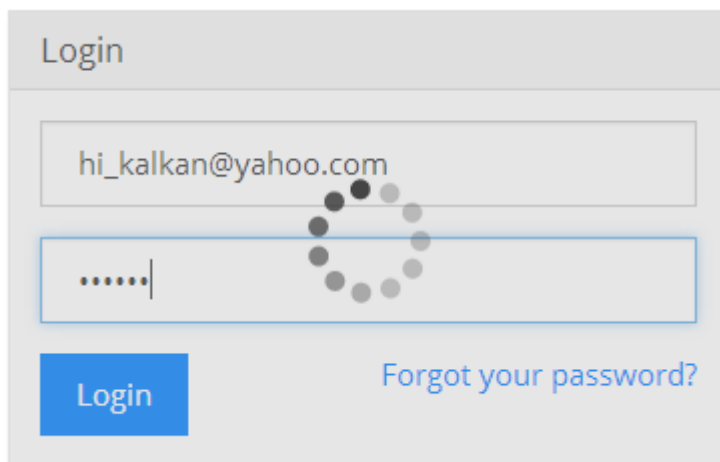
UI Block API 使用 `blockUI` 这个 js 库来实现效果的，如果使用这个 api 需要在页面引用 `blockUI` 的 js 库和 `abp.blockUI.js` 文件

UI Busy API 指示页面繁忙的 API，如 ajax 请求中

例子

```
abp.ui.setBusy('#MyLoginForm');
abp.ui.clearBusy('#MyLoginForm');
```

效果



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。

例如

```
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**。

### 5.3.5 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。

### 5.3.6 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。

#### (2) 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'
```

## 5.4 ABP 展现层—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。

### 5.4.1 创建菜单

一个应用程序可能包含不同的模块，而每个模块都可能拥有它自己的菜单项。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
  - User management
  - 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>();
```

### 5.4.2 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。

## 5.5 ABP 展现层—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。

### 5.5.1 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。

### 5.5.2 非 Ajax 请求

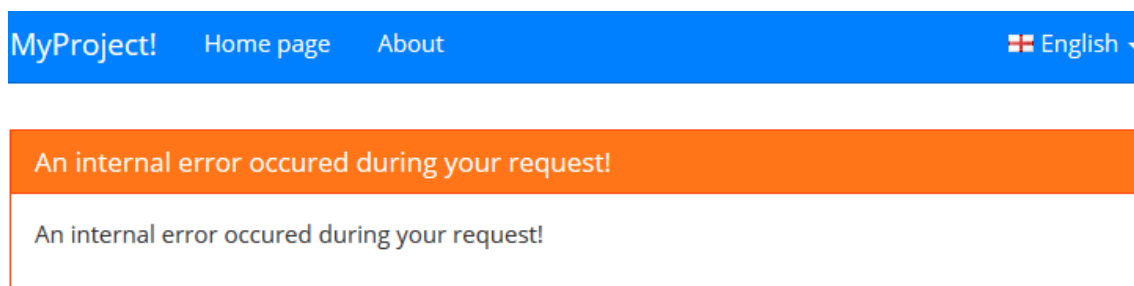
如果不是 Ajax 请求，那么将会显示一个错误页面。

#### (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。

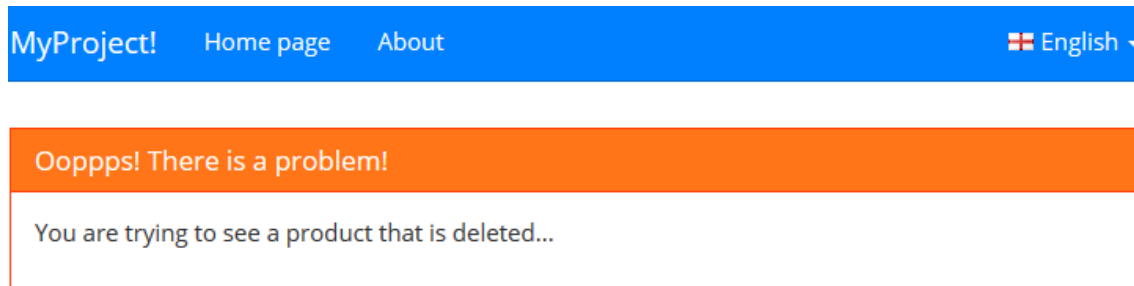
#### (2) 用户异常友好化 (UserFriendlyException)

`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("A sample exception message...");
}
```

```
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)。

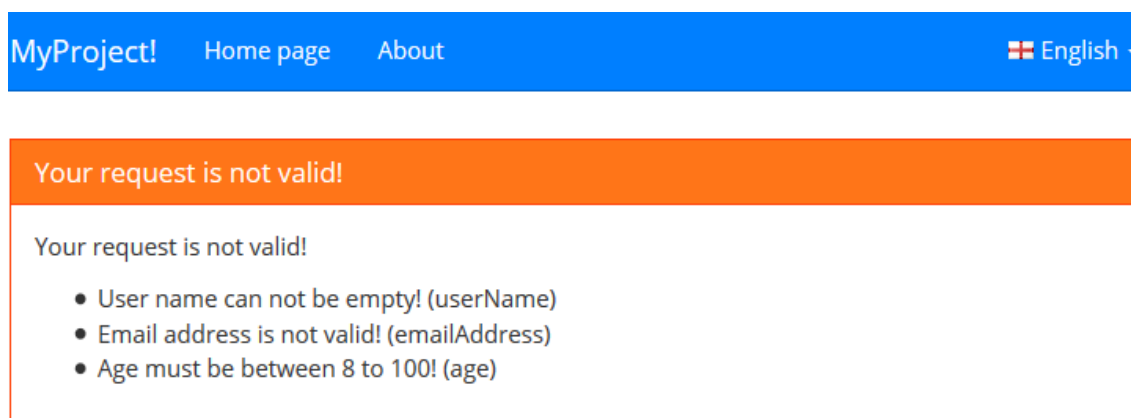
### (3) 错误模型(Error model)

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`。





### 5.5.3 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](#)。

### 5.5.4 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](#)。

## 5.6 ABP 展现层—嵌入资源文件

(5.1、5.2、5.4 由 NoZero 翻译，5.3 由菜刀翻译，5.5 由 Carl 翻译，5.6)

## 6 ABP 基础设施层

### 6.1 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。

---

#### 6.1.1 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。

---

△ 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》](#)

---

#### 6.1.2 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("MyConnectionStringName")
{

}

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。

### 6.1.3 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 :
EfRepositoryBase<SimpleTaskSystemDbContext, TEntity, TPrimaryKey>
    where TEntity : class, IEntity<TPrimaryKey>
{
    public
SimpleTaskSystemRepositoryBase(IDbContextProvider<SimpleTaskSystemDbCon
text> 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<SimpleTaskSystemDbCon
text> 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<SimpleTaskSystemDbContext, TEntity, TPrimaryKey>`。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<Person>` 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<TEntity>`（或者 `IRepository<TEntity, TPrimaryKey>`）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<Task>` 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。

## 6.2 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。

---



### 6.2.1 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](#)》。

### 6.2.2 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<T>` 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。

---

### 6.2.3 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<Person>` 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<TEntity>`（或者 `IRepository<TEntity, TPrimaryKey>`）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<Task>` 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, TPrimaryKey>)
}

public class TaskRepository : MyRepositoryBase<Task>, ITaskRepository
{
    public TaskRepository(ISessionProvider sessionProvider)
        : base(sessionProvider)
    {
    }

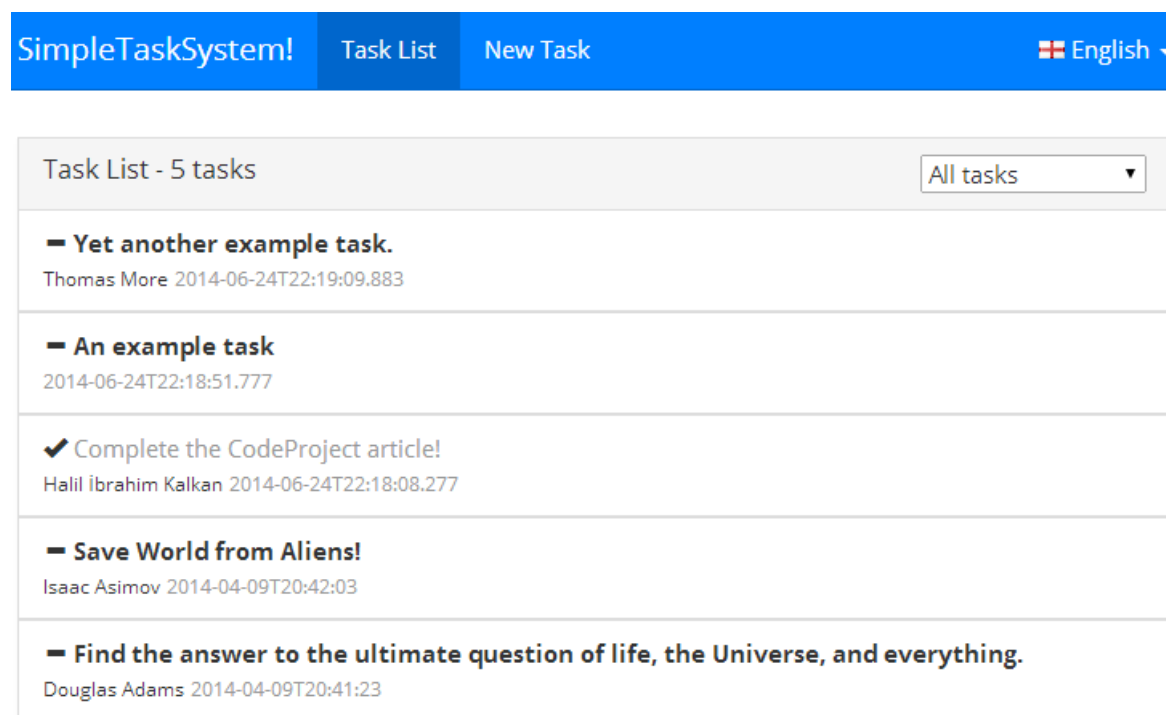
    // 添加本仓储类的专用方法
}
```

(6.1、6.2 由深圳-Leo 翻译)

## 7 ABP 实例一：ASP.NET Boilerplate

本实例是基于 ABP 而打造的一个综合应用，详见：

<https://github.com/aspnetboilerplate/aspnetboilerplate-samples>。



### 7.1 引言

DRY——别重复你的行为!这是一名优秀的开发人员在开发一个软件时重要的概念之一。我们从简单的方法、类以及模块开始实现这个原则。那么开发一个全新的WEB 应用程序呢?当开发企业级应用系统时，我们软件开发人员有相似的需求。

企业级网站应用程序需要登入网页,用户/角色管理这些基础设施,用户/应用程序配置管理,多语言...诸如此类的。同样地,一个高质量且大型的软件会做的最佳实践,像是分层架构(Layered Architecture),领域驱动设计(DDD, Domain Driven Design),依赖注入(DI, Dependency Injection)。同样地,我们使用对象关系映射工具(ORM,

Object-replational Mapping),数据库迁移(Database Migration),日志(Logging)... 等等。当涉及到用户界面时(UI),也没有太大的区别。

启动一个全新的企业级网站应用程序是一个艰难的工作。这是因为所有的应用程序都会有一些相同且重复的任务,我们重复做着相同的工作。许多公司会针对这些重复的任务来开发他们专属的应用程序框架(Application Framework)或是类库(Library),来避免重复开发相同的东西。而其它公司则是复制一些现存的应用程序某部份的程序代码到新网站上,并且会预备好一个开发的起点(start point)。第一个方法在你的公司足够大且有时间开发那些框架/类库的时候非常好用。

作为一个软件架构师,我也为我的公司开发了一些框架。但是,仍有些地方让我感到不太舒服:许多公司重复相同的任务。如果我可以分享更多知识/技术,那么这些重复任务就会减少吗?如果 DRY 理论是套用到全部而不是只有部份项目上或是某些公司上呢?这听起来就像个乌托邦(注: 比喻一个理想中完美的世界),但我认为现在就可以开始!

## 7.2 什么是 ASP.Net Boilerplate?

ABP 是一个新的现代网站应用程序的初始环境;它使用了最佳实践和许多受欢迎的工具。它的目标是成为一个所有人都可以使用,一个适用各种项目目标的应用程序框架以及项目样版。它做了那些事呢?

### 服务器端

- 基于最新版的 ASP.Net MVC 和 WebAPI
- 实现了领域驱动设计(Domain Driven Design)(实体,仓储,领域服务,应用服务,数据传输对象,工作单元... 等等)
- 实现了分层架构(Layered Architecture)(领域层,应用层,表现层和基础设施层)
- 提供一个开发上可重用的基础结构以及针对大型项目所需要的模块,并且这些模块都是可组合式的
- 采用那些你已经(或可能)已经在使用的最受欢迎框架/类库
- 提供一个基础且更易于使用的依赖注入机制(使用 Castle Windsor 作为 DI



的容器)

- 提供一个严谨的对象模型和基类,轻易的通过对象关系映射 (Object-Relational Mapping)。 (直接支持 EntityFramework 和 NHibernate)。
- 支持且实现了数据库迁移(database migration)。
- 引入了一个简单且具弹性的多语言(localization)系统。
- 为服务器端的全局领域事(domain event)件引入了事件总线(EventBus)
- 管理异常处理和验证
- 为应用服务创建了动态 Web API 层
- 提供基类和帮助类予当你需要实现某些通用任务时
- 使用约定大于配置

#### 客户端

- 为单页面应用(Single-Page Application)(有 AngularJs 和 Durandaljs)和多页面应用提供项目样版。样版都是基于 Twiter Bootstrap。
- 流行的 Javascript 类库都被引入进来,并且都已套用预设的配置
- 为了让调用应用服务(注: 使用动态的 Web API 层)变得更容易而创建动态 Javascript 代理(dynamic javascript proxies)
- 为了某些传递讯息型任务(Summon task)而引入一些独特的 API,而所谓的传递讯息任务像是: 显示警告和通知,阻塞式 UI,发出 Ajax 请求....等等。

除了上述的那些基础设施,还有一个名叫 zero 的模块正在开发中。它将会提供基于角色和权限的授权系统(实现最终版本的 ASP.Net Identity Framework),也提供配置系统,多租户...等等能力。

## 7.3 ABP 不适用于那些场合?

ABP 使用一些最佳实践提供了一个应用程序开发模型。它拥有一些基类,接口以及许多工具,这些都让它能够更易于建构出一个可维护的大型应用程序,但....

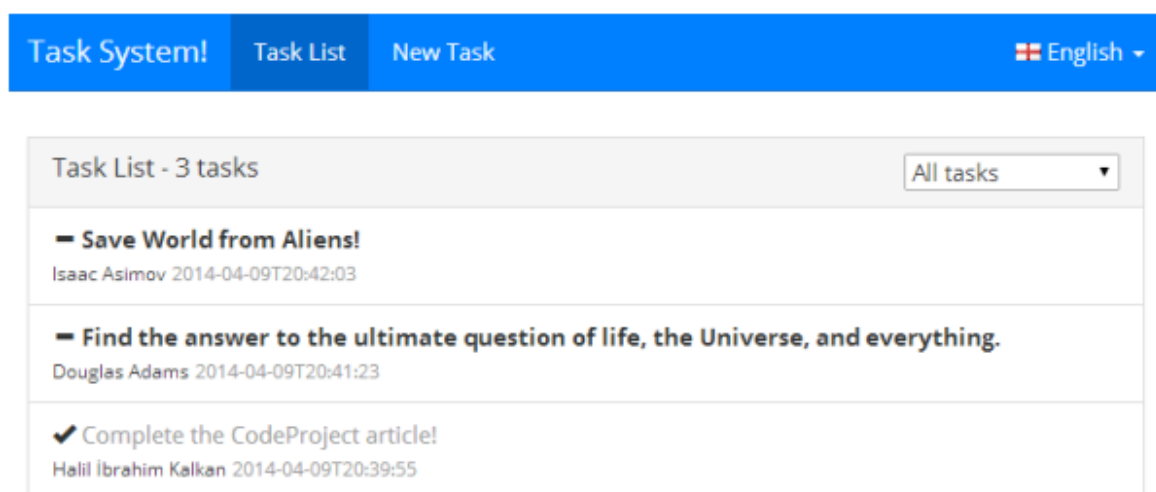
- 它不是一种快速开发工具 RAD(Rapid Application Development), 它不会提供足够的基础设施让你能不需要编写代码就能够建构出应用程序。但是,它只提供了一个基础设施让你能够以最佳实践开发应用程序。

- 它不是一个程序代码产生器。它具有能够在执行时期动态地产生出程序代码的能力,但无法产生开发时期的程序代码。
- 它不是一个无所不包的框架。实际上,它使用许多常用特定功能目标的工具/类库(像是在 ORM 方面采用 NHibernate 和 EntityFramework, 纪录方面采用 Log4Net, Castle Windsor 作为 DI 容器, AngularJs 则是 SPA 的框架)

## 7.4 开始

在本篇文章中,我将会展示该如何使用 ABP 来开发一个单页面且响应式的网站应用程序(Single-Page Responsive Web Application),我将其称为 ABP。我将会使用 DurandalJs 作为 SPA 框架并且我以 NHibernate 作为 ORM 框架。我已准备了另一篇文章,该篇文章实现了相同的应用程序,但使用的是 AngularJs 和 EntityFramework。

这个示例应用程序名为简易型任务系统(Simple Task System),并且它是由两个页面所组成: 其一是为了要列出所有任务,另一个则是添加新的任务。一个任务可以被关联到人员(Person),任务的状态可以是启用或完成。应用程序在多语言方面支持两种语言。应用系统的任务列表贴图如下所示:



## 7.5 使用模板创建空的网站应用程序

ABP 提供一个启动样版(startup template)给新的项目(你也是可以手动创建你的项目然后再使用 nuget 来取得 ABP 相关组件,但是样版的做法会更简单)。前往使用

样版创建你的应用程序。你可以选择 SPA(Single-Page Application)项目然后挑选你要使用 AngularJs 还是 DurandalJs。或是你可以选择 MPA(典型,多页面应用程序)项目。接着你可以选择 EntityFramework 或 NHibernate 来作为你的 ORM 框架。

### ASP.NET Boilerplate Templates

Easily start using ASP.NET Boilerplate with your favorite frameworks! Select tools you like and let it to create an **NLayered** solution for you.

#### 1. Select Architecture



#### 2. Select Object-Relational Mapper



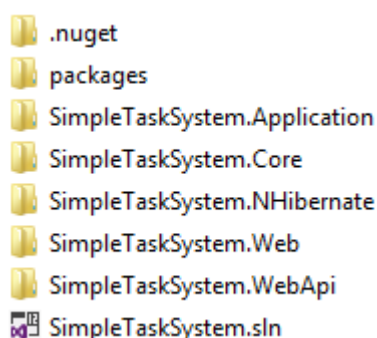
#### 3. Choice your project's name

Enter your solution's name (without spaces)

SimpleTaskSystem

CREATE MY PROJECT!

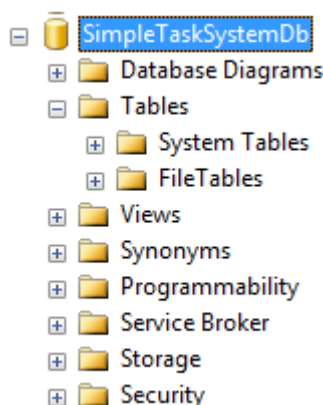
将这个项目命名为 SimpleTaskSystem,并且创建了一个 SPA 项目,其使用 Durandaljs 和 NHibernate。以 zip 文件的方式下载项目程序代码。解开 zip 文件后,得到一个解决方案,其中已经包含 DDD 的每一层类库项目:



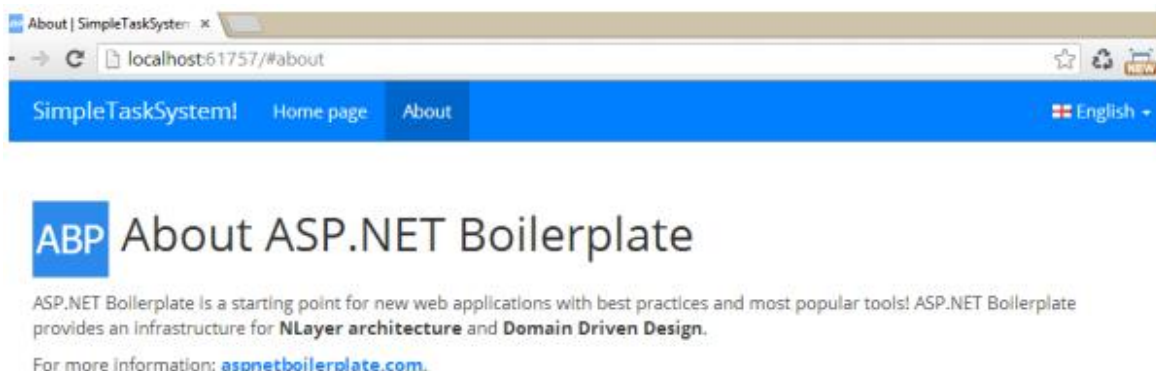
这个刚刚创建出来的项目,它用的是 .Net Framework 4.5.1 版本,我建议使用 Visual Studio 2013 来开启这个项目。这里唯一有一个在执行此项目前需要准备的事情就是创建一个数据库。SPA 样版假定你是使用 SQL Server 2008 或更新的版本。但是你可以很容易地替换成其它的 DBMS。详见项目的 web.config 中的连接字符串:

```
<add name="Default" connectionString="Server=localhost; Database=SimpleTaskSystemDb; Trusted_Connection=True;" />
```

你可以在这里修改连接字符串。我不修改数据库的名称,所以我会建立一个空的数据库,其名为 SimpleTaskSystemDB 于 SQL Server 中:



至此,你的项目已经能够执行了!将其使用 VS2013 开启并按下 F5:



样板是由两个页面所组成: 其一是首页(Home),另一个则是关于(About)页面。它的本地化语言支持英语和土耳其语。并且它是单页应用(Single-Page Application)!试着浏览于这两页之间,你将会看到只有内容区块会被改变,导航菜单则是固定的,所有的 JavaScript 和样式都只会加载一次。并且它是响应式(Responsive)的。试着去调整一下浏览器的大小。

现在,我将会展示如何逐层修改这个应用程序成为简易任务系统应用程序。

## 7.6 领域层

对于业务概念的表达和业务状态的信息以及业务逻辑的响应"(Eric Evans)[2]。在领域驱动设计(DDD)中,核心层(core layer)即为领域层。领域层定义你的实体,实现

你的业务逻辑...等等。

### 7.6.1 实体-Entities

实体即为 DDD 中核心概念之一。Eric Evans 将其描述为"一个对象,其并非简单地由它的属性所定义,而是由其一系列的活动轨迹(行为)和 ID 所定义"。因此,实体拥有 Id 且储存在数据库中(译者注:数据库储存实体在每次经过活动后状态的变更,这些变更纪录即为实体的活动轨迹。和现实生活中:一个人的价值在于他做了什么是一样的)。

我第一个实体即为 Task:

```
public class Task : Entity<long> {  
  
    public virtual Person AssignedPerson { get; set; }  
    public virtual string Description { get; set; }  
    public virtual DateTime CreateionTime { get; set; }  
    public virtual TaskState State { get; set; }  
  
    public Task() {  
        CreationTime = Datetime.Now;  
        State = TaskState.Active;  
    }  
}
```

这是一个简单的类,其继承自 Entity 这个基类,并且它以 long 作为其主键的型别。

TaskState 是一个 enum,其成员有 Active 和 Completed。第二个实体即为 Person:

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

在这个简单的程序中,一个任务会关联到一个人员。

实体都实现了 ABP 中的 IEntity<TPriamryKey>接口。因此,倘若你实体的主键的数据型别是 long,它必须实现 IEntity<long>。倘若你的实体主键是 int,则你就不需要定义主键型别而是直接实现 IEntity 接口。实务上,你可以只继承 Entity 或 Entity<TPriamryKey>(如上面示例中 Task 和 Person 这两个类那样)。IEntity 为实体

定义了 `Id` 属性。

### 7.6.2 仓储-Repository

领域和数据映射层之间的中介者,使用一种类似集合的接口来存取领域对象"(Martin Fowler)[3]。仓储,实际上,它被用来执行领域对象的数据库操作(实体或值类型)。

一般来说,给每个实体(或是聚合根)创建专属的仓储是常见的做法。ABP 为每个实体都提供了默认的仓储(我们将会见到如何使用这些默认仓储)。如果我们定义一个额外的方法,我们可以实现 `IRepository` 接口。我在 `Task` 仓储中实现了这个接口。

```
public interface ITaskRepository : IRepository<Task, long> {  
    List<Task> GetAllWithPeople(int? assignedPersonId, TaskState? state);  
}
```

能够为仓储量身打造接口是最好的。因此,我们可以由目前实现的程序代码中去拆解出接口。`IRepository` 接口为仓储定义了最通用的一些方法:

```
Count(System.Linq.Expressions.Expression<System.Func<TEntity,bool>>)  
Count()  
Delete(System.Linq.Expressions.Expression<System.Func<TEntity,bool>>)  
Delete(TPrimaryKey)  
Delete(TEntity)  
FirstOrDefault(System.Linq.Expressions.Expression<System.Func<TEntity,bool>>)  
FirstOrDefault(TPrimaryKey)  
Get(TPrimaryKey)  
GetAll()  
GetAllList(System.Linq.Expressions.Expression<System.Func<TEntity,bool>>)  
GetAllList()  
Insert(TEntity)  
InsertAndGetId(TEntity)  
InsertOrUpdate(TEntity)  
InsertOrUpdateAndGetId(TEntity)  
Load(TPrimaryKey)  
LongCount(System.Linq.Expressions.Expression<System.Func<TEntity,bool>>)  
LongCount()  
Query<T>(System.Func<System.Linq.IQueryable<TEntity>,T>)  
Single(System.Linq.Expressions.Expression<System.Func<TEntity,bool>>)  
Update(TEntity)
```

该接口定义了一系列基本的 **CRUD** 方法。因此,所有的仓储都会自动地实现所有列在这个接口上的所有方法。此外,除了标准的基本方法,你可以为仓储添加其所需要



的方法。如我所定义的 `GetAllWithPeople` 方法。

### 7.6.3 关于命名空间

当你研究这个示例应用程序的源代码时,你将会看类组件的意图以及其领域;相互关联的类/接口/枚举(enum)都在同一个命名空间中,而不是根据基础设施进行封装(packaging by infrastructure)。我想要将 `Task` 类迁移到 `TaskSystem.Entities` 命名空间中,`ITaskRepository` 则是搬到 `TTaskSystem.Tasks` 命名空间中,这是因为彼此之间相互关联。这更加契合领域驱动设计的本质。因此,我认为把所有的实体都搬到 `TaskSystem.Entity` 命名空间底下不是一个好的做法。或许你不认同,我能够理解为什么,因为我之前也曾这么做过一段时间直到前一阵子为止。但是在我看到一些问题后,我很强烈的建议把相关联的类/接口/枚举(enum)放在同一个命名空间底下,或许是不同的类库项目,但是却是同一个命名空间。你可以阅读 **Eric Evans** 所写的领域驱动设计一书中的"架构导向封装的陷阱"(The pitfalls of Infrastructure-Driven Packaging) 章节。

## 7.7 基础设施层

"支持泛型技术的能力之后,就能够具有更高的质量"(Eric Evans)。利用第三方类库和框架(例如 **ORM**)来实现你应用程序的抽象部份。在这个应用程序中,我将会使用基础设施层来:

- 使用 **FluentMigrator** 来进行数据库迁移系统
- 使用 **NHibernate** 和 **FluentNHibernate** 来实现仓储以及实体映射

### 7.7.1 数据库迁移

渐进式数据库设计(Evolutionary Database Design): 在几年之前,我们就已经有了许多让数据库设计就如同应用程序开发的技术。对于敏捷开发来说,这是一项非常重要的功能。**Martin Fowler** 在它的网站上如是说[3]。数据库迁移对于成就这项创意来说是非常关键的技术。对于没有这项技术的生产环境中其数据库维护来说,比有这

项技术的还要难很多。就算你只有一个重要的已上线系统也是一样的。

**FluentMigrator**[4]对于数据库迁移来说是一个好工具。它支持许多通用的数据库系统。我的在 **Person** 和 **Task** 数据表的迁移程序代码如下:

```
[Migration(2014041001)]

public class _01_CreaePersonTable : AutoReversingMigration {

    public override void Up() {

        Create.Table("StsPeople")

            .WithColumn("Id").AsInt32().Identity().PrimaryKey().NotNullable(
)

            .WithColumn("Name").AsString(32).NotNullable();

        Insert.IntoTable("StsPeople")

            .Row(new { Name = "Douglas Adams" } )

            .Row(new { Name = "Isaac Asimov" } )

            .Row(new { Name = "George Orwell" } )

            .Row(new { Name = "Thomas More" });

    }

}

[Migration(2014041002)]

public class _02_CreateTasksTable : AutoReversingMigration {

    public override void Up() {

        Create.Table("StsTasks")

            .WithColumn("Id").AsInt64().Identity().PrimaryKey().NotNullable(
)

            .WithColumn("AssignedPersonId").AsInt32().ForeignKey("TsPeople",
"Id").Nullable()

            .WithColumn("Description").AsString(256).NotNullable()

            .WithColumn("State").AsByte().NotNullable().WithDefaultValue(1)

            .WithColumn("CreationTime").AsDateTime().NotNullable().WithDefaul
t(SystemMethods.CurrentDateTime);

    }

}
```



在 **FluentMigrator** 中,数据库迁移是被定义在一个继承自 **Migration** 的类中。**AutoReversingMigration** 是一个快捷方式方法,当你的迁移允许自动地回滚。一个迁移类应该要拥有 **MigrationAttribute** 特性。它定义了迁移类的版本号码。所有的迁移都会依版本号码的顺序被套用。它可以是任意一个 **long** 所容许的数字范围内的某个数字。我使用迁移类的创建日期和同一天内再次启用所产生的递增值相加的结果作为版本号码。(例: 假设在 2014 年 4 月 24 日内启用第二次则其版本号为 2014042402)。要怎么设计你可以自行决定,只是要注意数字顺序的关联性。

**FluentMigrator** 储存最后套用到数据库中的数据表其版本号码。因此,它只会套用比目前数据库所使用的迁移程序代码版本号码还要大(新)的迁移程序代码。默认上,它使用 **VersionInfo** 数据表来纪录。如果你想要改变数据表的名称,你可以创建如下类:

```
[VersionTableMetaData]
public class VersionTable : DefaultVersionTableMetaData {
    public override string TableName {
        get { return "StsVersionInfo" ; }
    }
}
```

如你所见,我写了一个能够把所有数据表名称前缀 **Sts(Simple Task System)** 的类。这对于模块化应用程序来说是很重要的,因此,所有的模块都可以拥有它们专属的前缀,这可用来识别那些数据表是属于特定模块的。

创建我的数据表于数据库中。我使用 **FluentMigrator** 所提供的 **Migrate.exe** 工具于命令提示字符中:

```
Migrate.exe /connection "Server=localhost; Database=SimpleTaskSystemDb;
Trusted_Connectiotsn=True;" /db sqlserver /target
"SimpleTaskSystem.Infrastructure.NHibernate.dll"
```

为了方便,ABP 样版已包含了 **RunMigrations.bat** 文件。在我编译完我的项目于调试模式(Debug)时,我便执行了"RunMigrations.bat":

```
C:\WINDOWS\System32\cmd.exe

===== FluentMigrator =====
Source Code:
  http://github.com/schambers/fluentmigrator
Ask For Help:
  http://groups.google.com/group/fluentmigrator-google-group

[+] Using Database sqlserver and Connection String Server=localhost; Database=SimpleTaskSystemDb; Trusted_Connection=True;

VersionMigration migrating
[+] Beginning Transaction
[+] CreateTable StsVersionInfo
[+] Committing Transaction
[+] VersionMigration migrated

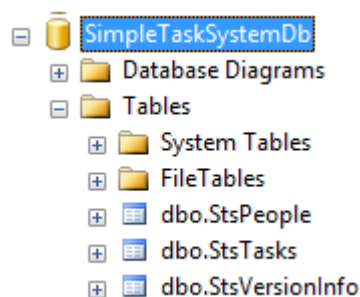
VersionUniqueMigration migrating
[+] Beginning Transaction
[+] CreateIndex StsVersionInfo (Version)
[+] AlterTable StsVersionInfo
[+] CreateColumn StsVersionInfo AppliedOn DateTime
[+] Committing Transaction
[+] VersionUniqueMigration migrated

VersionDescriptionMigration migrating
[+] Beginning Transaction
[+] AlterTable StsVersionInfo
[+] CreateColumn StsVersionInfo Description String
[+] Committing Transaction
[+] VersionDescriptionMigration migrated

2014041001: _01_CreatePersonTable migrating
[+] Beginning Transaction
[+] CreateTable StsPeople
[+] -> 1 Insert operations completed in 00:00:00.0040048 taking an average of 00:00:00.0040048
[+] Committing Transaction
[+] 2014041001: _01_CreatePersonTable migrated

2014041002: _02_CreateTasksTable migrating
[+] Beginning Transaction
[+] CreateTable StsTasks
[+] CreateForeignKey FK_StsTasks_AssignedPersonId_StsPeople_Id StsTasks(AssignedPersonId) StsPeople(Id)
[+] Committing Transaction
[+] 2014041002: _02_CreateTasksTable migrated
[+] Task completed.
```

如你所见,两个迁移文件都已执行完毕并且数据表也都创建好了:



### 7.7.2 实体映射

为了要取得/储存实体到数据库中,我们应该要能够映射实体与数据库中的数据表。NHibernate 中有一些参数可完成这个工作。在这里,我使用手动设定 **Fluent Mapping** 的方式(你可以使用约定的方式自动化映射,见 **FluentNHibernate** 的官方网站)。

```
public class PersonMap : EntityMap<Person> {
    public PersonMap()
        : base("StsPeople") {
        Map(x => x.Name);
    }
}

public class TaskMap : EntityMap<Task, long> {
    public TaskMap()
        : base("StsTasks") {
        Map(x => x.Description);
        Map(x => x.CreationTime);
        Map(x => x.State).CustomType<TaskState>();

        References(x => x.AssignedPerson)
            .Column("AssignedPersonId")
            .LazyLoad();
    }
}
```

**EntityMap** 是一个 **ABP** 所提供的类,它能够自动地映射 **Id** 属性并且能够在构造函数中取得数据表名称。因此,我正是从它继承而来并且映射实体中其它的属性。

### 7.7.3 仓储实现

我针对 **Task** 仓储定义了一个接口(**ITaskRepository**)于领域层中。在这里,我将会以 **NHibernate** 实现这个接口:

```
public class TaskRepository : NhRepositoryBase<Task, long> ,
ITaskRepository {
    public List<Task> GetAllWithPeople(int? assignedPersonId,
TaskState? state) {
        //在仓储方法中,我们不处理创建/释放 DB 联机 (Session) 和事务。ABP 会处理它

        var query = GetAll(); //GetAll() 回传 IQueryable<T>,因此我们可以用它
来查询

        //var query = Session.Query<Task>(); //可选的,我们可以直接使用
NHibernate 的 Session 来达成

        //添加一些 Where 条件

        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) //取得指定的 person
于单一查询中

            .ToList();
    }
}
```

**NhRepositoryBase** 实现了所有定义在 **IRepository** 接口中的方法。因此,你只需要实现你自己定义的那些方法就可以了,一如我实现了 **GetAllWithPeople**。

**GetAll()**方法回传 **IQueryable<TEntity>**,因此,你可以写附加条件直到调用 **ToList()** 为止。

当标准的仓储方法已足够实体使用,就不需要为实体再定义或是实现一个额外的仓储。

## 7.8 应用层

"定义一个软件能够做的工作,并且引导领域对象解决特定问题"(Eric Evans)。在理想的应用程序中,应用层并不引入领域信息和业务逻辑(这在实际上是不太可能的,但我们仍要尽可能做到)。它是表现层和领域层之间的中介者。

### 7.8.1 应用服务及数据传输对象

应用服务提供应用层所需的功能。一个应用服务方法以数据传输对象作为参数,并且回传数据传输对象。直接回传实体(或其它领域对象)会有很多问题(像是数据隐藏,串行化以及延迟加载等问题)。我强烈的建议不要以实体或任何领域对象作为应用服务的输出/入对象。应该是要使用数据传输对象作为输出/入对象才对。这样,表现层就完全与领域层隔绝。

因此,就让我们先从简单的开始,Person 应用服务:

```
public interface IPersonAppService : IApplicationService {  
    GetAllPeopleOutput GetAllPeople();  
}
```

所有的应用服务按约定都实现了 `IApplicationService` 接口。它确保依赖注入且提供某些 ABP 内建的功能(像是验证)。我只定义了一个方法,其名为 `GetAllPeople()` 并且它回传一个 DTO 名为 `GetAllPeopleOutput`。我命名 DTO 时,都是以方法名称加上 Input 或是 Output 作为后缀字。见 `GetAllPeopleOutput` 类:

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

一个输出型 DTO 实现了 `IOutputDto`(它继承了 `IDto`)。它并不做任何事,但是它是用来按约定识别 DTO 之用。`PersonDto` 是另一个 DTO 类,用于传递 Person 信息给表现层:

```
public class PersonDto : EntityDto {
```

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

**EntityDto** 是另一个 **ABP** 的帮助类,其定义了 **Id** 属性并且自动地实现了 **IDto** 接口。**IPersonAppService** 的实现方式如下:

```
public class PersonAppService : IPersonAppService // 可选地,你可以像我的  
TaskService 类那样继承自 ApplicationService  
{  
    private readonly IRepository<Person> _personRepository;  
  
    //ABP 让我们可以直接地注入 IRepository<Person> (而不需要创建任何仓储类)  
    public PersonAppService(IRepository<Person> personRepository) {  
        _personRepository = personRepository();  
    }  
  
    public GetAllPeopleOutput GetAllPeople() {  
        return new GetAllPeopleOutput {  
            People =  
Mapper.Map<List<PersonDto>>(_personRepository.GetAllList())  
        };  
    }  
}
```

**PersonAppService** 以 **IRepository<Person>** 作为其构造函数的参数。**ABP** 内建依赖注入系统并且它使用的是 **Castle Windsor**。所有的仓储和应用服务都能够自动地注册到 **IoC(Inversion of Control)** 容器中作为暂时性对象(**transient object**)。因此,你不需要去思考 **DI** 的细节。同样地,**ABP** 可以创建一个要准的仓储给指定的实体而不需要定义或实现一个仓储。

**GetAllPeople()** 方法简单的从数据库中取得所有 **People** 的列表(使用 **ABP** 的易用实现方式),并且使用 **AutoMapper[6]** 将其转型为 **PersonDto List**。**AutoMapper** 以不可思议(**incredibly**)的简单方式-仅靠约定(假如需要也可以配置使用)就能够将一个类映射成另一个类。我只定义了一行程序代码就能够组态映射的方式:

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

这段程序代码会在应用程序启动时执行并且创建出映射器。接下来,当我有需要

的时候,我只要调用 `Mapper.Map` 方法就可以使用 `Person` 对象来创建出 `PersoDto` 对象。欲取得更多 `AutoMapper` 的详情,可见其官方网站[6]。另一个应用服务 `TaskAppService` 的实现方式如下:

```
public class TaskAppService : ApplicationService, ITaskAppService {
    //这些成员的设定都会由构造函数注入来完成

    private readonly ITaskRepository _taskRepository;
    private readonly IRepository<Person> _personRepository;

    ///<summary>
    ///在构造函数中,我们可以取得我们所需要的类/接口
    ///它们都会由依赖注入系统自动地送到这边来
    ///</summary>

    public TaskAppService(ITaskRepository taskRepository,
        IRepository<Person> personRepository) {
        _taskRepository = taskRepository;
        _personRepository = personRepository;
    }

    public GetTaskOutput GetTasks(GetTasksInput input) {
        //调用 Task 仓储指定的 GetAllWithPeople 方法

        var tasks =
            _taskRepository.GetAllWithPeople(input.AssignedPersonId, input.State);

        //使用 AutoMapper 来自动化地转型 List<Task>为 List<TaskDto>
        return new GetTaskOutput {
            Tasks = Mapper.Map<List<TaskDto>>>(tasks)
        }
    }

    public void UpdateTask(UpdateTaskInput input) {
        //我们可以使用 Logger,它被定义在 ApplicationService 的基类中

        Logger.Info("Updating a task for input: " + input);

        //通过仓储标准的 Get 方法,以 Id 检索单个 Task 实体
    }
}
```

```
var task = _taskRepository.Get(input.TaskId);

//将检索出来的 Task 实体,其改动的属性值更新到数据库
if(input.State.HasValue) {
    task.State = input.State.Value;
}

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

//我们甚至没有调用仓储的更新方法
//这是因为一个应用服务方法预设上即为一个"工作单元"的控制范围
//ABP 会在工作单位控制范围尽头自动地储存所有的改动(前题是之前没有抛出任何异常)
}

public void CreateTask(CreateTaskInput input) {
    //我们可以使用 Logger,这是因为它被定义在 ApplicationService 类中
    Logger.Info("Creating a task for input : " + input);

    //使用 input 的属性创建一个新的 Task 实体
    var task = new Task { Description = input.Description };

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

    //由仓储的 Insert 方法储存实体
    _taskRepository.Insert(task);
}
}
```

在 `UpdateTask` 方法中,我们通过仓储取得了一个 `Task` 实体,并且储存修改后的属性。`State` 或/且 `AssignedPersonId` 可能会被改动。注意,我并没有调用



`_taskRepository.Update` 或是任何方法来储存改动到数据库中。这是因为,一个应用服务方法在 ABP 中,预设上即为一个工作单元。就工作单元来说,它基本上是开启了一条数据库连接,并且在方法开始的时候就启动了事务,而后在方法尽头自动地储存所有改动(提交这个事务)到数据库中。若在方法执行的过程中抛出异常,则它会自动地回滚这个事务。若这个工作单元的方法调用了另一个工作单元的方法,则它们会共享同一个事务。第一个被调用的工作单元方法会自动地处理连接的创建/释放以及事务管理。

想要了解更多 ABP 在工作单元方面的详情,可见 [document](#)。

## 7.8.2 DTO 验证

验证是至关重要的,但是它对于应用程序开发来说是有一点令人感到乏味的。ABP 提供了基础设施让验证变得更简单且好用。验证一个使用者的输入是应用层该做的事。一个应用服务方法应该要验证输入且在当输入验证不通过的时候抛出异常。ASP.Net MVC 和 WebAPI 都有内建的验证系统,它可以使用数据注记(Data Annotation)来达成验证功能(像是 `Required` 这个数据注记)。但是一个应用服务是一个无平台依赖的类,并非是继承自 `Controller`。幸运的是,ABP 提供了相似的机制给正规的应用服务方法(使用 `Castle` 动态代理器(Dynamic Proxies)和拦截器(Interception)来达成)。你的方法输入参数应该要先标记为 `IValidate`。`IInputDto` 已经有继承自 `IValidate`,因此你可以标记你的 DTO 类为 `IInputDto` 启用验证机制:

```
public class CreateTaskInput : IInputDto {  
    public int? AssignedPersonId { get; set; }  
  
    [Required]  
    public string Description { get; set; }  
}
```

在输入型 DTO 中,仅有 `Description` 属性是设定为必须的(required)。ABP 会自动地在调用应用服务方法之前进行检核,并且当它是 `null` 或是空字符串时抛出异常。所有的验证属性都在 `System.ComponentModel.DataAnnotations` 命名空间底下。如果这些标准的属性不足以让你使用,你可以实现 `ICustomValidate`:

```
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 !"));
        }
    }
}
```

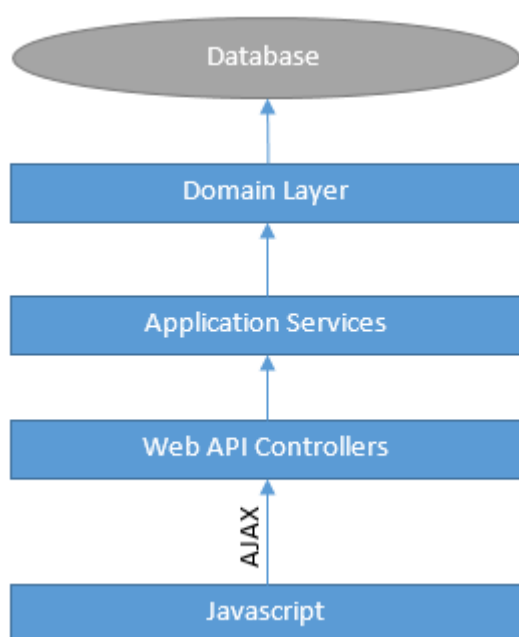
附带一题: **ABP** 会检核出应用服务的输入参数为 **null** 值,所以你不需要额外写语法来检核是否为 **null** 值。

我建议针对每一个应用服务方法将输入和输出都分别集成不同的类,即便这个应用服务方法的输入参数原本只有一个输入参数也包装成一个类。如此一来便能够在添加新参数到某个应用服务方法而不会让现存的客户端发生问题。

### 7.8.3 动态 Web API 控制器

表现层消费应用服务。在单页应用程序中,所有的数据都会使用 **Ajax** 在 **JavaScript** 与服务器端之间传送/接收。**ABP** 能够大幅简化 **JavaScript** 调用应用服务的方法。它是怎么做到的?让我来解释一下...

一个应用服务无法直接由 **JavaScript** 调用。我们或许会使用 **ASP.Net WebAPI** 来曝露服务给客户端(这里有许多其它的框架可以做到相同的事,例如 **Web Service**,**WCF**,**SignalR** 等等)。因此,其流程大致如下图:



JavaScript 通过 Ajax 调用 Web API 控制器的方法,Web API 控制器的方法接着会调用对应的应用服务方法,取得结果后回传给客户端。这非常自动化。ABP 在这部份的处理都是自动化的,并且可以动态地创建一个 Web API 控制器给应用服务。下面是所有用来创建 Web API 控制器给我的应用服务(Task 和 Person 应用服务)的程序代码如下:

```
DynamicApiControllerBuilder
    .ForAll<IApplicationService>(Assembly.GetAssembly)(typeof
(SimpleTaskSystemApplicationModule)), "taskssystem")
    .Build();
```

因此,所有 Task 和 Person 应用服务的方法都会以 ASP.Net Web API 曝露给客户端(ABP 的流式动态控制器创建 API 之法支持 Web API 的方法隐藏功能或是能够选择特定应用服务,你可以自行试用)。在表现层章节中,我们将会看到如何使用 ABP 的动态 JavaScript 代理器来调用 Web API 的方法。

## 7.9 表现层

"负责将信息显示给用户,并且解析用户的命令"(Eric Evans)。在领域驱动设计(DDD)中,表现层是显著的一层,因为我们可以看到并且点击它。

### 7.9.1 单页应用

Wikipedia 是这么说明 SPA:一个单页应用程序(SPA),也可说它是单页接口(SPI),它可以是一种网页应用程序或是一个网站,只要它能够提供如在桌面应用程序般的流畅体验。

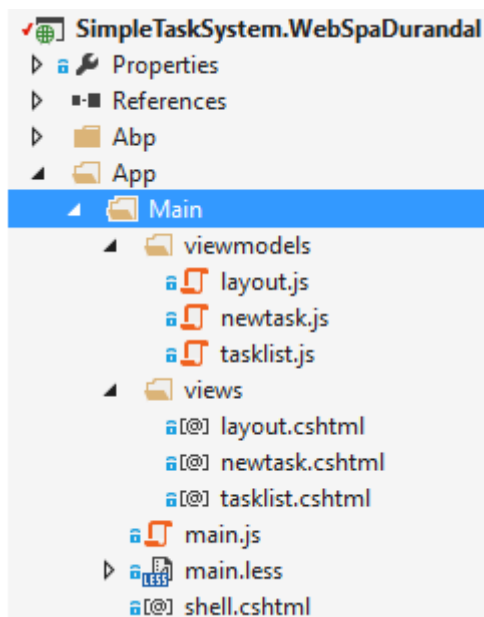
在 SPA 中,可以是所有的必需型程序代码(Html,JavaScript,CSS)都在一页中全部加载,或是让某些资源在用户有需要的时候动态地加载到这一页中。页面不会在任何一个执行过程中突然地重载(reload),也不会跳转到另一页面,虽然现代网页技术(诸如那些在 HTML5 中的功能)已经可以提供应用程序中所谓逻辑概念的页面能够被感知且被浏览。与单页应用程序互动,其实在背后通常都会需要动态地与网站服务器沟通。

有许多的框架和类库都提供了建构 SPA 的基础设施。ABP 可以和任何一个 SPA 框架协作,而且 ABP 特别为 DurandalJs 和 AngularJs 提供基础设施,让协作变得更容易。

Durandal 是一个 SPA 框架,并且我认为它是一个非常成功的开源项目。它是立足于在一些已成功并且常被使用的项目上: jQuery(用它来操作 DOM 和 Ajax 处理), Knockout.js(用来处理 MVVM,连接 JavaScript 的模型到 HTML),以及 require.js(用于管理 JavaScript 程序代码之间的依赖并且能够动态地从服务器端加载 JavaScript 文件)。欲取得更多信息与更丰富的文件,请参考 Durandal 的官方网站。

### 7.9.2 视图和视图模型

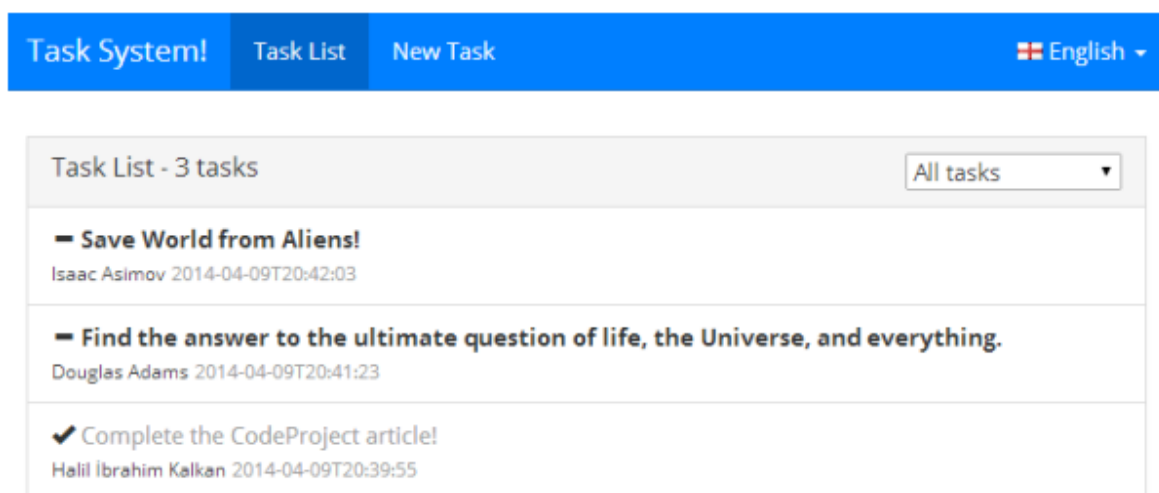
在 Durandal 中,页面是由视图和视图模型所组成。在 ABP 的启动样版中,有三个视图: Layout,Home 和 About。Layout 提供菜单和页面容器。home 和 about 则是将要被动态加载到 Layout 页面容器中的视图。我修改了视图和模型,为的是要创建我的简易任务系统应用程序。在修改视图后,其所有视图与视图模型的文件结构如下:



我有一个 **Layout** 以及两个视图: 任务列表(**Task List**)和新任务(**New Task**)。让我们开始来研究它们吧。

### (1) 任务列表

在任务列表中,列表中会有所有的任务以及它们的任务描述,受指派人员和创建日期。这里有一个 **combobox** 来过滤所有/启用/完成任务。完成的任务会显示为灰色(伴随 **OK** 图标),启用任务则是粗体(伴随一个负号的图示)。目前只有一个已完成的任务,即为"Complete the Codeproject article!",这个任务是指派给我的。因此,你才可以看到篇文章。



让我们先从视图模型作为开始。一个视图模型被用于与服务器端沟通,用来完成用户的命令(倾听任务,改变 **combobox**,点击图标来改变任务的状态)并且提供了一个

用于显示在视图上的模型。

```
define(['service!tasksystem/task'],
    function(taskService) {
        return function() {
            var that = this; //此为 this 的别名
            that.tasks = ko.mapping.fromJS([]); //任务列表
            that.localize =
abp.localization.getSource('SimpleTaskSystem');

            that.selectedTaskState = ko.observable(0); //设定 combobox 中任
务状态选项的默认选项值

            that.activate = function() {
                that.refreshTasks();
            }

            that.refreshTasks = function() {
                abp.ui.setbus( //让整个页面处于忙碌状态,直到 getTasks 方法完成
                    null,
                    taskService.getTasks({
                        state : that.selectedTaskState() > 0 ?
that.selectedTaskState() : null
                    }).done(function(data) {
                        ko.mapping.fromJS(data.tasks, that.tasks);
                    })
                );
            };
        };
    });

    that.changeTaskState = function(task) {
        var newState;
        if(task.state() == 1) {
            newState = 2;
        } else {
            newState = 1;
        }
    }
}
```

```
taskService.updateTask({
    taskId: task.id(),
    state: newState
}).done(function() {
    task.state(newState);
    abp.notify.info(that.localize('TaskUpdateMessage'));
});

};

};

});
```

在第一行中调用了 **require.js** 的 **define** 函数,通过这个函数来注册模块并且声明其依赖对象。依赖对象通常是另一个模块(或许可能只是个视图模型)。**'service!tasksystem/task'**即为一个特殊的 **ABP** 专有的语法,它参照到动态地为 **Task** 应用服务所产生的那个 **Web API** 控制器(还记得我在动态 **Web API** 控制器该章节中所定义的那个 **Task** 服务吗?)。**getTasks** 函数即是由 **ABP** 动态产出的。

**define** 函数中的第二个参数即为模块自己本身。它应为一个函数,它定义了这个模块。它的参数会通过 **Durandal** 提供给你所列的依赖对象来动态地注入参数。

**that.tasks** 是一个 **Knockout** 的受观察数组(**Observable Array**)。它通过 **Knockout.mapping** 函数来创建。**that.localize** 是一个函数,它被用于处理多语言工作。它是 **ABP** 的一个功能特性,能够在 **JavaScript** 中动态地转译文字成指定语言的文字(会在多语言章节中有更详细的说明)。**that.selectedTask** 是一个受观察属性(**Observable**),它被连接到 **combobox** 显示所有/启用/完成 各种任务的状态。

激活(**activate**)是一 **Durandal** 中一个特别的函数。这个函数会在视图激活后被 **Durandal** 自动地调用。因此,我们可以写一些程序代码在使用者加载这个视图的时候执行特定的动作。

在 **refreshTask** 方法中,我调用了 **Task** 应用服务的 **getTask** 方法,从服务器端加载所有任务。要从 **JavaScript** 调用到应用服务,可以通过 **ABP** 的动态 **Web API** 控制器和动态的 **JavaScript** 客户端代理(**client proxies**)来做到。**getTasks** 函数与

`TaskAppService.GetTasks` 有着同样的参数。该函数会回传一个 jQuery 的 Promise 对象,因此,你可以写 `done` 处理器(handler)来取得 `Task` 应用服务的 `GetTasks` 的回传值。`taskService.getTasks` 也处理错误,并且在有必要的时候显示错误讯息给用户看。如果 `done` 处理器被调用了,你就可以确定没发生任何错误。在 `done` 处理器中,我添加了从服务器端取得的任务信息到 `that.task` 这个受观察数组(`Observable Array`)中。

`changeTaskState` 也是极为类似的。它被用于将任务从启用状态变更为完成状态,反之亦然。在 `done` 处理器中,你可以看到本地化语言和讯息通知 API 的使用方式。

一如视图模型是一个 JavaScript 文件,视图也是一个 HTML 文件。在 HTML 文件中,你可以连接视图模型到 HTML 元素上。ABP 更进一步地将这个能力做到在 Razor 上: 动态 cshtml 文件而不是静态 HTML 文件。因此,你可以写 C# 程序代码来创建视图。见如下任务列表视图:

```
<div class="panel panel-default">
  <div class="panel-heading" style="position: relative;">
    <div class="row">
      <h3 class="panel-title col-xs-">
        @L("TaskList") - <span data-bind="text:
abp.utils.formatString(localize('Xtasks'), tasks().length)"></span>
      </h3>
      <div class="col-xs-6 text-right">
        <select data-bind="value: selectedTaskState, event:
{ change: refreshTasks}">
          <option value="0">@L("AllTasks")</option>
          <option value="1">@L("ActiveTasks")</option>
          <option value="2">@L("CompletedTasks")</option>
        </select>
      </div>
    </div>
  </div>
  <ul class="list-group" data-bind="foreach: tasks">
    <div class="list-group-item">
      <span class="task-state-icon glyphicon" data-bind="click:
$parent.changeTaskState,
```



```
        css: {'glyphicon-minus' : state() == 1,
'glyphicon-ok' : state() == 2}"></span>
        <span data-bind="html: description(), css: { 'task-
description-active' : state() == 1,
'task-
description-completed' : state() == 2 }"></span>
        <br />
        <span data-bind="visible: assignedPersonId()">
            <span class="task-assignedto" data-bind="text:
assignedPersonName"></span>
        </span>
        <span class="task-creationtime" data-bind="text:
moment(creationTime()).fromNow()"></span>
    </div>
</ul>
</div>
```


这个视图是设计成能够与 **Twitter Bootstrap** 协作。用到的 **CSS** 类皆为 **Bootstrap** 的类。但是这并非重点。有两个真正的重点:

首先: 我们可以使用 `@L("TaskList")` 来取得语言字符串。`L` 方法被定义在 `AbpWebViewPage` 类中(见 `SimpleTaskSystemWebViewPageBase` 类,它继承自 `AbpWebViewPage`)。你可以在视图使用 `L` 作为 `LocalizationHelper.GetString(...)` 的快捷方式方法(在多语言章节中有更详细的说明)。就因为这是 **Razor** 视图,我们可以直接使用 **C#** 程序代码在视图中。因此,我们可以创建动态 **HTML** 于服务器端。要记得的是,这个视图只会被加载一次,因为它是 **SPA**。

其次: 我们可以通过 **Knockout** 的 `data-bind` 属性来操作 **JavaScript** 视图模型中的属性(例如: `data-bind="foreach: tasks"`)和其方法(例如 `abp.utils.formatString`)。因此,我们可以创建动态 **HTML** 于我们的客户端。一如 `click: $parent.changeTaskState`, 这是将图示的点击(**click**)事件与 **JavaScript** 视图模型中的 `changeTaskState` 函数做连接的动作。同样地,我们连接 **combobox** 的更动(**change**)事件与 `refreshTasks` 函数。想要了解更多详情,见 **Knockout** 官方网站。

## (2) 新任务


"New Task"视图是相对简单的一个视图。页面上会有任务描述以及一个可选的人员选单:

SimpleTaskSystem! 

**Task description**

Build the Utopian framework!

**Assign to**

Thomas More 

Create the task

这个页面的视图模型如下所示:

```
define(['service!tasksystem/person', 'service!taskssytem/task',
'plugins/history'],
function(personService, taskService, history) {
    var localize = abp.localization.getSource('SimpleTaskSystem');
    return function() {
        var that = this;
        var _$view = null;
        var _$form = null;

        that.people = ko.mapping.fromJS([]);
        that.task = {
            description : ko.observable(''),
            assignedPersonId: ko.observable(0)
        };

        that.canActivate = function() {
            return personService.getAllPeople().done(function(data) {
```

```
        ko.mapping.fromJS(data.people, that.people);
    });
};

that.attached = function(view, parent) {
    _$view = $(view);
    _$form = _$view.find('form');
    _$form.validate();
};

that.saveTask = function() {
    if(!_$form.valid()) {
        return;
    }
    abp.ui.setBusy(_$view,
        taskService.createTask(ko.mapping.toJS(that.task))
            .done(function() {

abp.notify.info(abp.utils.formatString(localize("TaskCreatedMessage"),
that.task.description()));

        history.navigate('');
    })
    );
};

};

});
```

**canActivate** 在 **Durandal** 中是一个特别的函数。在此函数中,你可以回传 **true/false** 来控制允许/拒绝进入到这个页面。**Durandal** 也接受一个 **Promise** 对象参数。在这个例子中,它等待 **Promise** 的结果来决定是否要启用视图。你的 **Promise** 对象应该只回传 **true/false**。**ABP** 覆写掉这个行为让 **Promise** 对象可以不强迫一定要回传 **true/false**。因此,你可以直接回传由 **ABP** 动态 **JavaScript** 代理器所回传的 **Promise** 对象。(如上述 **canActivate** 方法中所示)。

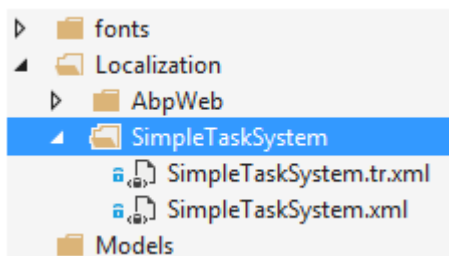
**attached** 也是另一个 **Durandal** 中特别的函数,它会在当你的视图附加到

DOM(Document Object Model)的时候被调用,并且安全地操作它。

`saveTask` 用于把任务储存到服务器。它会先验证窗体之后再调用 `taskService.createTask` 函数(还记得这个函数是自动地且动态地由 ABP 所创建吗?并且它会回传 `Promise` 对象)。在此,你看到了两个 ABP 的 API 调用。`abp.notify.info` 会在将任务储存到服务器后显示通知讯息。`abp.ui.setbusy` 用于设定 DOM 处于忙碌状态(显示加载的进度并且阻塞 UI 元素,所有的检示都采用这个模式)。它会在 `Promise` 对象回传后(有可能是发生错误或是已成功)接受这个 `Promise` 对象并且取消 UI 的忙碌状态。ABP 使用一些 jQuery 插件来处理这些动作。

### 7.9.3 本地化

ABP 提供了一个强健且具弹性的多语言系统。你可以储存你的语言文本于资源文件(resource file),XML 文件,甚至是定制的数据源(custom source)。在本章节中,我将会示范如何使用 XML 文件。先在简易任务系统项目的 `Localization` 文件夹中引入 XML 文件:



在此,即为 `SimpleTaskSystem.xml` 的完整内容:

```
<?xml version="1.0" encoding="utf-8" >
<localizationDictionary culture="en">
  <text>
    <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="AllTasks" value="All tasks" />
    <text name="ActiveTasks" value="Active tasks" />
    <text name="CompletedTasks" value="Completed tasks" />
  </text>
</localizationDictionary>
```

```
<text name="TaskDescription" value="Task description" />
<text name="EnterDescriptionHere" value="Task description" />
<text name="AssignTo" value="Assign to" />
<text name="SelectPerson" value="Select person" />
<text name="CreatTheTask" value="Create the task" />
<text name="TaskUpdatedMessage" value="Task has been
successfully updated." />
<text name="TaskCreatedMessage" value="Task {0} has been created
successfull." />
</texts>
</localizationDictionary>
```

这是一个简单的 XML 文件,它采用键-值对(name-value pair)的格式来描述语言文字的对映。**culture** 属性定义了这个文件的国码。在项目中也有一个土耳其语言的 XML 文件。语言文件要能够在 C#/JavaScript 中被使用,需要先注册到 ABP 中:

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

语言来源必须使用独一无二的名称(在 **SimpleTaskSystem** 中)。这样,不同的来源(储存成不同的格式或数据源)才能在应用程序中共存。**XmlLocalizationSource** 也需要一个文件夹地址才能读取语言文件。(在本示例是 **/Localization/SimpleTaskSystem**)。

接下来我们就可以在我们有需要的时候取得语言文字。在 C# 中,我们有两种取得语言文字的方法:

```
//直接调用
var s1 = LocalizationHelper.GetString("SimpleTaskSystem", "NewTask");

//延迟取值
var source = LocalizationHelper.GetSource("SimpleTaskSystem");
```

```
var s2 = source.GetString("NewTask");
```

这些方法都可以通过当前语言环境取得语言文字。

### 7.9.4 JavaScript API

有些是每个应用程序在客户端 **JavaScript** 中都会用到的功能。举例来说: 为了要显示成功讯息,要阻塞 **UI** 中某些元素,要显示讯息盒...等等。已经有需多类库(**jQuery** 插件)可以做到这些。但是它们都有着不同的 **API**。**ABP** 定义了一些通用的 **API** 来解决这个问题。因此,倘若你想要改变通讯插件,你只需要实现一个简单的 **API**。同样地 **jQuery** 插件也可以使用 **ABP** 的 **API** 来实现。并非是直接调用插件的通知 **API**,你可以调用 **ABP** 的通知 **API**。在此,我将要解析这些 **API**。

#### (1) Logging API

当你想要在客户端写一些简单的日志,如你所知,你可以使用 `console.log(...)` API。但是这个 **API** 并不是所有浏览器都支持,并且你的 **JavaScript** 有可能因此而发生问题。因此,你应该先检查这个 **API** 能否在这个浏览器上使用。同样地,你或许会想要写到不同的地方。**ABP** 定义了一个安全的日志函数:

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

同样地,你可以在 `abp.log.level` 设定日志的等级(例:`abp.log.levels.INFO`,这样就不会写到 `debug` 日志中)。这些函数默认会写入日志到 `console` 中。但是你若想要覆写这个行为也是很容易就能做到。

#### (2) Notification API

我们喜欢在某些事发生时,显示一些梦幻且能自动消失的讯息,像是当一个东西被储存后或是有问题发生时。**ABP** 定义了一些 **API**:

```
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');
```

讯息通知 API 默认是以 **toastr** 这个类库来实现的。你可以使用你喜欢的讯息通知类库来实现。

### (3) MessageBox API

**Message API** 用于显示讯息信息给用户。使用点击 **OK** 后关闭讯息窗口/对话框。示例如下:

```
abp.message.info('some info message', 'some optional title');
abp.message.warn('some warning message', 'some optional title');
abp.message.error('some error message', 'some optional title');
```

目前尚未实现,可以将它实现成显示对话框或是讯息盒。

### (4) UI Block API

这个 **API** 是用来阻塞整个页面或是只阻塞某个页面中的元素。因此,使用者无法点击它。**ABP** 的 **API** 如下:

```
abp.ui.block() //阻塞整个页面
abp.ui.block($("#MyDivElement")); //你可以使用任一种 jQuery 选择器...
abp.ui.block('#MyDivElement'); //...或者直接用选择器
abp.ui.unblock(); //解除整个页面
abp.ui.unblock('#MyDivElement'); //解除特定的元素
```

### (5) UI Busy API

有的时候你或许会想要令页面/某些元素处于忙碌状态。举例来说,你或许要阻塞窗体并且显示忙碌的相关讯息于窗体正提交到服务器的时候。**ABP** 提供的 **API** 如下:

```
abp.ui.setBusy('#MyRegisterForm');
abp.ui.clearbusy('#MyRegisterForm');
```

**setBusy** 可以将 **Promise** 对象作为其第二个参数来自动地调用 **clearBusy** 于 **Promise** 已完成的时候。见示例项目中 **newTask** 这个视图模型的使用方式。

### (6) 其它

其它好用的 **API** 在未来会陆续加入到 **ABP** 中,来标准化通用任务。也有些针对通用功能的工具型函数(一如 **abp.utils.formatString**,它的使用方式类似 **C#** 中的 **string.format**)。

---

△ 注意:

如果你想要实现这些 API, 在不同的文件中实现它们, 并且引入这些 js 文件到你的页面中, 但是要注意引入的位置都需要在 `abp.js` 之后。

## 7.10 更多

### 7.10.1 模块系统

ABP 被设计成模块化。它提供了基础设施来创建多个通用模块, 这些通用模块可用在各种不同的应用程序。一个模块可以依赖其它模块。一个应用程序是由多个模块结合而成。一个模块即为一个组件 (**Assembly**), 在这个组件中会有一个继承自 **AbpModule** 的类。在本篇文章中的示例应用程序中已经解释了, 所有层都被定义成分离的模块。举例来说, 应用层定义了一个模块如下:

```
///<summary>
///这个项目中的应用层模块
///</summary>

[DependsOn(typeof(SimpleTaskSystemcoreModule))]

public class SimpleTaskSystemApplicationModule : AbpModule {

    public override void Initialize() {

        //这段程序代码让组件得已通过注册这个类到依赖注入系统中, 此为约定的做法

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

    //我们必须宣告完映射器后才能够使用 AutoMapper

    DtoMappings.Map();

}

}
```

ABP 在应用系统启动阶段分别调用模块中的 **PreInitialize**, **Initialize** 和 **PostInitialize** 方法。假如模块 A 依赖于模块 B, 模块 B 会在模块 A 之前就先被初始化。实际的方法调用顺序: **PreInitialize-B**, **PreInitialize-A**, **Initialize-B**, **Initialize-A**, **PostInitialize-B** 和 **PostInitialize-A**。对于所有依赖关系来说都是这个顺序。

**Initialize** 这个方法是依赖注入配置设定的地方。在此, 你看到这个模块注册所有



类到它的组件中,这个行为是依循约定而为(见下一节)。接着它会使用 **AutoMapper**(在这个示例中指定使用这个类库)映射类。这个模块也定义了依赖(应用层只依赖于应用程序中的领域(核心)层)。

### 7.10.2 依赖注入和约定

当你的应用程序有遵循最佳实践和一些约定的时候,ABP 就可以让你以透明化的方式使用依赖注入系统。它会自动地注册所有的仓储,领域服务,应用服务,MVC 控制器和 Web API 控制器。举例来说,你有一个 **IPersonAppService** 接口,并且 **PersonAppService** 类实现这个接口:

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

ABP 会自动地注册这个类,这是因为它实现了 **IApplicationService** 接口(它只是一个空接口)。它被注册为暂时性对象(当有需要的时候才创建)。当你注册(使用构造函数注入) **IPersonAppService** 接口到类中,**PersonAppService** 对象会被创建且自动地送到构造函数中。见详细文件于依赖注入,且它在 ABP 中被实现。

## 7.11 结论

在本篇文章中,我介绍了一个新的应用程序框架: **ASP.Net Boilerplate**。它是一个开发现代网站应用程序的起点,并且使用最佳实践和最受欢迎的工具。因为它非常的新,它或许会有些错误。但是它并没有限制住你。你可以以它作为起始点并且建构出你的应用程序。它是新的且正在开发和扩充中。我已经开始将它应用到我公司的产品上了。

你可以写错误报告(bug reports),议题(issue)和功能请求到 **github**。因为它是个开源项目,你可以 **fork** 它并且送拉式请求(pull request)给 ABP 源码贡献你的成果。我

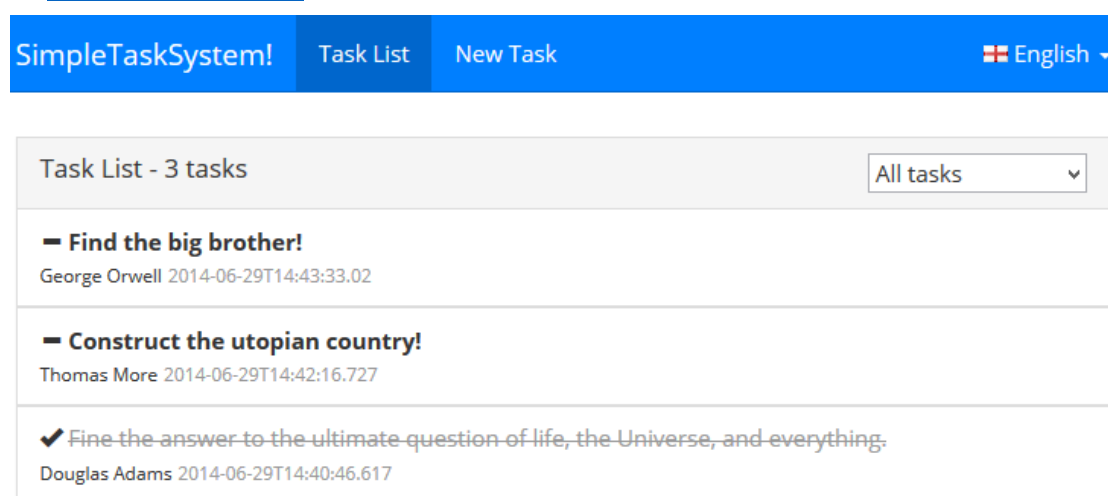
希望它能够成为我们所有.Net 开发人员的起点。因此,它将会是开发时的好伙伴。

(翻译: 台湾-小张, 校订: 成都-乐忧)

## 8 ABP 实例二：单页面网站应用程序

使用 AngularJs、ASP.NET MVC、Web API、EntityFramework 和 ASP.NET Boilerplate 构建多层架构（NLayered）、本地化(localized)、结构良好的单页面网站应用程序（Single-Page Web Application）。

[点击下载示例程序](#)



### 8.1 简介

在本文中，我将会向你展示如果基于以下技术开发一个单页面网站程序(Single-Page Web Application (SPA))。

- Web 框架：ASP.NET MVC 和 ASP.NET Web API
- SPA 框架：AngularJs
- ORM(Object-Relational Mapping)框架：EntityFramework
- 依赖注入(Dependency Injection)框架：Castle Windsor
- Html/CSS 框架：Twitter Bootstrap
- 日志框架：Log4Net, 对象转换(object-to-object mapping)框架：AutoMapper
- Abp 作为启动模板和应用框架

Abp 是一个开源应用程序框架，Abp 组织以上这些框架和类库使得你更简单的开始开发应用程序。它提供了基础设施让我们基于最佳实践来开发应用程序。它支持

依赖注入(Dependency Injection)、领域驱动设计(Domain Driven Design)和多层架构(Layered Architecture)。示例程序还实现了验证(validation)、异常处理(exception handling)、本地化(localization)和响应式设计(responsive design)。

## 8.2 基于 Abp 创建应用程序

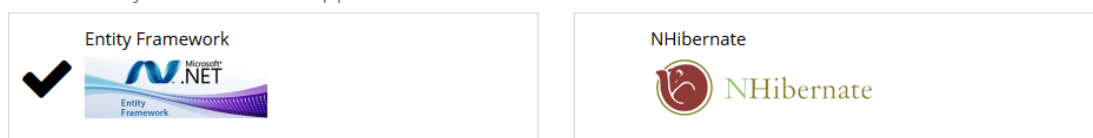
基于 Abp 提供的程序模板(templates)开发应用能节省我们大量的时间, Abp 程序模板合并(combines)和配置(configures)了最好的工具去构建企业级应用程序。

前往 [aspnetboilerplate.com/Templates](http://aspnetboilerplate.com/Templates) 去基于模板构建应用程序吧。

### 1. Select Architecture



### 2. Select Object-Relational Mapper



### 3. Choice your project's name

Enter your solution's name (without spaces)

如上图所示,我选择了 AngularJs 和 EntityFramework 去构建单页面 SPA(Single Page Application)应用程序。而且,将项目命名为 SimpleTaskSystem。创建和下载我们的解决方案。

此解决方案包括 5 个项目。Core 项目作为领域(业务)层(domain (business) layer), Application 项目作为应用程序层(application layer), WebApi 项目实现了 Web Api 控制器。Web 项目作为表现层(presentation layer), 最后 EntityFramework 项目作为 EntityFramework 的实现。

### △ 注意:

如果你下载了示例解决方案, 你可以看到解决方案包含 7 个项目。那是由于多了两个支持 NHibernate 和 Durandal 的项目。如果你对 NHibernate 和 Durandal 不感兴趣, 可以忽略他们。

## 8.3 创建实体

正在创建的此示例程序的功能是创建 **tasks** 并将任务分配给 **people**。因此，我需要 **Task** 和 **Person** 实体。

**Task** 实体简单的定义了 **Description**、**CreationTime** 和 **State**。当然它还有一个可选的引用指向 **Person(AssignedPerson)**：

```
public class Task : Entity<long>
{
    [ForeignKey("AssignedPersonId")]
    public virtual Person AssignedPerson { get; set; }

    public virtual int? AssignedPersonId { get; set; }

    public virtual string Description { get; set; }

    public virtual DateTime CreationTime { get; set; }

    public virtual TaskState State { get; set; }

    public Task()
    {
        CreationTime = DateTime.Now;
        State = TaskState.Active;
    }
}
```

**Person** 实体更简单，只是定义了 **Name** 成员：

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

Abp 提供了包括了 **Id** 属性的 **Entity** 类。我从此类派生实体类。由于 **Task** 派生于 **Entity**，因此 **Task** 的主键属性 **Id** 是 **long** 类型的。**Person** 的主键 **Id** 是 **int** 类型，因

为 `int` 是默认的主键类型。因此我没有特别指定。

我将实体定义在了 `Core` 项目中，是由于实体是领域/业务层(domain/business layer)的一部分。

## 8.4 创建 DbContext

如你所知，`EntityFramework` 基于 `DbContext` 进行工作。我们需要先定义 `DbContext`，`Abp` 模板已经为我们创建了它。我只是添加了 `IDbSet` 属性 `Task` 和 `Person`。下边是我的 `DbContext` 类：

```
public class SimpleTaskSystemDbContext : AbpDbContext
{
    public virtual IDbSet<Task> Tasks { get; set; }

    public virtual IDbSet<Person> People { get; set; }

    public SimpleTaskSystemDbContext()
        : base("Default")
    {

    }

    public SimpleTaskSystemDbContext(string nameOrConnectionString)
        : base(nameOrConnectionString)
    {

    }
}
```

它使用了默认的连接字符串，它定义在了 `web.config` 中如下所示：

```
<add name="MainDb" connectionString="Server=localhost;
Database=SimpleTaskSystemDb; Trusted_Connection=True;"
providerName="System.Data.SqlClient" />
```

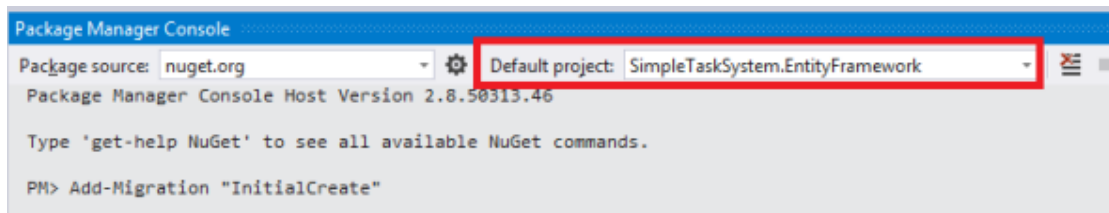
## 8.5 创建 Database Migrations

我们将使用 EntityFramework 的 Code First Migration 来创建和保持数据库结构。Abp 模板默认启用了 Migration，也添加了一个配置(Configuration)类如下代码所示：

```
//内部访问权限，密封不可派生配置类
DbMigrationsConfiguration<SimpleTaskSystem.EntityFramework.SimpleTaskSystemDbContext>
{
    public Configuration()
    {
        AutomaticMigrationsEnabled = false;

        protected override void
Seed(SimpleTaskSystem.EntityFramework.SimpleTaskSystemDbContext
context)
    {
        context.People.AddOrUpdate(
            p => p.Name,
            new Person {Name = "Isaac Asimov"},
            new Person {Name = "Thomas More"},
            new Person {Name = "George Orwell"},
            new Person {Name = "Douglas Adams"}
        );
    }
}
```

在 **Seed** 方法中，我添加了 4 个人作为初始数据。现在，我们去创建 initial migration。打开 **Package Manager Console**，输入以下命令：



Add-Migration "InitialCreate"命令用于创建一个名为 InitialCreate 的类。如下所示：

```
public partial class InitialCreate : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.StsPeople",
            c => new
            {
                Id = c.Int(nullable: false, identity: true),
                Name = c.String(),
            })
            .PrimaryKey(t => t.Id);

        CreateTable(
            "dbo.StsTasks",
            c => new
            {
                Id = c.Long(nullable: false, identity: true),
                AssignedPersonId = c.Int(),
                Description = c.String(),
                CreationTime = c.DateTime(nullable: false),
                State = c.Byte(nullable: false),
            })
            .PrimaryKey(t => t.Id)
            .ForeignKey("dbo.StsPeople", t => t.AssignedPersonId)
            .Index(t => t.AssignedPersonId);
    }
}
```



```
public override void Down()
{
    DropForeignKey("dbo.StsTasks", "AssignedPersonId",
"dbo.StsPeople");

    DropIndex("dbo.StsTasks", new[] { "AssignedPersonId" });

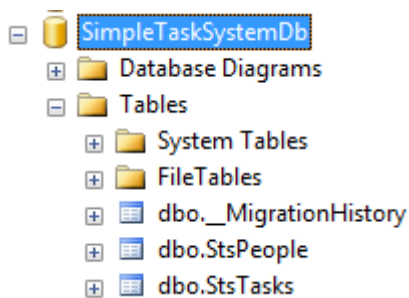
    DropTable("dbo.StsTasks");

    DropTable("dbo.StsPeople");
}
}
```

我们创建了创建数据库所需的类，但是并没有创建数据库。使用以下命令创建数据库：

```
PM> Update-Database
```

此命令运行 **Migration**，创建数据库，并初始化数据。



当我们修改了实体类，我们只需要简单的使用 **Add-Migration** 命令创建 **Migration** 类，使用 **Update-Database** 命令更新数据库即可。请查看 [entity framework's](#) 文档获取更多信息。

## 8.6 定义仓储

在领域驱动设计中，仓储通常对应相应的数据库。Abp 使用 **IRepository** 接口为每个实体创建自动仓储。**IRepository** 定义了常用的方法，如 **Select**、**insert**、**update**、**deleted** 等等。

我们可以根据需要扩展 **Repository**。我将扩展 **Task** 仓储。为了将接口从实现分开，因此我先为仓储定义接口。以下是 **Task** 仓储的接口：

```
public interface ITaskRepository : IRepository<Task, long>
```

```
{  
    List<Task> GetAllWithPeople(int? assignedPersonId, TaskState?  
state);  
}
```

它扩展了 Abp 的 `IReposity` 接口。因此, `ITaskRepository` 继承了所有的默认方法。它还可以添加自己的方法就像我定义的 `GetAllWithPeople(...)`。

没有必要为 **Person** 创建仓储, 默认方法已经对我来说已经足够了。Abp 提供一种注入仓储对象而不需要创建仓储类的方法。我们将在之后的 `TaskAppService` 类的 `Build application services` 部分看到。

我将仓储(repository)接口定义在了 **Core** 项目中, 是由于实体是领域/业务层(domain/business layer)的一部分。

## 8.7 实现仓储

我应该实现之前定义的 `ITaskRepository` 接口。我将在 `EntityFramework` 项目中实现仓储。因此, `EntityFramework` 完全依赖领域层(domain layer)。

当我们创建了项目模板, Abp 在应用程序中创建了一个通用的仓储基础类: `SimpleTaskSystemRepositoryBase`。拥有这样一个基础类是比较好的实践, 之后可以为我们的仓储添加一些通用方法。你可以在代码中查看定义。这里我只是从基础类派生了 `TaskRepository`:

```
public class TaskRepository : SimpleTaskSystemRepositoryBase<Task,  
long>, ITaskRepository  
{  
    public List<Task> GetAllWithPeople(int? assignedPersonId, TaskState?  
state)  
    {  
        //在仓储方法中, 我不需要处理数据库的连接、数据上下文(DbContexts)和事务  
(transactions)  
  
        var query = GetAll(); //GetAll() 返回 IQueryable<T>, 所以我们可以查询  
这个对象。  
  
        //var query = Context.Tasks.AsQueryable(); //可替换方法, 我们可以直接
```

使用 EF 的上下文对象 DbContext。

```
//var query = Table.AsQueryable(); //另一个可替换方法： 我们可以直接使用  
用 'Table' 属性，替换 'Context.Tasks'，他们是等效的。
```

```
//在这里添加其他逻辑
```

```
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) //结果包含 AssignedPerson  
属性对象  
    .ToList();  
}
```

TaskRepository 派生于 SimpleTaskSystemRepositoryBase，而且实现了 ITaskRepository。

GetAllWithPeople 是我们指定的方法用于获取包含 AssignedPerson(pre-fetched)的 tasks，也可以加入其他的过滤条件。我们可以自由的在 Repository 中使用 Context(EF's DbContext)和数据库。Abp 为我们管理数据库连接(connection)，事务(transaction)，创建和释放 DbContext(请查看[文档](#)获取更多信息)。

## 8.8 构建应用程序服务

应用程序服务是领域层通过门面模式独立的一个表示层。这里我在项目的 `Application` 程序集定义了应用程序服务。首先，为 `task application service` 定义接口：

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

`ITaskAppService` 继承 `IApplicationService` 接口，这样 ASP.NET Boilerplate 自动的会为这个类提供一些特性（如：依赖注入和验证）。现在，让我们来实现 `ITaskAppService` 接口：

```
public class TaskAppService : ApplicationService, ITaskAppService
{
    //These members set in constructor using constructor injection.

    private readonly ITaskRepository _taskRepository;
    private readonly IRepository<Person> _personRepository;

    /// <summary>
    ///In constructor, we can get needed classes/interfaces.
    ///They are sent here by dependency injection system automatically.
    /// </summary>
    public TaskAppService(ITaskRepository taskRepository,
        IRepository<Person> personRepository)
    {
        _taskRepository = taskRepository;
        _personRepository = personRepository;
    }

    public GetTasksOutput GetTasks(GetTasksInput input)
```

```
{

    //Called specific GetAllWithPeople method of task repository.
    var tasks =
        _taskRepository.GetAllWithPeople(input.AssignedPersonId, input.State);

    //Used AutoMapper to automatically convert List<Task> to
    List<TaskDto>.

    return new GetTasksOutput
    {
        Tasks = Mapper.Map<List<TaskDto>>(tasks)
    };
}

public void UpdateTask(UpdateTaskInput input)
{
    //We can use Logger, it's defined in ApplicationService base
    class.
    Logger.Info("Updating a task for input: " + input);

    //Retrieving a task entity with given id using standard Get
    method of repositories.
    var task = _taskRepository.Get(input.TaskId);

    //Updating changed properties of the retrieved task entity.

    if (input.State.HasValue)
    {
        task.State = input.State.Value;
    }

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

```
    }

    //We even do not call Update method of the repository.
    //Because an application service method is a 'unit of work' scope
    as default.

    //ABP automatically saves all changes when a 'unit of work' scope
    ends (without any exception).
}

public void CreateTask(CreateTaskInput input)
{
    //We can use Logger, it's defined in ApplicationService class.
    Logger.Info("Creating a task for input: " + input);

    //Creating a new Task entity with given input's properties
    var task = new Task { Description = input.Description };

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

    //Saving entity with standard Insert method of repositories.
    _taskRepository.Insert(task);
}
}
```

**TaskAppService** 类实现的是数据库仓储操作。这是通过使用构造器注入（**constructor injection**）来获取实例。**ASP.NET Boilerplate** 内部已经实现了依赖注入，这使我们能够自由的使用构造器注入或属性注入来获取对象实例（需要了解更多关于依赖注入的知识，详见 **ASP.NET Boilerplate** 文档）。

注意，这里我们通过注入 **IRepository** 接口来使用 **PersonRepository** 类，**ABP** 会自动的创建仓储和我们的实体类。如果接口 **IRepository** 默认提供的方法足够我们使用，就不需要自己创建仓储类。应用程序服务的方法通过 **DTOs** 来进行数据传输

的，这是一种很好的习惯，而且我确实建议这样做。但是你不是必须这样做，只要你可以处理将实体暴露给表示层的问题。

在实现 **GetTasks** 方法之前，我优先实现了 **GetAllWithPeople** 方法。这里返回了 **List**，但是在表现层中我需要的是 **List** 类型。**AutoMapper** 组件能够帮助我们自动的从 **Task** 对象类型转换成 **TaskDto** 类型。**GetTasksInput** 和 **GetTasksOutput** 两个特别的 **DTOs** 是专门为 **GetTasks** 方法定义的。

在 **UpdateTask** 中，重新从数据库中获取了 **Task** 对象（使用 **IRepository** 的 **Get** 方法获取），然后改变属性值到 **Task** 对象中。注意，这里我没有调用 **Update** 方法来进行数据存储；**ABP** 实现了工作单元模式，所有在应用程序服务的数据变更方法都是一个工作单元，因此在方法执行的最后都会被自动的应用于数据库中。

在 **CreateTask** 方法中，我简单的创建了一个新的 **Task** 对象，然后使用 **IRepository** 接口的 **insert** 方法将数据添加到数据库中。

**ABP** 的 **ApplicationService** 类包含了一些属性，能够很方便的开发应用程序服务。例如，**Logger** 属性用于日志记录，所以从 **ApplicationService** 创建的 **TaskAppService** 类，能够使用 **Logger** 属性。能够从 **TaskAppService** 衍生的这些可选属性必须要实现 **IApplicationService**（注意，**ITaskAppService** 继承了接口 **IApplicationService**）。

## 8.9 验证

**ABP** 自动的为应用程序服务方法输入数据提供验证，前提是入参实现了 **IInputDtp** 接口（或者直接实现 **IValidate** 接口）。

**CreateTask** 方法以 **CreateTaskInput** 作为参数：

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

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

这里，修饰符是 **Required**，你可以使用很多的 **Data Annotation attributes** 特性来进行验证。如果你想要自定义一些自己的验证，就像 **UpdateTaskInput** 类一样，你通过实现 **ICustomValidate** 接口来自定义验证。

```
public class UpdateTaskInput : IInputDto, ICustomValidate
{
    [Range(1, long.MaxValue)]
    public long TaskId { get; set; }

    public int? AssignedPersonId { get; set; }

    public TaskState? State { get; set; }

    public void AddValidationErrors(List<ValidationResult> results)
    {
        if (AssignedPersonId == null && State == null)
        {
            results.Add(new ValidationResult("Both of AssignedPersonId
and State can not be null in order to update a Task!", new[]
{ "AssignedPersonId", "State" }));
        }
    }

    public override string ToString()
    {
        return string.Format("[UpdateTask > TaskId = {0},
AssignedPersonId = {1}, State = {2}]", TaskId, AssignedPersonId,
State);
    }
}
```

**AddValidationErrors** 方法是你可以自定义验证的代码区域。



## 8.10 异常处理

请注意，我们没有做过任何的异常处理。Abp 会自动处理 **exception**，记录 **log**，返回恰当的错误给客户端（**Client**）。此外，在客户端处理错误消息并显示给用户。实际上，也可以在 **ASP.NET MVC** 和 **Web API** 的控制器中使用这种做法。由于通过 **Web API** 暴露 **TaskAppService**，因此我们不需要处理异常。请查看[异常处理](#)获取更多的信息。

## 8.11 构建 Web API 服务

我想要将应用程序服务（**application services**）暴露给远程客户端（**client**）。这样，**AngularJs** 应用就可以很方便的用 **AJAX** 调用服务（**service**）方法。

Abp 提供了一个自动的方式将应用程序服务（**application services**）方法暴露给 **ASP.NET Web API**。我只需要如下那样使用 **DynamicApiControllerBuilder**：

```
DynamicApiControllerBuilder.ForAll<IApplcationService>(Assembly.GetAss  
sembly(typeof (SimpleTaskSystemApplicationModule)),  
"tasksystem").Build();
```

对于这个例子，Abp 会找到 **Application** 层中所有继承于 **IApplcationService** 的接口，并为每个 **application** 服务类创建一个 **Web API** 控制器（**Controller**）。接下来我们将看看如何通过 **AJAX** 调用这些服务（**services**）。

## 8.12 开发单页面应用（SPA）

接下来我们实现一个单页面的应用，**AngularJs** 是构建单页面应用最好的框架。Abp 框架提供了创建单页面应用的模板，我们可以很方便的创建一个基于 **AngularJs** 的单页面应用。Abp 单页面模板有两个渐进平滑效果的页面(**Home and About**)，并且使用 **Bootstrap** 作为 **html** 框架，因为它是响应式的。本地化支持英语和土耳其语（你也可以很方便的增加其他语言，或者删除这两个默认语言）。

首先，我们要使用 **AngularUI-Router** 改变 Abp 模板的路由，**AngularJs** 提供了基于状态的路由模式，在这个例子中，我们有两个视图，**task list** 任务列表和 **new**

**task** 新建任务。因此，我们首先要通过 **app.js** 定义路由。

```
app.config([
    '$stateProvider', '$urlRouterProvider',
    function ($stateProvider, $urlRouterProvider) {
        $urlRouterProvider.otherwise('/');
        $stateProvider
            .state('tasklist', {
                url: '/',
                templateUrl: '/App/Main/views/task/list.cshtml',
                menu: 'TaskList' //Matches to name of 'TaskList' menu in
SimpleTaskSystemNavigationProvider
            })
            .state('newtask', {
                url: '/new',
                templateUrl: '/App/Main/views/task/new.cshtml',
                menu: 'NewTask' //Matches to name of 'NewTask' menu in
SimpleTaskSystemNavigationProvider
            });
    }
]);
```

**app.js** 是最关键的 **JavaScript** 文件，用来配置应用的启动。需要注意的是，在这里我们使用的是 **cshtml** 文件格式作为视图。通常情况下，在 **AngularJs** 中我们使用 **html** 文件作为视图(译者注：大多数 **webapp** 也是 **html** 视图)。在 **Abp** 框架中，我们可以使用 **cshtml** 作为前端视图。这样的话呢，我们可以使用强大的 **razor** 视图引擎来创建 **html** 页面。

**Abp** 框架提供了基本的功能，可以很方便的为应用创建菜单项。我们可以使用 **c#**和 **JavaScript** 创建菜单，也可以两者结合。**SimpleTaskSystemNavigationProvider** 这个类用来创建菜单，**header.js/header.cshtml** 文件使用 **AngularJs** 显示菜单，**header.cshtml** 是菜单的视图，**header.js** 是菜单的脚本。

第一步：我们先创建一个显示任务列表的 **Angular** 控制器。

```
(function() {
    var app = angular.module('app');
```

```
var controllerId = 'sts.views.task.list';
app.controller(controllerId, [
    '$scope', 'Abp.services.tasksystem.task',
    function($scope, taskService) {
        var vm = this;

        vm.localize = Abp.localization.getSource('SimpleTaskSystem');

        vm.tasks = [];

        $scope.selectedTaskState = 0;

        $scope.$watch('selectedTaskState', function(value) {
            vm.refreshTasks();
        });

        vm.refreshTasks = function() {
            Abp.ui.setBusy( //页面加载之前，显示一个等待的图标
                null,
                taskService.getTasks({ //通过 JavaScript 直接调用应用层服务器端的方法
                    state: $scope.selectedTaskState > 0 ?
$scope.selectedTaskState : null
                }).success(function(data) {
                    vm.tasks = data.tasks;
                })
            );
        };

        vm.changeTaskState = function(task) {
            var newState;

            if (task.state == 1) {
                newState = 2; //任务完成
            }
        };
    }
]);
```

```
        } else {  
            newState = 1; //任务激活  
        }  
  
        taskService.updateTask({  
            taskId: task.id,  
            state: newState  
        }).success(function() {  
            task.state = newState;  
            Abp.notify.info(vm.localize('TaskUpdatedMessage'));  
        });  
    };  
  
    vm.getTaskCountText = function() {  
        return Abp.utils.formatString(vm.localize('Xtasks'),  
vm.tasks.length);  
    };  
}  
});  
})();
```

我们定义的控制器的名字是“**sts.views.task.list**”，这是我个人的习惯，当然你也可以定义一个简单的名字，比如：**ListController**。**AngularJs** 也可以使用依赖注入，在上面这个例子，我们注入**\$scope**和 **Abp.services.tasksystem.task**，第一个是**Angular** 范围的变量，第二个是自动创建的 **ITaskAppService**（在 **webapi** 层创建的服务接口实现）的客户端 **JavaScript** 代理服务。

**vm.taks** 是要显示在视图中的任务列表，**vm.refreshTasks** 方法使用 **taskService** 得到任务列表后填充这个列表。当 **selectedTaskState** 任务状态改变的时候这个方法被调用。（observed using **\$scope.\$watch**）

就像上面我们看到的，调用服务器端的方法非常直接简单！这就是 **Abp** 框架牛逼的地方，它可以生成 **Web API** 层和用于同 **Web API** 层交互的 **JavaScript** 代理层。这样的话，我们就可以在客户端调用应用层的服务，就像调用一个简单的 **JavaScript** 一样！在 **AngularJs** 下，它是完全智能的。（使用 **Angular's \$http service**）

下面让我们看一下任务列表视图的代码：

```
<div class="panel panel-default" ng-controller="sts.views.task.list as vm">

    <div class="panel-heading" style="position: relative;">

        <div class="row">

            <!-- Title -->

            <h3 class="panel-title col-xs-6">

                @L("TaskList") - <span>{{vm.getTaskCountText()}}</span>

            </h3>

            <!-- Task state combobox -->

            <div class="col-xs-6 text-right">

                <select ng-model="selectedTaskState">

                    <option value="0">@L("AllTasks")</option>

                    <option value="1">@L("ActiveTasks")</option>

                    <option value="2">@L("CompletedTasks")</option>

                </select>

            </div>

        </div>

    </div>

    <!-- Task list -->

    <ul class="list-group" ng-repeat="task in vm.tasks">

        <div class="list-group-item">

            <span class="task-state-icon glyphicon" ng-
click="vm.changeTaskState(task)" ng-class="{ 'glyphicon-minus':
task.state == 1, 'glyphicon-ok': task.state == 2 }"></span>

            <span ng-class="{ 'task-description-active': task.state == 1,
'task-description-completed': task.state ==
2 }">{{task.description}}</span>

            <br />

            <span ng-show="task.assignedPersonId > 0">
```

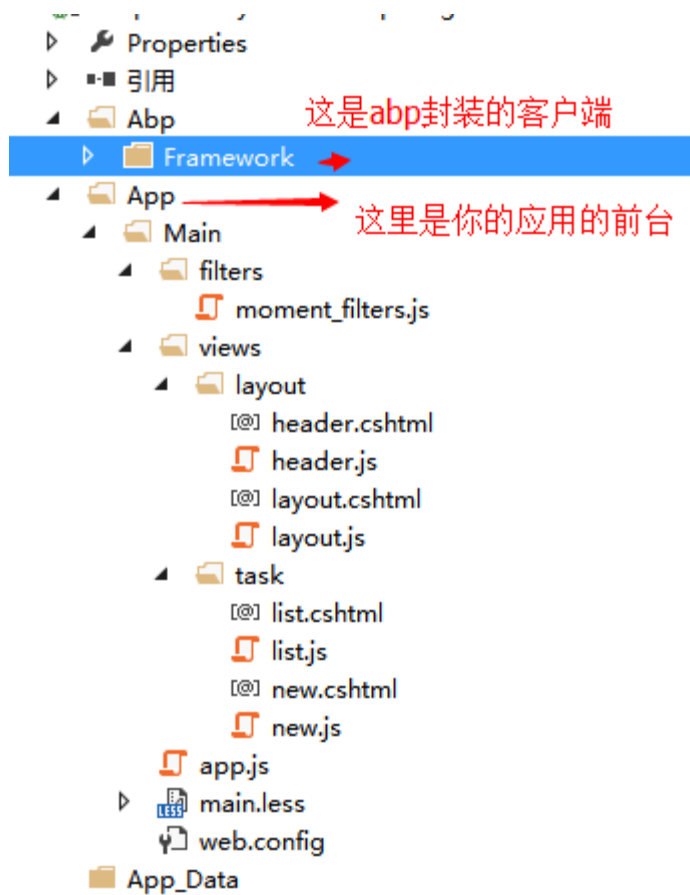
```
        <span class="task-  
assignedto">{{task.assignedPersonName}}</span>  
    </span>  
    <span class="task-creationtime">{{task.creationTime}}</span>  
</div>  
</ul>  
</div>
```

**ng-controller** 属性（第一行代码）绑定控制器到视图，**@L("TaskList")** 得到本地化的文本，这是可能的，因为它是一个 **cshtml** 视图。

**ng-model** 绑定组合框和 **JavaScript** 变量，当列表框改变，变量就会改变；当变量改变，列表框也会改变。

**ng-repeat** 是 **AngularJs** 的一个指令，用来为数组中的每一个值来渲染同一个 **html**，当数组改变的时候（比如我们这个例子，当添加了一条记录），自动反应这种变化到视图了。这就 **AngularJs** 是强大的特性之一。

译者注：**Abp** 到底是如何实现简单调用服务器端方法的呢？大家可以打开一个单页面应用的模板，在 **web** 层下有两个目录，如图:大家一看就明白了。



## 8.13 本地化

Abp 提供了一个灵活又强大的本地化系统。你可以使用 XML 文件或者资源（Resource）文件作为本地化资源。你也可以自定义本地化资源。请查看文档获取更多信息。在本示例程序中，我将使用 XML 文件作为本地化资源（在 Web 项目的 Localization 文件夹下）。

```
<?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="AllTasks" value="All tasks" />
    <text name="ActiveTasks" value="Active tasks" />
  </texts>
</localizationDictionary>
```

```
<text name="CompletedTasks" value="Completed tasks" />
<text name="TaskDescription" value="Task description" />
<text name="EnterDescriptionHere" value="Task description" />
<text name="AssignTo" value="Assign to" />
<text name="SelectPerson" value="Select person" />
<text name="CreateTheTask" value="Create the task" />
<text name="TaskUpdatedMessage" value="Task has been successfully
updated." />
<text name="TaskCreatedMessage" value="Task {0} has been created
successfully." />
</texts>
</localizationDictionary>
```

(翻译: 上海-半冷、天道)

<谢谢观看, 全文完结>